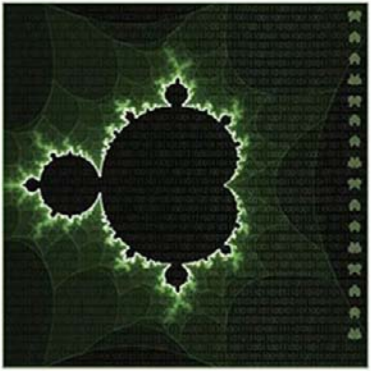
THOMSON

EMERGENCE IN GAMES

- Provides a detailed, theoretical foundation for understanding emergence in games
- Offers a practical approach to implementing emergence in games
- Defines the next step in game development—a more realistic, open, and natural interaction and behavior in game worlds



GAME DEVELOPMENT SERIES

PENNY SWEETSER

Emergence in Games

Penny Sweetser

Charles River Media
A part of Course Technology, Cengage Learning

COURSE TECHNOLOGY
CENGAGE Learning™

Australia • Brazil • Japan • Korea • Mexico • Singapore • Spain • United Kingdom • United States

COURSE TECHNOLOGY
CENGAGE Learning™

Publisher and General Manager, Course Technology PTR: Stacy L. Hiquet

Associate Director of Marketing: Sarah Panella

Manager of Editorial Services: Heather Talbot

Marketing Manager: Jordan Casey

Senior Acquisitions Editor: Emi Smith

Project/Copy Editor: Kezia Endsley

Technical Reviewer: Iain McManus

CRM Editorial Services Coordinator: Jen Blaney

Interior Layout Tech: Judith Littlefield

Cover Designer: Tyler Creative Services

CD-ROM Producer: Brandon Penticuff

Indexer: Valerie Haynes Perry

Proofreader: Michael Beady

Library of Congress Control Number: 2007939355
ISBN-13: 978-1-58450-551-8
ISBN-10: 1-58450-551-6
eISBN-10: 1-58450-605-9

Course Technology
25 Thomson Place
Boston, MA 02210
USA

Printed in the United States of America
1 2 3 4 5 6 7 11 10 09 08

I would like to dedicate this book to my partner,
Peter Philip Tadeusz Surawski,

as well as my family, Bill, Gay, Terry, Sean, and Jane Sweetser

This page intentionally left blank

Contents

Acknowledgments ix
About the Author xi

1 Introduction 1

Emergence 2
Emergent Gameplay 3
Emergence in Games 5
Who This Book Is For 15
How This Book Is Organized 15

2 Emergence 19

Complex Systems 20
Complexity 28
Scientific Approaches 29
Chaos Theory 32
Artificial Life 35
Emergence 39
Summary 41
Additional Reading 41
Class Exercises 42

3 Playing Games 43

Player Interaction 43
The Evolution of Gameplay 54
What Players Want 64
Future of Gameplay 74
Summary 77
Class Exercises 77

4 Emergence in Games 79
Board Games 80
Game Worlds 83
Characters and Agents 88
Emergent Narrative 94
Social Emergence 102
Developing for Emergence 108
Emergent Games 112
Summary 113
Class Exercises 114

5 Techniques for Emergence 117
Linear Techniques 118
Approximate Reasoning 126
Machine Learning 132
Complex Systems 144
Artificial Life 149
Choosing a Technique 163
Summary 165
Class Exercises 167

6 Game Worlds 169
Active Game Environment 170
Property-Based Objects 201
Emergent Game Worlds 227
Summary 231
Class Exercises 232

7 Characters and Agents 233
Sensing 234
Acting 242
Summary 304
Class Exercises 306

8 Emergent Narrative 307

Narrative Structure 308
Narrative Elements 312
Summary 354
Class Exercises 355

9 Social Emergence 357

Economies 358
Social Structures 384
Communities 396
Artificial Social Networks 400
Summary 407
Class Exercises 408

10 Conclusion 411

Emergent Game Worlds 412
Emergent Characters and Agents 413
Emergent Narrative 414
Social Emergence 416
Developers 417
Players 417
The Future of Gameplay Is Emerging 419

Appendix A Bibliography 421

Appendix B Glossary 431

Index 439

This page intentionally left blank

Acknowledgments

I would first and foremost like to thank my partner, Peter Surawski, for the help and support he has given me in writing this book. Not only has he put up with me during the process (and the several years of research that proceeded), but he has also contributed substantially to the book by helping me to create images, as well as acting as a sounding board for ideas and passages. I would also like to thank my mother, the English teacher, for reading most of the book and providing much-needed feedback and corrections.

I would also like to acknowledge the support, feedback, and encouragement of many of my friends and colleagues. In particular, I would like to thank Michelle McPartland, Adam Bryant, Jeff van Dyck, Kim Sellentin, Anne Ozdowska, George Fidler, Daniel Lehtonen, and Richard Lagarto. Their feedback and wisdom greatly helped to improve the quality of the book.

I would like to sincerely thank the game development experts who kindly contributed interviews for the book—Richard Evans, Craig Reynolds, Jeff van Dyck, and Brendan Rogers. Their insights, wisdom, and expertise provided depth and richness to the book from their varied perspectives. I am also grateful to Maryse Alvis who helped me in setting up interviews with developers at Electronic Arts.

I must also thank everyone who contributed to my research and Ph.D. work, which formed the basis for this book. I thank my supervisors, Prof. Janet Wiles and Dr. Peta Wyeth, for their feedback and direction. I also thank Penny Drennan, Dr. Daniel Johnson, and Jane Sweetser for their feedback, support, and contributions to my research work.

I'd also like to thank Charles River Media and Thomson Learning for giving me the opportunity to write this book. I'd particularly like to acknowledge all the editors who contributed to the process and the final product, including Jennifer Blaney, Emi Smith, Jenifer Niles, Kezia Endsley, and Iain McManus.

Finally, I would like to thank the many game developers who have inspired me by their creative genius and outstanding advances in video game technology. Particularly, I would like to thank Doug Church, who first inspired me to pursue emergence in games. I would also like to thank the masters of game design, who continually push the boundaries of what is possible to create what is extraordinary in games, including Will Wright, Warren Spector, and Peter Molyneux.

About the Author

Dr. Penny Sweetser completed her Ph.D. on "An Emergent Approach to Game Design—Development and Play" in 2006. Her research focused on player enjoyment in games and ways to enhance enjoyment through new technologies. During her research, she developed a model for player enjoyment in games, *GameFlow*, which has been explored and extended in subsequent research and applied in the development of computer games. She is a strong advocate of the player's experience and creating more player freedom and control in games.

In 2005, she started working for The Creative Assembly, Australia, where she worked as a designer on *Medieval II: Total War* and *Medieval II: Total War Kingdoms*. Prior to working at The Creative Assembly, Penny lectured on games design and tutored software engineering, artificial intelligence, and human-computer interaction at The University of Queensland. Penny has also worked as a research assistant for the Australasian CRC for Interaction Design and the University of Queensland Usability Laboratory.

Penny is an enthusiastic game player and enjoys playing games of all genres. She particularly enjoys strategy games, role-playing games, and first-person shooters. She also likes all things sci-fi and horror. Some of her favorite games of all time include *StarCraft*, *Vampire: The Masquerade—Bloodlines*, *The Elder Scrolls III: Morrowind*, *Might & Magic 7*, *F.E.A.R.*, *Bioshock*, and *Age of Mythology*.

This page intentionally left blank

1 Introduction

In This Chapter

- Emergence
- Emergent Gameplay
- Emergence in Games
- Who This Book Is For
- How This Book Is Organized

Emergence in games is a topic that has garnered much debate in recent years. Developers and players speculate that it would be fantastic if we could achieve gameplay that is open and natural, where players can choose their own strategies and the gameplay is limited only by the player's imagination and creativity. In recent years, games have taken steps toward this fantasy with emergent interactions made possible by Valve's Source engine and the availability of advanced physics middleware, as well as emergent gameplay achieved in games such as *The Sims* and *SimCity*. However, developers quickly realize the trade-off between emergence and design. Giving the players control has meant letting go of their own.

Despite all the recent attention, the concept of emergence is still quite ambiguous and undefined in games. Many people think it would be great to have emergent gameplay, but what does this mean and how do game developers achieve it? Throughout this book, I define *emergence* and examine the concept from various perspectives. I also provide many ideas and examples of how to incorporate emergence into current games. This book aims to serve as food for thought and a starting place for growing a wealth of knowledge, understanding, research, and applications for emergence in games. The future of game development lies in more open, interactive, and emergent gameplay. The future starts now.

Emergence

The concept of *emergence* describes the properties, behaviors, and structure that occur at higher levels of a system, which are not present or predictable at lower levels. In biological, physical, and social systems, there is the potential for something new to be created from simple entities interacting with their local environment and with each other. When these entities come together to form the whole, the whole is not merely a collection of these entities, it is something else entirely.

A brain is not a collection of neurons; it is a thinking machine. A human is not several connected systems; it is a sentient being. A society is not a group of co-located people; it is a powerful network capable of phenomenal behavior. The whole that is created from the collection is something new, with new properties, behavior, structure, and potential.

Local and Global Emergence

Emergence can occur at different levels and to varying degrees. An important distinction to make is the difference between local emergence and global emergence. *Local emergence* is the collective behavior that appears in small, localized parts of a system. *Global emergence* occurs when the collective behavior of the entities relates to the system as a whole. A system must be sufficiently rich, with highly interdependent entities, for global emergent behavior to exist, such as in brains, humans, and societies.

Elements and Rules of Emergence

Systems that exhibit emergence have a common set of elements and adhere to a common set of rules:

- Global phenomena emerge from local interactions of many simple entities
- There is no evidence of the global phenomena at the local level
- Global phenomena follow a different set of dynamics

Complex systems are distinguished from systems that are merely "complicated" by the possibility of emergence. Entities in complex systems do not merely coexist; they are interconnected and interdependent. In the case of global emergence, the whole is not only more than the sum of its parts; it is something new and different.

In Chapter 2, you'll explore the concepts of complex systems, chaos theory, artificial life, and emergence, to gain an appreciation for the fundamentals of emergence and the space of possibilities for emergence in games.

EMERGENT GAMEPLAY

Emergent gameplay is made possible by defining simple, global rules; behavior; and properties for game objects and their interaction in the game world and with the player. Emergent gameplay occurs when interactions between objects in the game world or the player's actions result in a second order of consequence that was not planned, or perhaps even predicted, by the game developers, yet the game behaves in a rational and acceptable way.

Emergent gameplay allows the game world to be more interactive and reactive, creating a wider range of possibilities for actions, strategies, and gameplay. Local emergent gameplay occurs when a section of a game allows for new behavior that does not have knock-on effects (or greater consequences) for the rest of the game. Global emergent gameplay occurs when the simple low-level rules and properties of game objects interact to create new, high-level gameplay that alters how the game as a whole plays out.

Emergent game systems empower players by putting them center stage, giving them the freedom to experiment, greater control over the game, a sense of agency, and less of a feeling of uncovering a path set for them by the designers. Consequently, the game can be more satisfying and interesting for the players. Emergent games also have high replayability, because each time the players play the game, they make different decisions, which change the game as a whole and result in different possibilities. In Chapter 3, I discuss the key elements of player interaction in games, review the history of gameplay from a player interaction perspective, examine the elements of player enjoyment, and look to the future of game development.

LEVELS OF EMERGENCE

The emergence that has been possible in previous games has been quite limited. Games could potentially allow the players to play the game in a way that was not designed or implemented by the game developer, but that works nonetheless. There are three potential orders (or levels) of emergence in games. These levels can be referred to as first-order, second-order, and third-order emergence.

First Order

First-order emergence in games occurs when local interactions have knock-on or chain reaction effects. The player's actions spread throughout the game world, affecting not only the immediate target, but also the nearby elements of the game world. First-order emergence is becoming commonplace in games, especially since the advent of Valve's Source engine and other advanced physics middleware. The Source engine is a 3D games engine developed by Valve Corporation. Valve's

Source engine features an advanced physics system, which allows flexible and realistic physical and environmental modeling in games. Valve's game *Half-Life 2* (see Figure 1.1) uses the Source engine to create realistic environments that allow complex interactions with game world objects. Games that use property-based objects, such as *Half-Life 2*, allow for a wide range of interactions and local knock-on effects.

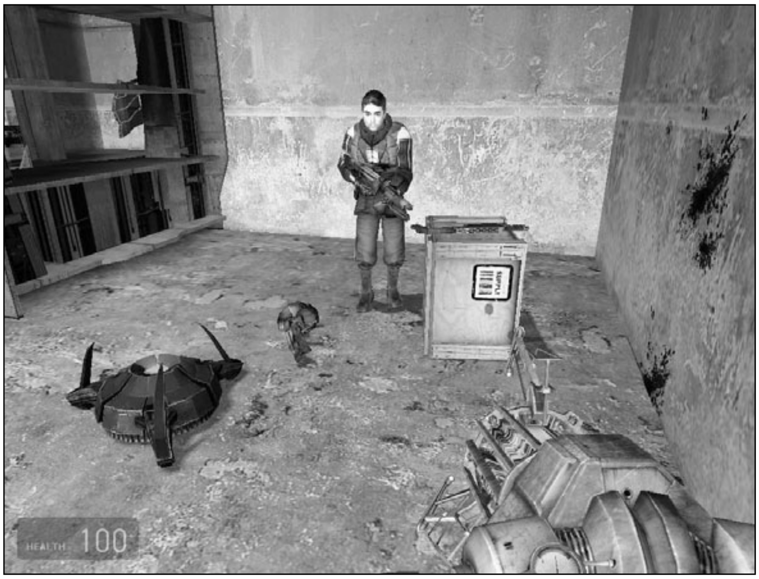


FIGURE 1.1 The Source engine used in *Half-Life 2* allows realistic physics and complex interactions. © Valve Corporation. Used with permission.

Second Order

Second-order emergence occurs when players use the basic elements of a game environment to form their own strategies and solve problems in new ways. Game characters might also be able to use or combine their basic actions to exhibit new behaviors or strategies. These types of emergence are still local effects, as they have a limited range of effect and do not impact the game as a whole. However, they allow considerably more player freedom and creativity and change how individual parts of the game play out.

Third Order

Third-order emergence pertains to the game as a whole, where the emergence occurs on a global scale. The boundaries of the game are suitably flexible to allow the players to carve new and unique paths through the game. New gameplay occurs that changes the game as a whole. The game allows for divergence in narrative, game flow, character interactions, or social systems.

Third-order emergence is the holy grail of emergence in games, but by no means the only type of value. Rather, the key is to develop emergence that will improve the player's experience of the game in some way, and never for its own sake. The use of the simplest method that will achieve the desired results is always the best.

Emergence in Games

Emergence can play a part in games in various ways. In Chapter 4, I outline the major components, including game worlds, characters and agents, emergent narrative, and social emergence. I also identify and discuss some of the major concerns of game developers in developing for emergence. Each of the major components is explored in-depth in Chapters 6 to 9. Before I enter into the specifics of each area, I first take a look at some algorithms and techniques that you can use to create emergent behavior in your games in Chapter 5.

Techniques for Emergence

In Chapter 5, I discuss various programming techniques and algorithms from fuzzy logic, complex systems, artificial life, and machine learning that can be used to create emergence in games. I also outline some traditional techniques that are prevalent in current games. The design, application, and considerations of using these techniques in games are discussed. The basic techniques outlined in Chapter 5 are used as the foundation for the models and frameworks presented in later chapters. I also discuss the considerations of choosing the right technique for the right application.

Game Worlds

Game worlds are the possibility spaces of games. The space, terrain, objects, physics, and environmental effects dictate the possibilities for actions and interactions that compose and constrain the gameplay. The elements of the game world (for example, the weapons, chairs, walls, and enemies) are the basic elements of gameplay, similar to the board and pieces in chess. The laws of physics and rules of interaction are the game rules, which constrain the possibility space. Within this space are the allowable actions and interactions of the players. Creating emergent game worlds involves designing types of objects, interactions, and rules, rather than specific, localized gameplay.

Interactions in the game world are the foundation of the gameplay and the types of interactions depend on the game genre. In role-playing games, interactions include talking to characters, using spells or abilities, collecting items, gaining experience, and upgrading abilities. In real-time strategy games, interactions include training units, constructing buildings, collecting resources, upgrading, attacking, and defending. In first-person shooter games, the players can run, jump, duck, hide, kick, and shoot. The gameplay is made up of how the players use these basic interactions to solve problems, achieve goals, and advance through the game.

The key to creating emergent gameplay is to define a simple, general set of elements and rules that can give rise to a wide variety of interesting, challenging behaviors and interactions in varying situations. The simpler and more generalizable the rules, the easier they will be to test, tune, and perfect for emergent gameplay. The simplest solution that gives the desired results is always the best. As with any emergent system, the fundamental set of rules and elements stay constant, but their situation and configuration change over time. The sensitivity of the elements to changing situations and the interaction of the elements with each other and the players are what create emergent gameplay.

Game worlds can be divided into two fundamental components—environment and objects. The environment is the space, including boundaries such as terrain, sky, and walls, as well as the physical space (for example, air in an earth-based game or water in an underwater game). The game environment in most games is inert and unresponsive to players, objects, and events. Game objects are the entities that populate the game world. There are a wide variety of objects in game worlds, which vary by game genre. Characters and agents are even types of objects, which I will discuss later. Together, the environment and objects make up the game world and their properties and behavior determine the interactions that are possible.

Environment

The environment is the central component of an emergent game system; it defines the game world and the interactions that are possible within the world. The rules that are defined for the interactions within the environment itself dictate the rules that will apply to entities that exist in the environment, such as objects and agents. Therefore, defining the rules of behavior of the environment itself is a crucial step in developing a game world that facilitates emergent behavior.

The environment in most games is inert and unresponsive to player actions. Chapter 6 covers a framework for what I call an "active" game world, which can be used to model environmental systems, such as heat, pressure, and fluid flow in games. The Active Game World model uses simplified equations from thermodynamics, implemented with a cellular automaton (see Figure 1.2). The Active Game World model, based on simple interactions between cells of the environment, provides a foundation for emergent behavior to occur in game objects and agents, as well as in the environment itself.

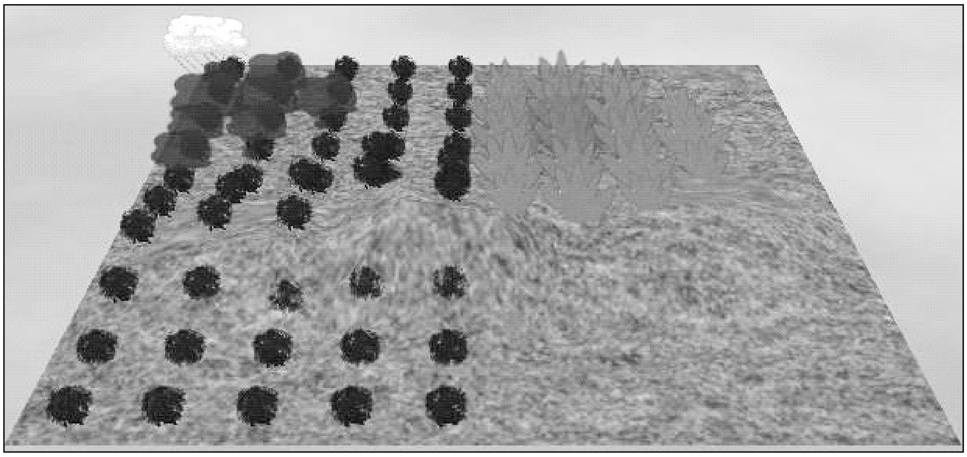


FIGURE 1.2 The Active Game World models physical systems with cellular automata.

Objects

Game objects are an integral part of any game world; they compose the major source of player interactions. Objects in games are numerous and varied, including weapons (for example, guns and swords) in first-person shooter games, quest items (for example, the holy grail or a diary) in role-playing games, and buildings (for example, barracks and factories) in strategy games. Each type of game object interacts with the game environment and with the players in different ways, which gives rise to interesting possibilities for actions for the players, but complicates the job of the game developer.

Some games have allowed more freedom and variation through property-based objects and rules for how the objects interact. Using a global design, the game objects behave more realistically and are more interactive, because they are encoded with types of behavior and rules for interacting, rather than specific interactions in specific situations. These objects afford emergent behavior and player interactions that were not necessarily foreseen by the developers.

Chapter 6 presents a framework for creating property-based game objects that can be integrated into the Active Game World model (see Figure 1.3). Objects are also imbued with high-level properties, based on their structure, to constrain the possible physical interactions of the objects. The high-level property tags that are attached to objects can be used to create affordances for interactions with the player and other objects. The resulting model is flexible and extensible, allowing the game world to respond consistently and realistically to a wide range of events and player actions in any situation in the game.

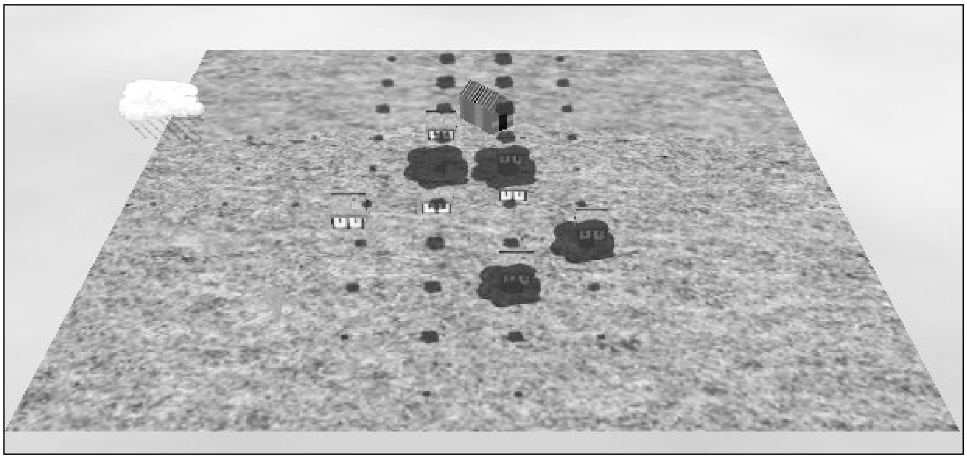


FIGURE 1.3 The Active Game World uses property-based game objects.

Characters and Agents

Characters and agents are important types of objects in game worlds, as they give the game life, story, and atmosphere. Characters and agents serve many purposes and hold varied positions in games, which contributes to making the game world rich, interesting, and complex. For example, strategy games include units (for example, marines) that the players control and role-playing games include characters that fill a wide range of different roles in society, from kings to goblins. More than anything else in the game world, players identify with and expect lifelike behavior from game characters.

Agents are a vital ingredient in creating an emergent game world. Introducing entities that have a choice of how to react to the changing environment amplifies the variation and unpredictability of a system. Reactive agents can extend emergent behavior and gameplay by adding a new level of complexity to the game world. As agents can choose how to react to the environment, they can actively change the state of the world in ways that might not have occurred without their intervention. Also, differences between individual agents and types of agents, such as composition, structure, goals, personality, and so on, can add variation and complexity. Not only can agents choose how to react to a given situation, different agents will choose to react in varying ways in the same situation.

Characters and agents can create emergence in games by being given an awareness of their environment and an ability to react to the changing state of the environment. The agents then become part of the living system of the game, which they sense, react to, and alter. Agents can be given the ability to respond to the player and other agents, events, and conditions in their environment, as well as their own goals and motivations, by having a model of their environment and a set of rules for reacting. Characters and agents that follow simple rules for behavior, taking into account the complex environment around them, will become emergent entities in the game world.

Sensing

The agents in most games rely heavily on the prior knowledge of their designers and little on their current situation. Many agents in games, such as units in strategy games and villagers in role-playing games, do not react to the environment in any way. Giving an agent an awareness of its environment and a way to sense and model the situation is the most crucial step in creating reactive, dynamic, and emergent behavior. The more information and intelligence embedded into the environment, the simpler the agents themselves can become.

The ideal framework for facilitating emergent agent behavior is to have simple agents in a complex environment. The emergence comes from the interactions between agents, between the agents and the player, and the collective interactions of the agents with the game world. In order to achieve this, the agents must be given a way to sense and model their environment. Some common approaches to sensing game environments are probing, broadcasting, and influence mapping. A framework for using each of these approaches in an emergent game system is presented in Chapter 7.

Acting

After the agent has sensed its environment and has an understanding of its situation, it must choose an action. Even if the agent has a sophisticated world model, if it fails to act or react appropriately, it will appear lifeless and unintelligent. There are a wide range of specific actions that agents are required to take in game worlds, which vary depending on game genre. There are two major types of actions that agents are required to take—individual actions and group actions. Individual actions require the agent to behave autonomously and make decisions based on its own situation and needs. Group actions require the agent to play a role in a group of agents, which involves cooperation and coordination.

Individual

Agents that act individually are usually game characters or enemies. In first-person shooter games, a large proportion of the agents are there to fight the player. The primary actions of these agents are to run, jump, dodge, hide, and shoot enemies. In role-playing games, agents include friendly and enemy characters, as well as monsters and animals. The actions of these agents include talking, fighting, walking, and appearing to follow normal lives and routines. In strategy games, individual agents (or units) must move, attack, guard, and hold positions. Agents in sports games must move around the field or court, score goals, pass, tackle, and so on. The cars in racing games drive around the track, dodge or ram other cars, and sometimes perform stunts. The most common actions for agents in all of these types of games are movement and decision-making. Chapter 7 discusses how characters can use their environmental model to guide their movement and presents a simple, flexible, general-purpose framework that you can use for agent decision-making (see Figure 1.4).

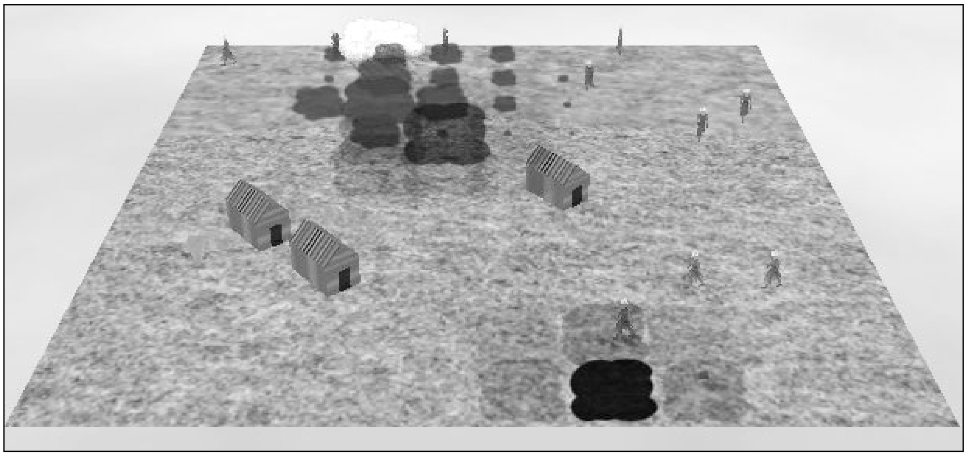


FIGURE 1.4 Reactive agents in the Active Game World.

Group

Many games have groups of agents that must interact, coordinate, and cooperate. This is particularly important in team-based games, such as strategy games and sports games. When there are two or more sides fighting or competing, the agents must cooperate in an organized way to have any chance of success. Emergence has a lot of potential to improve group behavior, with a focus on self-organization, rather than top-down orchestration. The two most important group actions in games are group movement and tactics. Chapter 7 discusses methods for achieving emergent group movement in games using agent-based steering behaviors and presents a framework for creating emergent group tactics using an agent-based approach (see Figure 1.5).

Emergent Narrative

A game's narrative is the story that is being told, uncovered, or created as the players make their way through the game. This story might take the form of a single, lin-

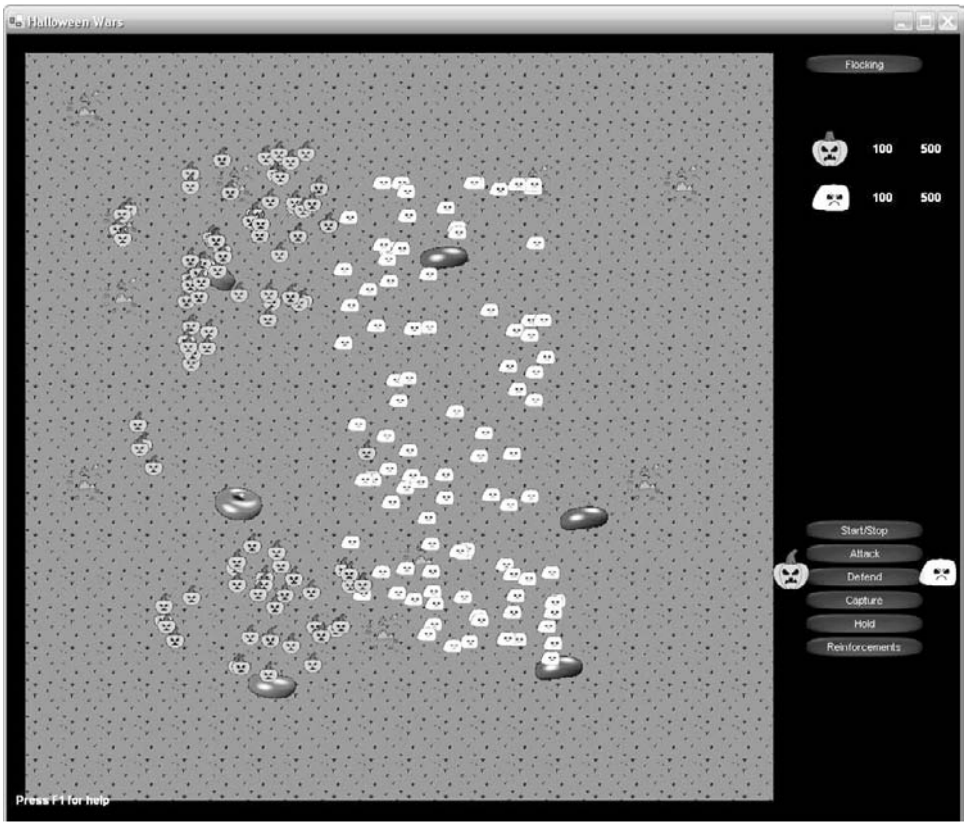


FIGURE 1.5 Emergent group tactics in *Halloween Wars.*

ear plot that is divulged to the players at selected points in time. Alternatively, it could be the deep, underlying truth of the game world that requires the players to solve puzzles and investigate the world. It could also be the product of the players' interactions in the game world—the internal story that the players create about their character or challenges as they play the game. No matter the format of the narrative, it is central to the enjoyment and understanding of all games, even games that do not have a story. In creating emergent narrative, the developer is tailoring the narrative to the players' experience and putting them center stage.

Narrative Structure

If you examine forms of narrative in games from the player's perspective, there are three main categories that can be identified. The first is the traditional "player as receiver" model that is drawn from other forms of storytelling, such as movies and books. In this form, the story is entirely prewritten and is simply transmitted to the players. The players receive the story and have no potential to affect the outcome or progression. A similar type of narrative is "player as discoverer," in which the story is embedded in the game world and the players must uncover the pre-existing plot. The third, and considerably different form, is "player as creator," which involves the players actively creating and affecting the story as a product of their actions and interactions. Player as creator narrative is emergent. Some, or all, of the story is a product of the players' interactions in the game world, interactions between objects or characters in the game world, and knock-on effects. The narrative is not predetermined and scripted; it emerges from interactions between entities in the game world. Chapter 8 explains a few simple ways to achieve emergent narrative in games, using the narrative elements of storyline and conversation.

Narrative Elements

There are two key elements that can be used to create narrative in games—storyline and conversation. Narrative is formed by telling stories about events, people, and places. A player's actions in a game can form a kind of internal narrative, but it is not until the retelling that it becomes a story. The storyline is the overarching plot, as well as subplots, that play out in the game. As discussed in the previous section, the storyline can be received, discovered, or created by the player. Conversations are a more informal, continuous form of narrative. The player can engage in conversations with various characters throughout the game, or observe conversations between other characters, to gain small pieces of information about events, people, and places in the game. By allowing emergence in storylines and conversations in games, you can create emergent narrative.

Storyline

There are several components of storylines in games that can be used to create a compelling narrative. These components are backstory, storytelling, story creation, and post-game narrative. A backstory presents events that occurred prior to the start of the game and can be used to establish setting, character, and motivation. Storytelling is used throughout a game to impart further information about the plot or game world to the players, usually via cutscenes. Story creation is the more interactive form of storytelling in which the players perform certain actions, such as completing missions or quests, to create subplots or advance the overall plot. Finally, post-game narrative is storytelling that occurs after the game is completed, which can be used to create a story out of a player's journey through the game. Each of these components can be used to create narrative in an emergent game. Chapter 8 provides a framework for developing an emergent storyline using these components.

Conversation

Chapter 8 describes a conversation system for enabling emergent narrative through conversation. The system involves the use of a core set of variables that affect a character's conversation and a set of rules for how the conversation is affected. The rules and variables of the system are simple, but allow emergent conversations in the context of a complex game world. A simple conversation system that is sensitive to the state of the game world, characters, and player can create emergent conversations between the player and characters, or between game characters.

SOCIAL EMERGENCE

Of all the forms of emergence in games, social emergence is by far the most complex and unpredictable in current games. When millions of people come together to play popular massively multiplayer online role-playing games, such as *World of Warcraft* (see Figure 1.6) or *Lineage*, the result is comparable to the divergence and complexity of a large city. Rather than trying to find ways of creating emergent social systems in games, it is more a matter of trying to understand, model, constrain, and support the complexities that arise naturally from human interactions. In order to do this, game developers must draw on psychology, sociology, economics, and even law.

[2. Trade] [Talexis]: lol
[2. Trade] [Remeko]: lol nvm then
[2. Trade] [Talexis]: h8 it when i'm a couple S or G short >.<
[1. General] [Bigstank]: nice people here unlike BB thats forsure
[2. Trade] [Rubedo]: LF Rogue to open a [Khorium Lockbox] for me
[2. Trade] [Hampter]: LOOKING for somone to CRAFT [Spellstrike Pants] pst

FIGURE 1.6 The massively multiplayer online role-playing game *World of Warcraft.* World of WarCraft® images provided courtesy of Blizzard Entertainment, Inc.

The most prevalent forms of social emergence in games include emergent economies, social structures, and communities. Chapter 9 draws on the lessons learned from developers of major online games, as well as research into the complexities of these worlds from the perspectives of psychology, sociology, economics, and law. The result is an exploration of the major highlights and issues, as well as guidelines and suggestions for supporting and harnessing these forms of social emergence in games. Finally, you'll look at how artificial social networks can be created in single-player games and the emergence of artificial social communities.

Economies

The virtual economies in many *massively multiplayer online games* (games with thousands or millions of players playing simultaneously in persistent online worlds) have become as complex, intricate, and difficult to manage as real-world economies. Virtual economies are emergent systems that change dynamically with supply and demand, based on the trading patterns of the world's inhabitants. They share many characteristics with real-world economies, such as trading and banking, but are also subject to many of the same problems, such as inflation and gambling. Chapter 9 examines the emergence of economies and trading in massively multiplayer games, as well as the considerations and potential for developers.

Social Structures

The individual interactions, motivations, and behavior of players in massively multiplayer online games give rise to complex social structures that share many common elements with real societies. Emergent social structures in online games include governments and political parties that form around common beliefs, desires, or goals. Virtual governments even institute laws and punish law-breakers. Virtual crimes are becoming more commonplace and varied, with thefts, assaults, prostitution, and bullying. Even online mafia has emerged in some games, where powerful players threaten new players into giving them protection money, as well as carrying out organized crime. Chapter 9 discusses the emergence of social structures in games and the considerations for game developers.

Communities

Players invest large amounts of their time into games and form strong social bonds with other players over time. Communities of like-minded players come together inside and outside of game worlds, forming guilds, forums, competition ladders, and mod communities. Supporting the development and continuance of these communities is important for developers in sustaining interest in their games. Social bonds between players can keep them playing the game far longer than almost any gameplay mechanism. Chapter 9 discusses the various forms of game communities and what developers can do to encourage and support these communities.

Artificial Social Networks

The emergence of social structures and communities in massively multiplayer games provides insight into the power of social dynamics in games, as well as inspiration for translating these dynamics into artificial social networks for single-player and multiplayer games. Giving game characters the ability to form social relationships with the player, as well as other game characters, has the potential to add more life and interaction to game worlds. Using social networks to determine the flow of information in game worlds and character behavior in social situations can allow for emergent social interactions and character behavior. Chapter 9 presents a model for social networks in games.

Who This Book Is For

This book is primarily written for game developers and future game developers who hope to extend their games to enhance the player experience by allowing emergent behavior and gameplay. It broaches questions from game design and game programming and a practical understanding of both these areas will allow the greatest benefit from this book. The book should be interesting and accessible for anyone in game development, but some concepts and examples are more advanced. If you are a game developer, researcher, or student of any level, you should be able to gain a greater understanding and appreciation for the importance, potential, and pitfalls of emergence in games from this book.

How This Book Is Organized

The aim of this book is to provide a grounded understanding of emergence in games for game developers, researchers, and students, as well as practical skills to create games with emergent gameplay. The knowledge and lessons in this book are founded in complex systems theory and research, and grounded in the past, present, and future of practical game development. Chapters 2 through 9 provide the necessary background knowledge, theory, game development applications, and practical skills to commence designing, implementing, or researching emergence in games. Chapter 10 provides a summary of the book, concluding remarks, and a discussion of the future of game design. A bibliography of relevant resources, including books, journals, conferences, Web sites, and games, is also included, along with a glossary of the key terms used throughout the book.

Theory

Chapters 2 to 5 provide the fundamental theory of complex systems, game development, and computer science that later chapters will build upon.

- *Chapter 2* provides a broad, interdisciplinary introduction to complex systems and emergence.
- *Chapter 3* explores the past, present, and future of game development from a player-centered perspective.
- *Chapter 4* overviews emergence in games and identifies key areas for more in-depth exploration in later chapters.
- *Chapter 5* explains several key techniques for creating emergence in games, which will be built upon in subsequent chapters.

PRACTICE

Chapters 6 to 9 take a practical approach to creating emergent gameplay in commercial games. Four key areas are identified and explored in detail: game worlds, characters and agents, emergent narrative, and social emergence.

- *Chapter 6* explains methods to create emergent gameplay via interactions in game worlds.
- *Chapter 7* explores emergence through game artificial intelligence, including characters and agents.
- *Chapter 8* delves into creating emergent narrative through storytelling and conversations in games.
- *Chapter 9* discusses emergent social communities, including communities inside game worlds and external game communities.

WHAT'S ON THE CD

ON THE CD

The accompanying CD-ROM includes demonstrations, color images, and source code that illustrate the examples and solutions presented in this book. Please refer to the Web site www.emergenceingames.com for updates. The contents of the CD-ROM are divided into:

- *Demonstrations*—Include code, data, and executables for examples presented in the book. Source code and listings from the book are included in subfolders by chapter number. Demos were compiled using MS Visual C++ 2003. Executables for each demo are also included.
- *Games*—A selection of related freeware games is included on the CD-ROM.
- *Images*—Color versions of the screenshots included in the book are provided on the CD-ROM in JPG format.
- *Papers*—A selection of related research papers is included on the CD-ROM.

Chapter Structure

Each chapter includes suggested readings that will provide further information, details, or examples of topics that are discussed. The keywords that are used in each section are also defined in the keyword summaries. Each chapter concludes with a set of class exercises that you can do on your own or that teachers can administer to their classes. These exercises are best used for group work in tutorials or labs.

Chapters 6 through 9 include interviews with industry experts in the areas of design, programming, research, and audio. These interviews provide specific examples of applications of emergence in games from varying industry perspectives, as well as tips for game developers and researchers. Examples and case studies are also illustrated with code or demonstrations where appropriate, which are included on the accompanying CD-ROM. The next chapter commences your journey into emergence in games with an exploration of the history and foundations of emergence and complex systems in the world at large.

ON THE CD

This page intentionally left blank

2 Emergence

In This Chapter

- Complex Systems
- Complexity
- Scientific Approaches
- Chaos Theory
- Artificial Life
- Emergence

Complex systems have been studied in computer science for at least 50 years. Before computers, they were studied in physics, biology, and even philosophy. In fact, it was Aristotle, over 2,000 years ago, who first recognized the profound concept that a whole can be more than the sum of its parts. The ideas are not new, but the implications are still as profound as ever, and the promise of emergence is still as enticing, especially when one contemplates the possibilities in the world of computer games. But before I get to games, let's first explore the concepts of complex systems, chaos theory, artificial life, and, of course, emergence, to truly understand the space of possibilities which we, as game developers of the future, are facing.

This chapter explores complex systems and their incarnation in physics, biology, and society. It also defines complexity and examines various scientific approaches to understanding complexity and complex systems. You will investigate chaos, order, complexity, and emergence at the edge of chaos. The chapter will then go on to discuss artificial life and agent-based systems. The chapter finally ends up at emergence, which is the end of the journey, but by this stage you will have a thorough appreciation of what it means and where it has come from. Emergence is the key to every system, theory, and methodology you will visit along the way.

Complex Systems

If something is *complex*, it is so complicated or intricate that it is hard to understand or deal with. *Complex* also refers to something that is composed of many interconnected parts, or has a complicated arrangement of parts. Both of these definitions are true of complex systems.

A *complex system* is a system that consists of many interconnected and interdependent parts. The parts themselves may be simple or complex, but the real complexity comes from their interaction. From the outside, the system appears convoluted and impossible to understand. But a closer look reveals that it is made up of many simpler components, each following a set of behaviors and interacting with its local environment.

The result of these behaviors and interactions is far more than what would be expected by examining a component in isolation. The collective behavior of the system is not equal to the simple sum of its parts—it is something that is dynamic, organic, and alive.

A common and familiar example of an extremely complex system is the human brain. Our brains consist of approximately 100 billion neurons and each of these neurons is connected to about one to ten thousand other neurons via synapses. A synapse is the channel for the transfer of chemical signals between neurons. When a neuron fires, it releases chemicals, called neurotransmitters, that diffuse across the synaptic cleft and interact with the receptors of connected neurons. These reactions can either have an excitatory or inhibitory effect on the receptor neuron, increasing or decreasing the chance that it will, in turn, fire and propagate the signal.

The role that the individual neuron has played is quite simple, depending on the chemicals that pass across the synapse to its receptors, it either exceeds the potential to fire and propagate the signal or it does not. But as an entire network of neurons that are continually firing or inhibiting signals, with such an enormous connectivity between neurons, the result is something that could not be imagined from the workings of a single component.

The human brain, capable of a plethora of thoughts, emotions, language, motor control, consciousness, and self-awareness, emerges, unexpectedly and inexplicably, from these simple components interacting with neighboring neurons in a seemingly disorganized manner (see Figure 2.1).

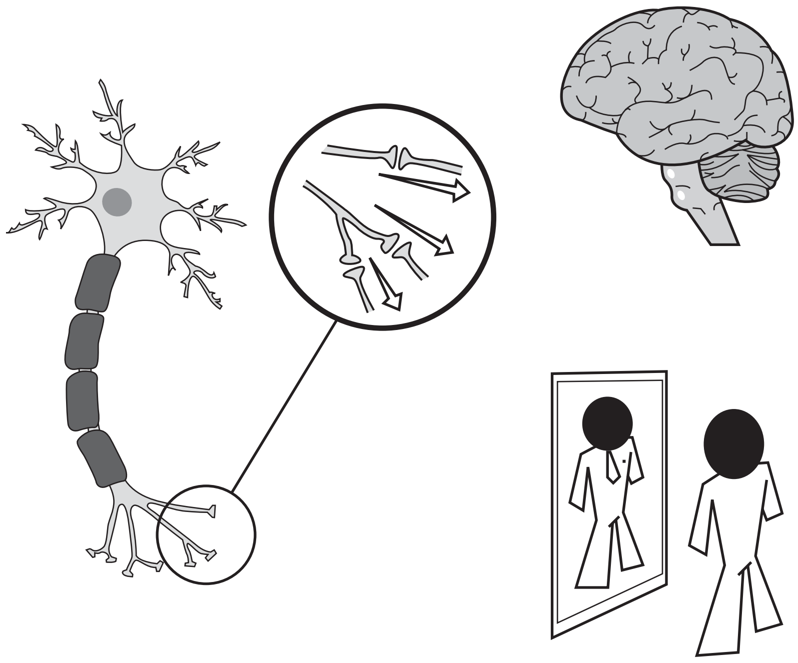


FIGURE 2.1 Neuron, synapses, brain, and consciousness.

A complex system is more than the sum of its parts, because it is as much about how the parts interact, interconnect, and affect each other as it is about the composition and behavior of the parts themselves. Simply looking at the parts in isolation and inferring the overall behavior of the system as an aggregate of these parts does not give an accurate description of the system.

The interaction of the parts is so fundamental to the system as a whole that the system cannot be described without describing its parts and the parts cannot be described without describing how they relate to each other. The complex behavior of the system is said to be *emergent*, it cannot be simply inferred by the behavior of its components.

The components of complex systems do not just interact, they are interdependent. Without the role that each component or entity plays in the system being fulfilled, the other parts either lose their emergent complexity or they may not be able to continue to function and exist. You can attempt to understand the degree of interdependence of these complex systems and their substructure by removing or modifying a part of the system and observing the effects on the rest of the system. This process is called *subdivision*.

Properties of Complex Systems

Most complex systems have a common set of properties. These properties include elements, interactions, formation, diversity, environment, and activities. Consider these properties in relation to the human brain (see Table 2.1):

- *Elements* are the basic components of the complex system. In the human brain, the elements are the neurons.
- *Interactions* occur between these elements to give rise to the overall complex behavior of the system. In the brain, the interactions are the synapses, or connections between neurons.
- The system and its components are formed by some process of *formation*. The pathways through the brain are formed by learning how to respond to the outside world.
- Complex systems have a *diverse* range of behaviors and states. Diversity in the brain comes from the connectivity of the neurons and the great many potential pathways that exist through the brain.
- Complex systems also exist within *environments* that they must respond to and interact with. The brain's environment is the body and its many systems (for example, organs, limbs, and eyes) that the brain must interact with in order to control functions and the behavior of the body. The brain must also perceive and synthesize stimuli from the outside world and learn how to respond and interact with the world.
- Finally, complex systems carry out *activities* in order to achieve certain objectives or for some purpose. The activities carried out by the brain are the aforementioned plethora of thoughts, feelings, movements, and interactions that people face every day.

TABLE 2.1 Properties of Complex Systems and Their Manifestation in the Brain

Property	**Description**	**Brain**
Elements	Basic components of the system	Neurons
Interactions	Interactions that take place between elements	Synapses
Formation	How the interactions and elements are formed	Learning
Diversity	A diverse range of behaviors and states	Potential pathways
Environment	The environment where the interactions occur	Body, world
Activities	The activities that are carried out by the system	Thought, behavior, motor control

Examples of Complex Systems

There are many examples of complex systems that surround us every day. Life, itself, is a complex system, as are people, animals, families, governments, planets, and galaxies. Most complex systems are, themselves, composed of other complex systems, which are also composed of complex systems, and so on. For the purpose of understanding them, you can roughly divide complex systems into three main categories—physical systems, biological systems, and social systems (see Table 2.2).

It is important to note, however, that these categories are very much overlapping and interwoven themselves. Social systems are made up of many biological organisms (humans), which are made up of smaller biological organisms (for example, organs, tissue, and cells), that are in turn made up of physical substances (for example, atoms and molecules) and governed by the laws of physics. Therefore, these categories, and their boundaries, are fuzzy at best.

TABLE 2.2 Complex Systems in Physics, Biology, and Society

Physical	**Biological**	**Social**
Gases	DNA	Governments
Crystals	Cells	Culture
Lasers	Organisms	Military
Weather	Brain	Economies
Fluids	Humans	Families
Glasses	Proteins	Corporations
Earthquakes	Ecosystems	Flocks
Fractals	Embryos	Traffic
Solids	Ant colony	Internet
Galactic structures	Life	Online games

Physical Systems

Physical systems are made up of the simplest elements—atoms, protons, electrons—obeying the universal laws of physics. Yet, from the interactions of these base elements following the fundamental laws of the universe, come the most complex, emergent, and dynamic systems and behaviors (see Table 2.2). Some of the more fascinating examples include global weather patterns, crystallization (for example, snowflakes), fluid dynamics, and the formation of galactic structures (galaxies).

Snowflakes are formed via crystallization, which is the process of solid crystal formation. Crystallization is a liquid-solid phase transition, in which a solid grows by adding material from an adjacent liquid environment (Gaylord & Nishidate, 1996). Crystallization includes two major events: nucleation and crystal growth. In nucleation, the solute molecules dispersed in the liquid gather into clusters and the atoms arrange themselves in a defined and periodic manner that determines the crystal structure.

Crystal growth is the subsequent growth of the nuclei. Nucleation and growth continue while supersaturation exists. Supersaturation determines the rate of nucleation and growth and depending on which is predominant, crystals with different sizes and shapes are formed. A simple cellular automaton (see Chapter 5) can be implemented to simulate the formation of snowflakes (see Figure 2.2).

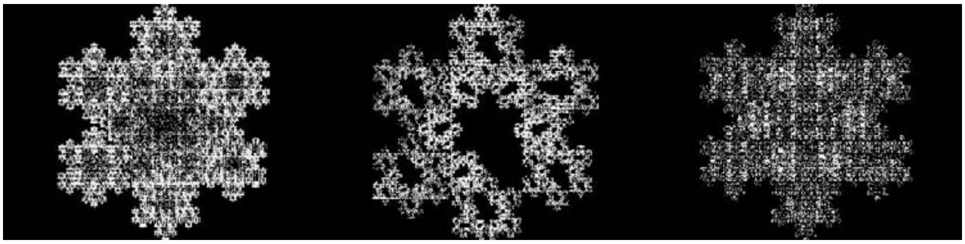


FIGURE 2.2 Simulated snowflakes.

Biological Systems

Life has the most profound, surprising, and diverse set of complex systems. From the DNA that encodes the proteins in every living organism, to the neurons that make up the brain and nervous system, to entire ecosystems comprised of thousands of different species living in equilibrium, life is the ultimate complex system.

Life is made up of nested complex systems, where it is possible to continuously drill down into one complex system and find many more. The world is a complex ecosystem, composed of many sub-ecosystems, such as forests, swamps, and oceans. Each sub-ecosystem is composed of a diverse set of organisms, from human beings to bacteria. Many organisms are composed of a variety of other systems and organisms, such as organs, nervous systems, circulatory systems, and so on, that must all function in harmony for life to continue. These systems are made up of tissues, cells, and molecules, and at the root of it all, the DNA that encodes the building blocks of life, proteins.

It is important to note that DNA does not encode the instructions for creating the organism, as you would assemble IKEA furniture. It has a much simpler function, which is only to encode proteins. The development of the organism is a reaction to the environment, such as the creation of a cell wall for protection or a cytoskeleton for structure. Over millions of years of reaction, reproduction, competition, and selection, life has evolved into the complex structures all around us. Life is composed of complex systems as it is made up of very simple elements reacting to their local environments in ways that create self-organized, complex structures with amazingly diverse properties and behaviors.

An interesting example of complex behavior in biological systems is via chemical signaling. Chemical signaling is an important process in communications between organisms in many communities, including animals, birds, insects, and bacteria. Chemical signals govern various types of social behaviors, such as mating, aggression, and movement. For example, in slime mold amoeba, the production and emission of cyclic adenosine monophosphate (cAMP) causes amoebas to cluster when food is scarce. Perhaps more interestingly, the behavior of ants in colonies can be attributed to chemical signals.

Ant colonies have no hierarchical structure or chain of command. Contrary to animated movies on the subject, the queen ant does not give direct orders to the workers. Instead, each ant reacts to stimuli in the form of chemical scent from larvae, other ants, intruders, food, waste, and leaves behind a chemical trail that provides stimulus to other ants.

Each ant is an autonomous unit that reacts depending on its local environment and its genetically encoded rules. Despite the lack of direct organization, ants demonstrate complex behavior (for example, building complex colonies, collective foraging for food, and piling bodies of their dead) and can solve geometric problems (for example, navigating around obstacles). The local interactions between ants give rise to the observed organizational behavior of an ant colony (see Figure 2.3).

Social Systems

Humans naturally organize themselves into families, groups, communities, societies, governments, and cultures. Humans have been social animals for thousands of years, but now their social groups span the entire world, not just their local habitat.

Due to technological advances, such as the Internet and jet propulsion, we have worldwide corporations, a global economy, and the United Nations. However, at the basic level, these social groups are still made up of individuals going about their daily lives, trying to fulfill basic needs through small-scale interactions. People have

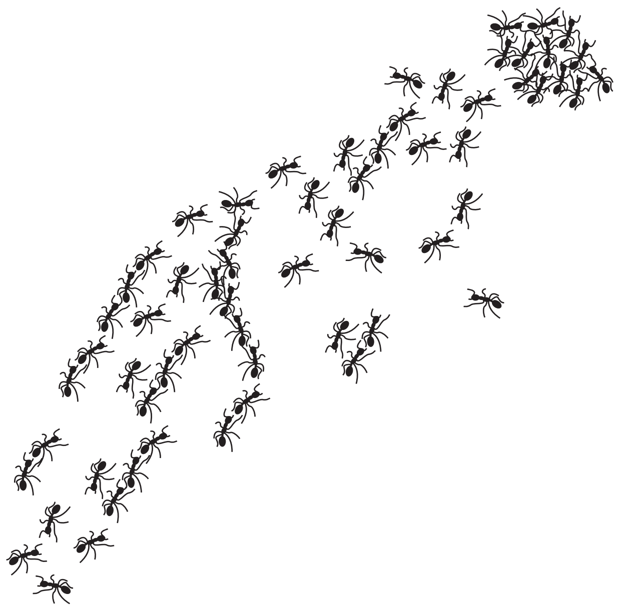


FIGURE 2.3 Ants responding to chemical signals appear organized.

simple rules, functions, actions, and interactions, even though they are, themselves, complex organisms. The complexity of an individual, however, is irrelevant at the global scale.

Economies emerge from a multitude of people trying to satisfy their material needs by individual acts of buying and selling, without any one entity controlling or planning the resulting economy. Families consist of individual members interacting with each other, to give rise to something that is more than just a group of people. Governments, though seemingly organized institutions, are themselves complex systems, made up of various branches (for example, taxation, transportation, and military) and levels trying to achieve a set of goals by performing a set of functions.

At a simpler scale, flocks of birds are groups of individuals adapting to the movement of their neighbors, which is very similar to the flow of traffic along streets or highways (see Figure 2.4). In an urban area, the flow of traffic is constrained by a set of rules (for example, legal places to drive, speed limits, intersections), but the overall dynamics are a result of individuals taking low-level actions (for example, speed up, slow down, turn right) in order to achieve their goals (for example, get to work on time).

FIGURE 2.4 Traffic flow at a busy intersection.

Key Terms

- *Complex* means complicated or intricate, hard to understand or deal with, composed of many interconnected parts, or having a complicated arrangement of parts.
- *Complex systems* are systems composed of many interdependent parts, where the whole system is more than the sum of its parts and its overall behavior is emergent.
- *Subdivision* is the removal of a section of a system to determine the effect on the whole.
- *Physical systems* are composed of atomic and sub-atomic components that obey the laws of physics.
- *Biological systems* range from ecosystems to organisms, where the elements can be as complex as animals or as simple as bacteria.
- *Social systems* are the result of interactions between individual humans or animals that create complex social networks and dynamics.

Complexity

Complexity, a concept from information and computation theory, is the amount of information that is required to describe a system. In other words, the complexity of a system depends on the level of detail that is required to describe the system.

There are many methods that are used to measure complexity in various fields, including algorithmic complexity, computational complexity, graph complexity, hierarchical complexity, Shannon's information, logical depth, simplicial complexity, and thermodynamic depth.

There are two types of complexity that are relevant to complex systems: static complexity and dynamic complexity. Static complexity relates to how an object or system is constructed, including the structure of the system and the interactions between components of the system. Static complexity is independent of the processes involved in encoding and decoding information in the system. For example, the static complexity of a network is determined by the number and type of nodes in the network. A network with few nodes (for example, a single perceptron) will have low static complexity, whereas a network with many layers of nodes will have high static complexity.

Dynamic complexity relates to the computational effort required to describe the state of a system. Static complexity affects dynamic complexity, but they are not equivalent. A system can have a simple structure and yet have complex dynamical behavior. For example, a network with a given number of nodes can have low connectivity (a few connections between nodes) or high connectivity (connections between many nodes). Each network will display different behavior and have varying levels of dynamic complexity. Each connection will necessitate additional information in describing the state of the system.

The number of bits required to specify the state of a given system depends on the number of possible states of that system. If the number of states is equal to *X*, the number of bits of information required is:

$$I = \log_2(X)$$

To specify the state of the system, the possible states must be enumerated, so that each state is uniquely identified. The number of states of the representation must equal the number of states of the system. For a string of *N* bits, there are 2^N possible states. Therefore, $X = 2^N$, and:

$$I = \log_2(2^N) = N$$

Complexity is a measure of the difficulty involved in understanding a system. For dynamic systems, the description includes how the system changes over time.

The dynamic complexity of a system is related to the dependence of the system's behavior on its components' behavior.

The more components that must be described in order to describe the whole system's behavior, the more information required and the greater the complexity of the system. The amount of information that is required to describe the behavior of a complex system is a measure of its complexity.

Key Terms

- *Complexity* is a measure of the amount of information that is required to describe a system.
- *Static complexity* relates to how an object or system is constructed, including the structure of the system and the interactions between components of the system.
- *Dynamic complexity* relates to the computational effort required to describe the state of a system.

Scientific Approaches

There are various ways of understanding and explaining complex systems. Some common scientific and philosophical approaches include reductionism, holism, and collectivism (see Figure 2.5). *Reductionism* is the idea that systems can be reduced to their basic components and reconstructed to gain an understanding of how the system works.

Holism holds that the behavior of a system cannot be understood by reducing it to its components, because the whole is more than the sum of its parts. *Collectivism* further adds that the dynamics between the parts and the whole is not static, instead there is continuous nonlinear feedback between a system and its components.

Reductionism

The reductionist approach seeks to decompose a system to its primitive components. The idea of reductionism is that the nature of complex entities can be reduced to the nature of sums of simpler or more fundamental entities. For example, it is theorized that biology can be reduced to chemistry and that chemistry can be reduced to physics. Ultimately, everything in the universe can be reduced to the laws of physics and all phenomena are constrained to the laws of physics. Reductionism is the approach favored by most physicists.

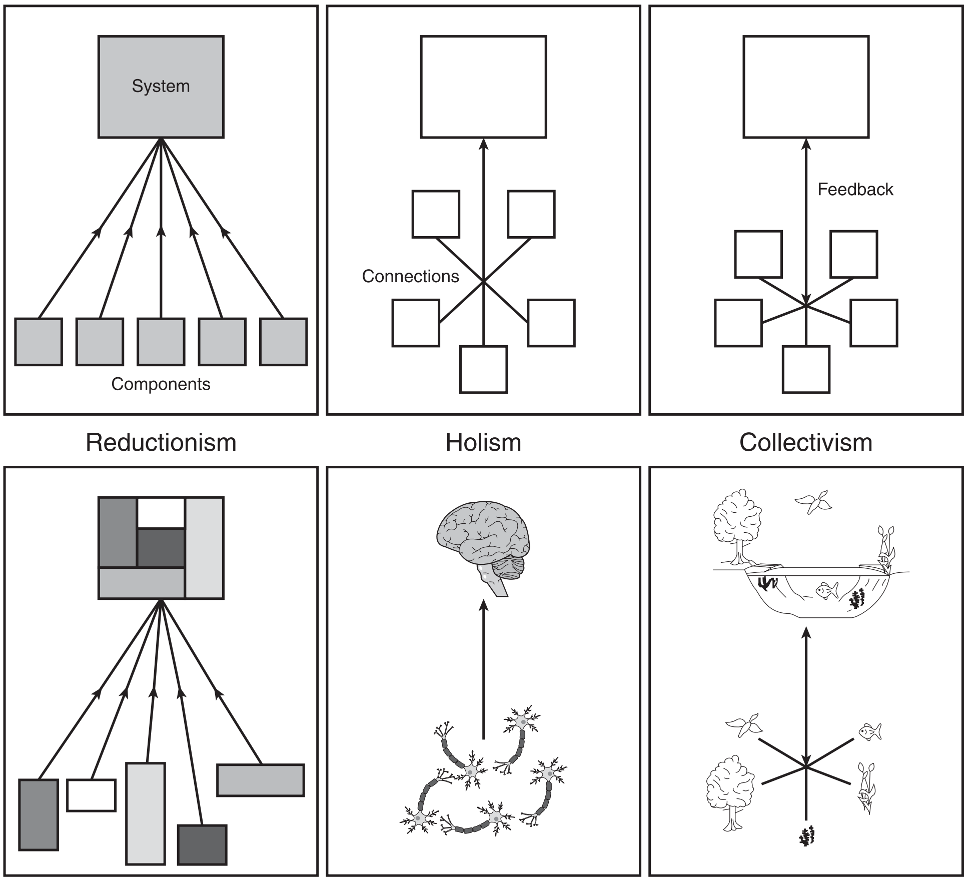


FIGURE 2.5 Reductionism, holism, and collectivism.

Analysis of a system via the reductionist approach follows the assumption that in order to understand a system, it must first be broken into its constituent parts. The understanding of the system is then gained by reconstructing the system from these parts. However, in systems where the overall behavior depends on the interaction between the components of the system, such as complex systems, the reductionist approach often misses the most crucial contributor to the system's behavior, the interconnections and interdependencies between elements. "The ability to reduce everything to simple fundamental laws does not imply the ability to start from those laws and reconstruct the universe" (Anderson, 1972).

Holism

At first glance, holism seems to characterize the nature of complex systems. Holism is the idea that the properties of a system cannot be determined or explained by the sum of its parts alone. The system as a whole must be considered to explain the behavior of the parts. It was Aristotle, in his *Metaphysics*, who first coined the phrase "the whole is more than the sum of its parts," which is now the catchphrase of complex systems. The term *holism* was first defined as "the tendency in nature to form wholes that are greater than the sum of the parts through creative evolution" (Smuts, 1926).

Holism, like complex systems, identifies that simple systems can produce unexpected, complex behavior. Holism also holds that these systems are irreducible and that it is not possible to model or approximate such a system unless the entire system, including its components and their interactions, is simulated. Furthermore, it is believed that higher-level properties of these systems can be emergent, and not predictable from the low-level components of the system. Holism is the philosophy that underpins connectionism, the foundation of neural network theory (see Chapter 5).

Collectivism

Collectivism identifies that the nonlinear feedback between the micro and macro levels of a system (the components and the whole) is essential to the dynamics of complex systems. A system is more than just a larger entity synthesized from smaller entities. In order to fully understand a complex system, the system must be viewed as a coherent whole, whose evolution is continuously refined by nonlinear feedback between its macroscopic state and it microscopic components.

For example, in an ecosystem composed of a diverse range of species, each species evolves with the other members of the ecosystem by following an evolution process that is itself a function of the evolving ecosystem. The individuals collectively define the co-evolving ecosystem and the ecosystem determines the evolutionary process. The nonlinear feedback between the individual species (microscopic level) and the global ecology (macroscopic level) that is the collective of the individual species drives the evolution of the entire system.

Another example of nonlinear feedback within a system is the ability of the human species to alter human DNA. This modified DNA then feeds back into the evolution of the species, creating a nonlinear feedback mechanism between the whole and the constituents. The co-evolutionary, self-organized, micro-macro feedback dynamics defined in collectivism are necessary to allow higher-level, emergent behavior. The nonlinear feedback between levels is what makes traditional linear analysis of complex systems difficult.

Key Terms

- *Reductionism* is the idea that complex things can be reduced to and understood from their basic components.
- *Holism* holds that the whole is more than the sum of its parts and that something cannot be understood by reducing it to its components.
- *Collectivism* asserts that the nonlinear feedback between the whole and its components make up its dynamics, not just the interactions between the components.

Additional Reading

The following references provide additional information on scientific approaches:

- Anderson, P. W. (1972) More is Different. *Science* 177 (4047), pp. 393–396.
- Smuts, J. (1926) *Holism and Evolution*. London: Macmillan.

Chaos Theory

Chaos theory describes the ability of nonlinear, dynamical systems to, under certain conditions, exhibit a phenomenon known as *chaos*. Chaos is irregular, deterministic behavior that is highly sensitive to initial conditions. In chaos theory, very simple, dynamical rules can give rise to extremely intricate behavior, such as weather, turbulence, and fractals (see Figure 2.6).

In chaos theory, small changes in local conditions can cause major, global, long-term effects. The most frequently cited example is the *butterfly effect*. The idea is that a butterfly flapping its wings in one part of the world can cause worldwide changes in weather, due to small, local changes in atmosphere. The butterfly effect illustrates the propensity of a system to be susceptible to initial conditions, an important characteristic of chaotic systems.

Chaos is often cited as an explanation for the difficulty in predicting weather. However, meteorologists do not work at a level of detail at which chaos is relevant. Their predictions involve large masses of atmosphere over short time spans, usually starting with a new set of data each day. Therefore, the flapping of a butterfly's wings and its potential long-term, global effects, have no bearing on weather predictions. Chaos theory is also at a loss to explain the structure, coherence, and self-organization of complex systems.

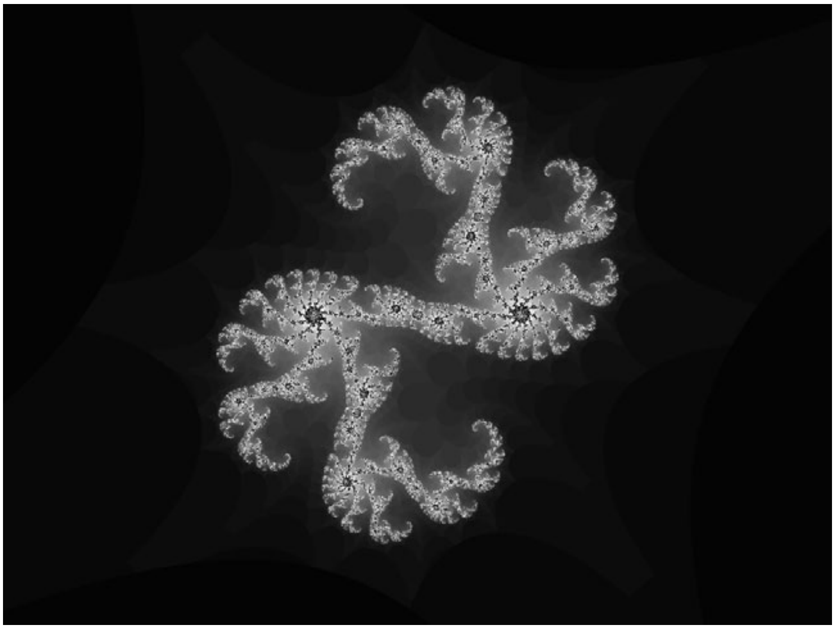


FIGURE 2.6 Fractals in chaos theory.

Edge of Chaos

More important to the study of complex systems is the concept of the *edge of chaos*. The edge of chaos is the region in complexity space toward which complex systems appear to naturally evolve. Systems that are poised at the edge of chaos are optimized to evolve, adapt, and develop emergent behaviors.

If you visualize a continuum from order to chaos, the area in which complexity is the greatest is toward the center (see Figure 2.7). In a completely ordered system, the outcome is deterministic, predictable, and a simple function of the input. An ordered system stagnates and its components behave more or less independently of one another, with no hope of evolving or adapting.

At the other end of the continuum, the behavior of the system is random and chaotic. Components of the system become too interdependent, mimicking each other's behavior or being overwhelmed by random noise.

Somewhere between the two extremes of complexity space is an optimal balance between order and chaos, where a system follows ordered rules, but where there is still a chance that something will change, and that something new will emerge. This balance is the edge of chaos, where the behavior of a system is best described as complex, neither locking into an ordered pattern, nor dissolving into chaos. At the edge

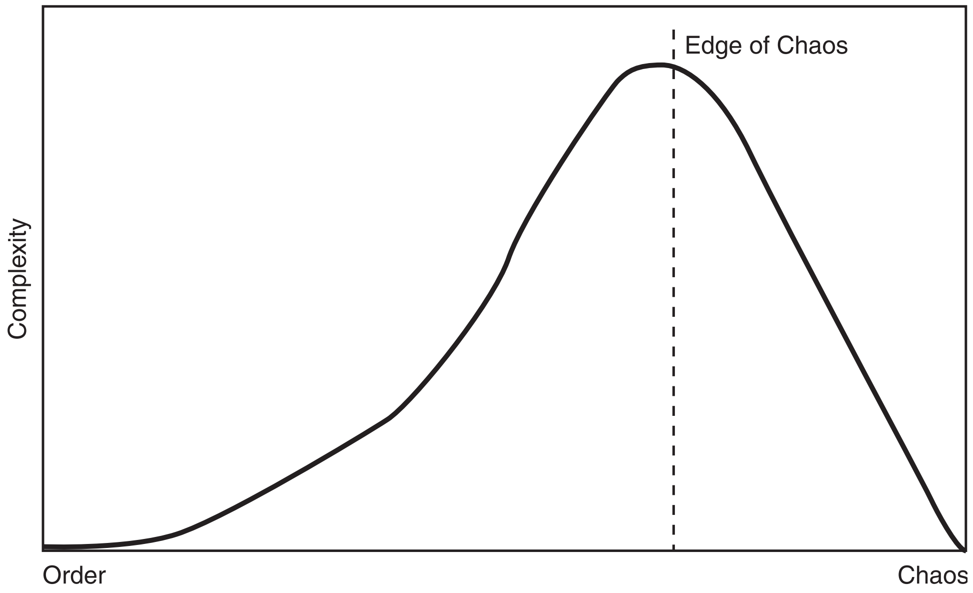


FIGURE 2.7 Complexity is at its peak between order and chaos.

of chaos, a system has enough stability to sustain itself, but enough randomness to evolve and adapt. Figure 2.8 illustrates order, edge of chaos, and chaos generated by a cellular automaton. Order is demonstrated by repeating straight lines, the edge of chaos is shown by repeating patterns with interesting variations, and chaos creates unpredictable and non-repeating random configurations.

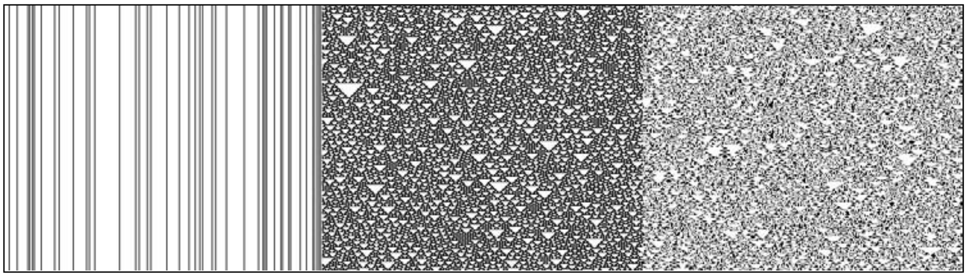


FIGURE 2.8 Order (left), edge of chaos (center), and chaos (right).

Key Terms

- *Chaos theory* describes the ability of nonlinear, dynamical systems to, under certain conditions, exhibit a phenomenon known as chaos.
- *Chaos* is irregular deterministic behavior that is highly sensitive to initial conditions.
- *Butterfly effect* is an example that illustrates the propensity of chaotic systems to be susceptible to initial conditions, in which it is proposed that a butterfly flapping its wings in one part of the world can cause worldwide changes in weather.
- *Edge of chaos* is the region in complexity space where the behavior of a system is best described as complex, neither locking into an ordered pattern, nor dissolving into chaos.

Artificial Life

Artificial life is the attempt to understand life as it is by examining life as it could be. It asserts that the way information is organized is as important to life as the physical substance that embodies the information. In artificial life, life is studied by using artificial components to capture the behavior of living systems. The premise is that if the artificial components are organized in a way that captures the organization of the living system, the artificial system will also exhibit the same higher-level behavior as the living system. Artificial life follows a bottom-up, holist or connectionist, approach. As with complex systems, the fundamental concept of artificial life is emergence.

Systems in artificial life have five general properties:

- A set of simple instructions about how individuals interact
- No master or director who directs the actions of the individuals
- Each instruction defines how individuals respond to their local environment
- No rules that direct the global behavior of the system
- Behaviors on higher levels than the individuals are emergent

Systems in artificial life include a large number of individuals, or agents, that are independently interacting with their local environment and each other. Through the multitude of simple, local interactions that occur, the collective manages to acquire properties, dynamics, and global behaviors that are not present or predictable on the scale of an individual.

The behavior that occurs often seems organized and directed, as though there were some higher power directing the movement of the individuals. The result is complex, self-organizing, and adaptive systems that carry out surprisingly complex and intricate tasks and behaviors. One of the primary modeling methods of artificial life is agent-based systems.

AGENT-BASED SYSTEMS

Agent-based systems are based on the idea that the complex, global behavior of a system is derived from the collective, simple, low-level interactions of the agents that make up the system. Agent-based models of real-world systems are used to simulate the interactions and processes that take place in those systems, in order to gain insight into the emergent structures and high-level properties of the systems. Any system whose top-level behavior is the result of the collective behavior of lower-level entities, such as biological systems, neural systems, social systems, and economic systems, can be modeled with an agent-based system.

Traditional models seek to characterize a system's top-level behavior by directly modeling the top-level variables. This top-down approach might be able to model the desired behavior, but it cannot provide an explanation for the behavior. In contrast, the approach used by agent-based systems is to determine a set of agents and rules that will result in the desired high-level behavior.

Once the desired high-level behavior has been achieved, the modeler has immediate and simultaneous access to both the top-level behavior and low-level dynamics of the system. Agent-based systems, therefore, provide an understanding of how the top-level behaviors are produced, as well as a framework for investigating how changes to the interactions or environment will propagate to the global behavior of the system.

There are four key points of agent-based systems that differentiate them from traditional top-down modeling approaches: low-level behaviors, open systems, multi-objective goals, and bounded rationality (see Table 2.3).

Autonomous Agents

Agent-based systems consist of a set of autonomous agents, each with their own set of properties and rules for behavior. Autonomous agents follow their rules in an attempt to satisfy a set of goals, which may be static or dynamic, in their changing environment.

An agent's goals can include desired local states, end goals, rewards to maximize, and internal needs that should be kept within desired bounds. Autonomous agents operate completely autonomously and do not follow instructions from other agents or by any directing program.

TABLE 2.3 Key Properties of Agent-Based Systems

Property	Description
Low-level behaviors	Low-level behaviors are modeled so that high-level properties and behaviors emerge.
Open systems	Agents are open to interact directly and freely with their environment.
Multi-objective goals	Agents must deal with many conflicting goals simultaneously.
Bounded rationality	More important for agents to adapt to their changing environments than to address a specific question within the environment (behavior-focused not knowledge-focused).

The form of the agents depends on the system that is being modeled and the agents' environment. One of the major components of an autonomous agent's environment is other agents, which means they spend a lot of time reacting and adapting to these agents. There are four behaviors that autonomous agent usually exhibit, namely *interaction* with environment, *goal-driven* behavior, *intelligence*, and *adaptation* (see Table 2.4).

TABLE 2.4 Properties of Autonomous Agents

Behavior	Description
Interaction	Senses the environment and reacts accordingly.
Goal-driven	Tries to fulfill a set of short and/or long-term goals.
Intelligence	Uses internal information processing and decision-making to select actions based on local information.
Adaptation	Anticipates future states and possibilities based on internal models.

Flocking

Flocking is the term given to the synchronous, fluid-like movement of a flock of birds. As they move, flocks give the impression of centralized control, as though some entity is directing the movement of the flock (see Figure 2.9). However, flocking is a decentralized process, with each bird in the flock moving according to its perception of its immediate environment and other nearby birds.

The smooth, coordinated, and dynamic flocking behavior emerges from the collective movements of the individual birds. Similar behavior is also exhibited by other animals that move in large groups, such as a school of fish or a swarm of bees.

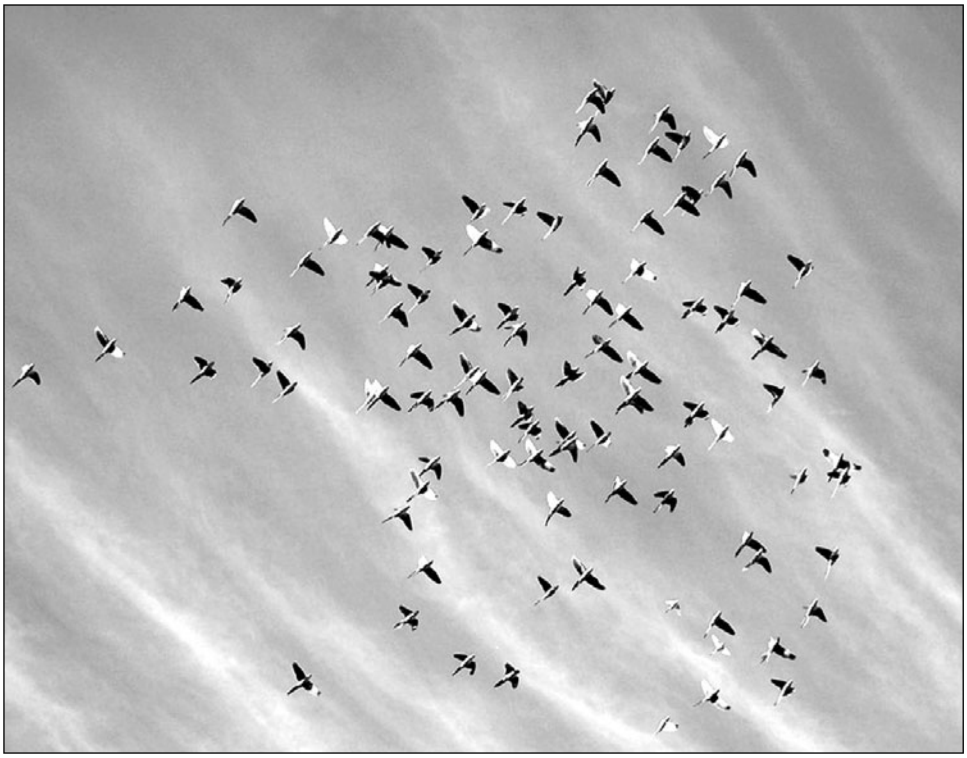


FIGURE 2.9 A flock of birds moving synchronously without global coordination.

Natural flocks consist of two, balanced, opposing behaviors, namely a desire to stay close to the flock and a desire to avoid collision within the flock. The urge to flock is the result of evolutionary pressure from several factors, which include protection from predators, improving survival of the shared gene pool, benefits of a larger search pattern when looking for food, and advantages for social and mating activities.

KEY TERMS

- *Artificial life* is the attempt to understand life as it is by examining life as it could be, by using artificial components to capture the behavior of living systems.
- *Agent-based systems* are based on the idea that the complex, global behavior of a system is derived from the collective, simple, low-level interactions of the agents that make up the system.
- *Autonomous agents* are simple entities with a set of properties that follow specific rules in order to achieve certain goals.
- *Flocking* is the term given to the synchronous, fluid-like movement of a flock of birds.

EMERGENCE

Emergence is the phenomenon that has been alluded to throughout this chapter. It is the core concept in both complex systems and artificial life. Emergence pertains to properties, behaviors, and structure that occur at higher levels of a system, which are not present or predictable at lower levels.

Both complex systems and artificial life have the common thread that there is the potential for something new to be created from simple entities interacting with their local environment. When these entities come together to form the whole, the whole is not merely a collection of these entities—it is something else entirely.

A brain is not a collection of neurons; it is a thinking machine. A human is not several connected systems; it is a sentient being. A society is not a group of co-located people; it is a powerful network capable of phenomenal things. The whole that is created from the collection is something new, with new properties, behavior, structure, and potential. This is emergence.

Emergence can occur at different levels and to varying degrees. An important distinction to make is the difference between local emergence and global emergence. Local emergence is the collective behavior that appears in small, localized parts of a system. Emergent properties of a gas, such as pressure and temperature, are examples of local emergence. They are emergent as they do not exist at the level of a single gas molecule and they are local phenomena as they are only persistent in their immediate environment. Removing a small sample of the gas does not change the pressure or temperature of the rest.

Global emergence occurs when the collective behavior of the entities relates to the system as a whole. A system must be sufficiently rich, with highly interdependent entities, for global emergent behavior to exist, such as in brains, humans, and societies. In brains, the large number of neurons with enormous interconnectivity gives rise to thought, emotions, and memory. In societies, the buying and selling habits of individuals creates global economies and marketplaces. Local emergent properties can be deduced from the lower-level entities, but global emergent properties cannot.

Systems that exhibit emergence have a common set of elements (see Table 2.5) and adhere to a common set of rules:

- Global phenomena emerge from local interactions of many simple entities
- There is no evidence of the global phenomena at the local level
- Global phenomena follow a different set of dynamics

TABLE 2.5 Common Elements of Emergent Systems

Element	Description
Entities	Low-level entities or agents that make up the system.
Rules	A set of allowable interactions for the entities.
Dynamic	Configuration of the entities changes over time.
Large state space	A large set of possible configurations.
Regularities	Persistent, recurring structures or patterns.
Self-organizing	Structure emerges from low-level interactions.

Complex systems are distinguished from systems that are merely "complicated" by the possibility of emergence. Entities in complex systems do not merely coexist, they are interconnected and interdependent. In the case of global emergence, the whole is not only more than the sum of its parts, it is something new and different.

The potential for emergence is compounded when the elements of a system have some capacity for adaptation and learning. The emergent behavior of a system can be compared to an organism's *phenotype* (or physical characteristics), whereas the individual entities are like the *genotype* (or genetic characteristics). The only way to affect or alter an organism's phenotype, or a complex system's high-level behavior, is by changing its genotype. These changes will then propagate back to the high-level, visible behavior of the system.

Key Terms

- *Emergence* pertains to properties, behaviors, and structure that occur at higher levels of a system, which are not present or predictable at lower levels.
- *Local emergence* is the collective behavior that appears in small, localized parts of a system and can be deduced from the lower-level entities.
- *Global emergence* occurs when the collective behavior of the entities relates to the system as a whole and it cannot be deduced from the lower-level entities.

Summary

In this chapter, you explored complex systems and emergence in the world at large, including examples in physics, biology, and society. You examined various scientific approaches to understanding complex systems, as well as key concepts such as chaos, order, complexity, and the edge of chaos. You also learned about artificial life and agent-based systems, including the phenomenon of *flocking*. Most importantly, you explored the meaning of emergence in complex systems, artificial life, chaos, and the world around you. With this grounding in complex systems and emergence in everyday life, you can now move on to explore the possibilities, presence, and application of emergence in games.

Additional Reading

The following papers provide a more in-depth discussion on emergence and complex systems:

- Bar-Yam, Y. (1997) *Dynamics of Complex Systems*. Reading, MA: Perseus Books.
- Holland, J. (1998) *Emergence: from Chaos to Order*. Oxford: Oxford University Press.
- Ilachinski, A. (2001) *Cellular Automata: A Discrete Universe*. Singapore: World Scientific.
- Johnson, S. (2001) *Emergence: the Connected Lives of Ants, Brains, Cities, and Software*. New York: Scribner.

Class Exercises

1. Think of three complex systems: one physical, one biological, one social.
   a. What are the elements, interactions, formation process, environment, and activities of these systems, and what makes these systems diverse?

	Physical	Biological	Social
System			
Material/Organism			
Elements			
Interactions			
Formation			
Environment			
Activities			

2. Choose one of these systems. What is the complexity of this system?
   a. How many possible states are there?
   b. How much information would it take to specify the state of the system?
3. Consider how you would explain your system using a reductionist, holist, and collectivist approach.
   a. Which provides the most accurate description?
4. Choose one of your systems and consider how you would represent it as an agent-based system.
   a. What would be your autonomous agents?
   b. What properties, goals, and rules would these agents have for interacting with their environment and other agents?
   c. What kind of local and global behaviors would emerge from these agents?
5. How well suited was an agent-based approach to modeling this system?
   a. What were the benefits and drawbacks of using these approaches in comparison to a traditional top-down approach?

3 Playing Games

In This Chapter

- Player Interaction
- The Evolution of Gameplay
- What Players Want
- Future of Gameplay

Games are a form of entertainment—developers make them for people to play and enjoy. To ignore the role of players in your games, and the importance of their feelings and contribution, is an oversight that I cannot overstate. Whatever the goal of your game—whether it is to make the best game ever, to make millions of dollars in sales, or to make the game you have always wanted to play—the players are the key component.

This chapter discusses the key elements of player interaction in games and lessons learnt from player feedback. The history of games is reviewed from a player interaction perspective (that is, how the player's role in games has changed over time). Player interaction and gameplay is then placed in the broader context of player enjoyment. Finally, you'll look to the future of game development and catch a glimpse of where emergent gameplay can take you as a developer.

Player Interaction

The primary element that separates games from other types of entertainment is *interaction*. Television and movies are watched, books are read, music is listened to, but games are played. The act of interacting with the game must provide a seamless transition into the game world. A scratch on a CD or DVD can destroy the experience of enjoying music or a movie. Missing pages or typing errors in a book can interrupt the reader's internal visualization of the story. In the same way, the interaction with a game must be flawless in order for the players to forget they are staring at a screen and moving a controller, to allow them to effortlessly step into the world of the game.

GATHERING FEEDBACK

In order to gain an understanding of players interacting in games and how a flawless transition into the game world can be achieved, it is necessary to consult the experts—the players. Acquiring player perspectives is central to enhancing the gaming experience, by understanding, and ultimately meeting, their desires and expectations.

There are several methods that can be used to gather information from players. The predominant methods used in game development and research are focus groups, playtesting, usability testing, and surveys. Each method has certain advantages and drawbacks, and each is suited to particular tasks (see Table 3.1).

TABLE 3.1 Key Player Feedback Methods

Method	Advantages	Disadvantages	Uses
Focus group	Fast, flexible	"Group think"	Idea generation
Playtesting	Test large group	Shallow feedback	General evaluation—is it fun?
Usability	In-depth, specific	Time-consuming, costly	Specific evaluation—interface
Survey	Quantitative, large group	Shallow, can be misleading	Specific or general evaluation

Focus Groups

Focus groups involve a group of players discussing their experiences in an open or structured manner. Focus groups are good for generating ideas, but are generally not suitable for evaluation due to phenomena such as "group think," which occurs when one vocal member of the group sways the opinions of the rest. Focus groups are better used for marketing or idea generation in early stages of development.

Playtesting

Playtesting is frequently carried out by game developers and involves large samples of subjects and structured questionnaires that follow a session of playing the game. Playtesting makes it easy to gain feedback from a large group of players, but the feedback is relatively shallow and general. For example, you might find out that your game is mostly fun or a bit too hard, but not specific details about where or why. Playtesting is good for identifying major bugs, issues, or trends, or for testing technical components pre-release, such as online multiplayer features.

Usability Testing

Usability testing involves the participants exploring the game and then attempting very specific tasks. It is commonly used to assess a game's interface, menus, and tutorial. Usability testing is very time-consuming (typically two or more hours per person) and usually only a handful of participants are tested. However, it provides very specific and in-depth feedback, including the opportunity to observe the players playing the game and the ability to question them on their experiences and feedback.

Surveys

Surveys allow developers and researchers to gain a large amount of quantitative data (numbers) rapidly. However, quantitative data can be misleading and open to interpretation. The usefulness of the survey also strongly depends on how well the survey and questions were designed. For example, when 60% of people rate your game as being "not fun," what should you do with this information? You know there is probably something wrong, but you don't know what or how to improve it. Surveys are a good way to follow up playtesting and usability sessions, and should be used in conjunction with qualitative methods (for example, interviews).

KEY TERMS

- *Focus groups* involve a group of players discussing their experiences in an open or structured manner.
- *Playtesting* involves large samples of subjects and structured questionnaires that follow a session of playing the game.
- *Usability testing* involves the participants exploring the game and then attempting very specific tasks, followed up by an interview and survey.
- *Surveys* are structured questionnaires that are usually administered to a large group of subjects, and typically follow playtesting and usability sessions.

Key Elements of Interaction

The wisdom gained by game developers through designing games and gathering feedback from players, along with research conducted into player interaction in game worlds, has brought to light five elements that are central to facilitating a player's transition into the game world. The five key elements of player interaction in games are *consistency*, *immersion*, *intuitiveness*, *freedom*, and *physics* (Sweetser & Johnson, 2004; see Table 3.2).

TABLE 3.2 Five Key Elements of Player Interaction in Games

Element	Description
Consistency	Relates to objects behaving in a consistent manner, enabling players to learn the rules of the game and to know when and how they can interact.
Immersion	Immersive games draw the players into the game and affect their senses and emotions through elements such as audio, graphics, and narrative.
Intuitiveness	Relates to meeting the players expectations, in terms of how they would expect to be able to interact with game objects and solve problems in the game world.
Freedom	Players want to be free to express their creativity and intentions by playing the game in the way that they want.
Physics	The physical elements of the game world, such as gravity, momentum, fire, and water, should behave in a way that the players expect.

Consistency

Consistency relates to objects behaving in a consistent manner, enabling players to learn the rules of the game, to know when they can interact with game objects, and to avoid frustration and confusion. Game worlds that behave consistently and in ways that the players can understand enable the players to become immersed. Conversely, inconsistencies in games remind the players that it is just a game, breaking their suspension of disbelief. There are two major elements of consistency:

- Visually similar objects should have similar behavior.
- Objects that have different behavior should be visually different.

It is important for objects that look the same to act the same. For example, players have reported becoming frustrated with game objects, such as glass windows or crates that break sometimes but don't break other times. Inconsistencies can make it difficult for players to learn the rules of the game, which can appear to be constantly changing. If the players learn in one instance to kick a barrel to break it, but the next time they kick a barrel it does not break, they can become confused and even frustrated with the game.

On the other end of the scale, it is important for objects that have different behaviors to look different, signaling to the players that a different kind of interaction is possible. For example, in the games *Bloodrayne* (see Figure 3.1) and *Dungeon Siege*, sections of the wall that can be destroyed are visibly different. This visual cue communicates to the players the potential for interaction. However, these visual cues should not be in the form of something unrealistic, such as a big red circle around the section of wall. Rather, it should be a subtle, realistic difference that the players can detect naturally, such as a worn-looking part of the wall that might be more fragile.

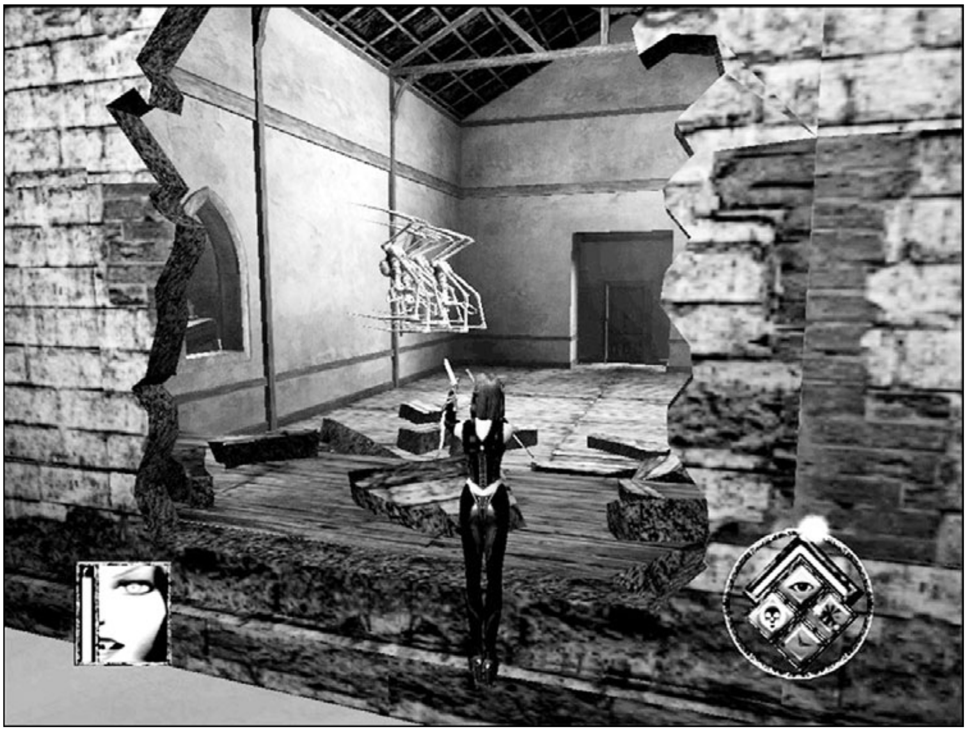


FIGURE 3.1 Sections of wall that can be destroyed in *Bloodrayne* are visibly different.
© Majesco Entertainment Company.

Game worlds that define rules and properties globally for types of objects, rather than locally for each specific object, are inherently consistent. For example, if players know that they can move objects and put objects on top of one another, they can deduce that they can stack objects to create a ladder. Games that obey a consistent set of rules for interaction allow the players to stay immersed in the game, sparing them from unpleasant surprises (Hecker, 2000).

Immersion

Immersion relates to game aspects such as audio, graphics, and narrative that draw the players into the game, enabling them to believe it is real and suspending their disbelief. Audio is very important for drawing players into a game. Effective game audio includes a powerful and moving soundtrack, as well as believable sound effects.

One way to tell whether a game is immersive is if it can cause an emotional response, such as fear or happiness. Sounds can be used to build up suspense, like in a horror movie when you know that something is creeping up on you, to the point that you're afraid and shifting in your seat. In the game *Medieval II: Total War* (see Figure 3.2), the music changes depending on the state of the game. For example, in the midst of battle, the music is fast-paced and exciting, to accelerate the players' heart rates and keep them immersed in the action of the game. The games *F.E.A.R.* and *BioShock* use sound and visual effects to create suspense, tension, and horror, which keep players on the edge of their seats throughout the game.

FIGURE 3.2 The music in *Medieval II: Total War* changes depending on the state of the game. © The Creative Assembly. Used with permission.

Research shows that sound has a greater effect on enjoyment for players who prefer first-person shooter games than for players who prefer other types of games (Sweetser & Johnson, 2004). Sound is central in first-person shooter games, because it provides immediate feedback and information to the players about what is happening around them in these information-rich environments, which include fast gameplay, different enemies, rapid movement, and numerous interactions with objects and the environment. Sound aids in setting the mood of the game and provides an additional level of immersion, by making players feel frightened or excited.

Players have reported that game graphics do not need to be spectacular, but inconsistent graphics can quickly shatter the suspension of disbelief that a game has created. As a rule, graphics should be consistent and there should be nothing that catches a player's eye as being wrong or out of place. For example, if players become stuck in a wall when adventuring in a dungeon or a monster attacks them through the wall, inconsistencies occur with the fantasy that the game has created (Hecker, 2000). Similarly, if a boom microphone appears in an emotional scene in a movie, the immersion the viewers feel—their suspension of disbelief—is instantly broken. The viewer of the movie or the player of the game is transported back to the real world, reminded and disappointed that the experience was fake.

A good introduction and a strong narrative, or storyline, are also highly important for immersion. A game's introduction gives the players the storyline and background, tells them who they are, why they're here, and what they're doing—their motivation for playing. The players then feel like they are part of the story and they want to find out more. As they play the game, more of the story is revealed, similar to reading a book, except they must complete certain tasks to be rewarded with the next installment. It has been found that narrative has a significant effect on the enjoyment of strategy game players (Sweetser & Johnson, 2004). Providing a motivating back-story and character development in strategy games can go a long way to increasing player enjoyment.

Intuitiveness

Intuitive interactions are necessary to ensure a seamless transition into the game world. Intuitiveness is about meeting players' expectations, in terms of how they would expect to be able to interact with game objects and solve problems in the game world. Player expectations are primarily built up by two types of experiences—the player's interactions with the real world and the player's previous interactions in other games. Intuitiveness of basic interactions also has a profound effect on gameplay, because players must combine basic interactions to form strategies and solve problems. Games that are more intuitive are easier to learn and provide less resistance to players becoming immersed in the game world.

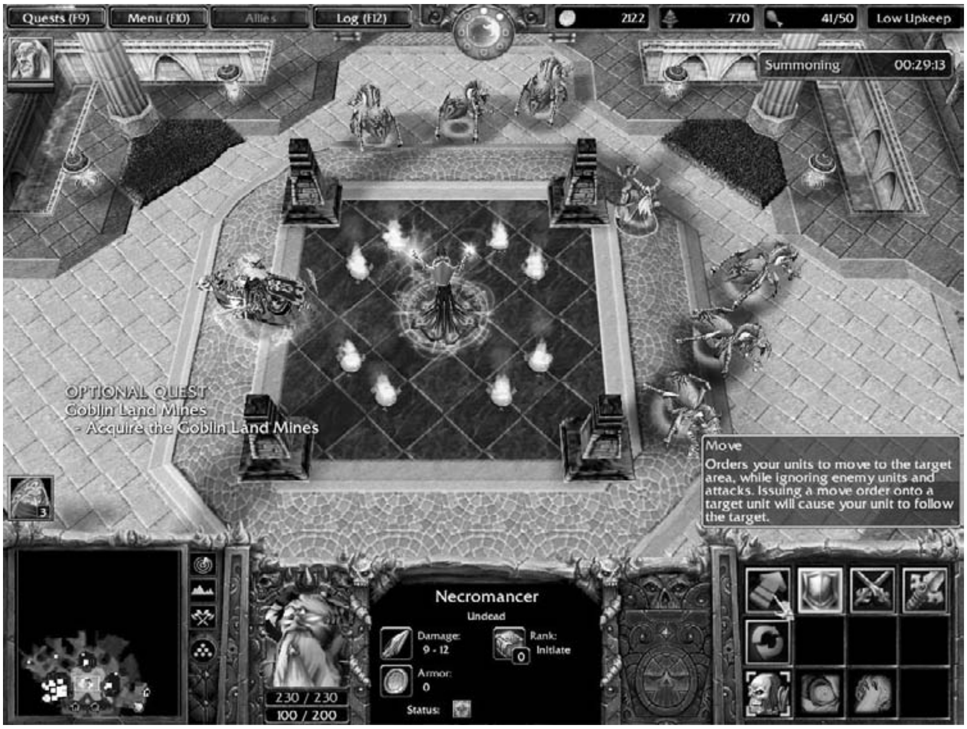
The intuitiveness of interactions in game worlds can be partly attributed to how the interactions correspond to interactions with the same objects in the real world. Game worlds are populated with objects that are visually similar to objects that people use every day, but that are functionally different. Not only can these interactions be counter-intuitive for the players, but they can often confuse and frustrate players. It is natural for players to expect that they can smash windows, stand on desks, break chairs, and use computers, because these are interactions they have learned are possible throughout their whole life. Game worlds that work in a way that reflect lifelong experiences (in the real world) are more intuitive and easier to understand for the average person, even in middle earth or galaxies far, far away.

An important benefit of making game worlds more intuitive is that they become easier to learn. Players are more likely to develop an intuitive understanding of the game elements if they are consistent with real-world elements. With the use of intuitive game elements, players are more likely to understand the elements, even when encountering them for the first time. As a result, the learning curve of the players is substantially decreased, which means that players spend less time learning and more time playing the game.

Inexperienced players only have experiences and concepts learned in real life to draw upon when trying to understand how to act in the game world. On the other hand, experienced players expect to be able to interact with game environments and objects in a particular way, learned from their prior experience playing games. It can be annoying when a game does not behave in the way that other games have trained you to expect. For this reason, it is important to adhere to industry standards and commonly used interaction mechanisms used in similar games. Trying to go against trends set by games like *Warcraft* (see Figure 3.3) and *Half-Life* can be an unnecessary, uphill struggle. If you do, you must be prepared to educate your players and be sure that your methods are an improvement on what players have learned to expect.

The intuitiveness of problem solving in games hinges heavily on the intuitiveness of basic interactions. The way a designer intends a problem to be solved might not be intuitive for the players, which can result in players resorting to trial and error. If players take an excessive amount of time to find out how to progress in a level or if they need to go to the Internet to get a walkthrough, there is an issue with the intuitiveness of the game. Therefore, it is important to conduct extensive testing to ensure that the players' expectations are met and that they will be able to solve problems and complete objectives in a reasonable time frame, rather than assuming the designer's intentions will be easily determined.

Intuitiveness has a greater effect on enjoyment for players who prefer first-person shooter games than for players who prefer other types of games (Sweetser & Johnson, 2004). Intuitive interactions with objects are important in first-person shooter games, because they require far more direct interaction with the environment (for example, direct manipulation of objects and interaction with scenery)



Move
Orders your units to move to the target area, while ignoring enemy units and attacks. Issuing a move order onto a target unit will cause your unit to follow the target.

FIGURE 3.3 The *Warcraft* series of games sets trends for real-time strategy games.
WarCraft III® images provided courtesy of Blizzard Entertainment, Inc.

than any other type of game. Also, first-person shooter games tend to follow a linear progression, so that it is often necessary to solve a problem in one section to move onto the next. Therefore, it is important that interactions in first-person shooter games are intuitive, because players will be carrying out a greater number of interactions with greater frequency and the completion of the game often depends on the success of every interaction.

Freedom

Freedom refers to the freedom that players have in expressing their creativity and intentions by playing a game in the way that they want, not the way that the designer had intended it to be played. In many games, the players are given a choice of a small number of static courses of action to take, which have been predefined by the game designers. The result is a limited set of solutions to each particular problem, which makes the game *linear* (there is only one path through the game). Consequently, the game must be played in the exact way it was specified, which is unlikely to accommodate player creativity.

Linear games force players to solve problems and perform tasks the way the designer had imagined. For example, many quests in role-playing games require the players to follow a set of very specific tasks, such as going somewhere and collecting an item. The path to complete the quest is entirely linear, with no room for freedom or expression by the players. Alternatively, what was intended can be so unintuitive to the players that they must use trial and error to work out what to do, which usually isn't fun. Players should be given the freedom to use the objects and resources they have before them to devise their own strategies and solutions to problems.

Players consider it important for there to be a variety of interactions available in each game world and that each game should have some kind of new and unique interaction (Sweetser & Johnson, 2004). Games that define global possibilities for actions the players can perform allow more open interactions in specific situations. Players have more freedom to express their creativity and gameplay can occur that was not anticipated by the designers (also known as *emergent gameplay*). Game worlds that are not full of predetermined one-to-one interactions are empowering to the players, because the gameplay becomes largely about exploring the possibility space and the game experiences become richer.

Physics

Modeling physics in games involves gravity, momentum, and the basic laws of physics behaving consistently with player expectations. The physics of game worlds can be quite puzzling to inexperienced game players. In the game world, only "explosive" barrels catch on fire, some objects are simply scenery that cannot be affected, some windows don't smash when shot, and sometimes flamethrowers work underwater. In order to be able to play computer games, it is necessary to "relearn the physics of the world like a child" (Smith, 2001).

It is important for physics to be consistent in games, to ensure the game reacts in the way that the players expect, to allow players to perform actions in an intuitive manner, and to keep them immersed in the game world. For example, if fire in the game behaves like fire in the real world, players will have an inherent understanding of how the fire works, without needing to be taught the new rules of fire within the game.

Gravity is important in games for actions such as jumping, falling, taking falling damage, trajectory when launching rockets, and so on. Modeling gravity accurately can give rise to realistic effects such as bouncing grenades around corners, falling off a platform, or rolling down a hill when shot. Momentum is also an important attribute of physics that needs to be modeled in games, especially in space simulations and first-person shooters. For example, if a player shoots an enemy or is shot by an enemy, then being pushed backward is natural. More important than modeling physics realistically, is modeling it believably and consistently. For example, games like *Unreal Tournament* and *Quake* can be in "low-gravity mode," which is not realistic but should still be consistent and believable.

The physical behavior of fire, explosions, and water in games is also important to players (Sweetser & Johnson, 2004). Flammable game objects should burn and ignite when affected by a flamethrower or incendiary grenade. When a flash grenade explodes next to a character it should adversely affect the character's sight and hearing, or when an explosion occurs, the players should be able to jump into a pool of water to be protected from damage. In the game *BioShock*, players and characters can jump into water to extinguish themselves if they are on fire. They can also use electric shocks on water to hurt enemies standing in the water.

Water is also an important substance to model consistently in games. For example, most weapons should not work under water, especially flamethrowers and fire-based weapons. Other attributes of water that are important to players are the effects of the flow and currents of the water, as well as visual effects such as ripples.

Physics in games has a greater effect on enjoyment for players who prefer first-person shooter games than for players who prefer other types of games (Sweetser & Johnson, 2004). Physics is vital in first-person shooter games, because typical behaviors include jumping, shooting, and exploding, which need to be modeled with a certain degree of realism. To play the game properly, the players need to be able to predict where their grenade will land, or what will happen if they jump off a ledge.

Key Terms

- *Consistency* relates to objects behaving in a consistent manner, enabling players to learn the rules of the game and to know when and how they can interact.
- *Immersion* relates to drawing the players into the game and affecting their senses and emotions through elements such as audio, graphics, and narrative.
- *Intuitiveness* is about meeting the players' expectations, in terms of how they would expect to be able to interact with game objects and solve problems in the game world.
- *Freedom* relates to the players' freedom to express their creativity and intentions by playing the game in the way that they want.
- *Physics* relates to the physical elements of the game world, such as gravity, momentum, fire, and water, which should behave in a way that the players expect.

Additional Reading

The following papers provide a more in-depth discussion on player interaction in games:

- Church, D. (2004) The State of Church: Doug Church on the Death of PC Gaming and the Future of Defining Gameplay. *Gamasutra*, November 23, 2004. Online at: http://www.gamasutra.com/features/20041123/hall_01.shtml.
- Smith, H. (2001) The Future of Game Design: Moving Beyond Deus Ex and Other Dated Paradigms. Online at: http://www.planetdeusex.com/witchboy/articles/thefuture.shtml.
- Sweetser, P. & Johnson, D. (2004) Player-Centred Game Environments: Assessing Playing Opinions, Experiences and Issues. Entertainment Computing—ICEC 2004: Third International Conference, *Lecture Notes in Computer Science*, 3166, pp. 321–332.

The Evolution of Gameplay

The history of gameplay shows a trend toward more interaction and player-centric gameplay. Games are becoming more realistic and immersive, coming closer to modeling the real world and the possible interactions. Players are being given ever more freedom, with game worlds becoming more intuitive, open, and emergent.

If we divide the history of gameplay into a timeline of interaction, there are four major eras worth distinguishing—interactive fiction, linear gameplay, sandbox games, and emergent gameplay. If we assign an approximate rating (low, medium, or high) to the key elements of interaction as defined in the previous section for each of these eras, there is a progression toward more interaction across the eras (see Table 3.3).

Interactive Fiction

Interactive fiction was the first step away from passive media, such as movies and books. In interactive fiction, the players are still very much the receivers of information, rather than active agents in the game world. Player interaction is in the form of limited choices between transmissions of a linear story.

TABLE 3.3 Key Elements of Interaction in Each Gameplay Era

Element	Interactive Fiction	Linear Gameplay	Sandbox Games	Emergent Gameplay
Consistency	Low	Low	High	High
Immersion	Low	Medium	Medium	High
Intuitiveness	Low	Low	High	High
Freedom	Low	Medium	High	High
Physics	Low	Medium	Medium	High

The players have no real choices, impact, or control of the game world. They simply act out a pre-scripted path, playing a slightly more active role than if they were to simply observe the story from the outside. Their role in the game becomes to "discover" the story through a limited set of actions, rather than being told the story with no participation. In terms of interactivity, interactive fiction is comparable to "choose your own adventure" books.

A popular form of interactive fiction in the 1980s was the text-based game, or text adventure. Popular text-based games include *Zork* and *The Hitchhiker's Guide to the Galaxy*. Text-based games have two methods of interaction with the players—input and output. The players input commands to change the state of the game and the game outputs a textual description of its state. Permissible input ranges from simple verb-noun pairs (such as "go west") to complex sentences that join multiple commands (such as "open door with key then go west").

Early text-based games often required a fair amount of guesswork on the players' behalf, not just in determining the right actions to take to solve problems, but also in figuring out possible valid commands to issue the game. Later text-based games were illustrated with static images, but the majority of the information was still conveyed via textual descriptions.

Interactive fiction was not just limited to text-based games. Graphic adventure games also became popular in the early 1990s. Two very popular graphic adventures were the *Monkey Island* series and the *Myst* series of games. The *Myst* series held the title of the highest selling game of all time, selling in excess of 12 million copies, before being overtaken by *The Sims*.

In *Myst*, gameplay consists of clicking on locations in the world to move there or clicking on objects to interact with them. The majority of the gameplay in *Myst* is based on solving problems, with no real enemies or conflict. Other graphic adventure games, such as the *Monkey Island* games and *Indiana Jones and the Last Crusade*, allowed the players to pick from a list of options displayed on the screen (such as Give, Pick up, Look at, Talk to, Push, and so on), removing the guesswork of previous text-based games.

The major difference between graphic adventures and text adventures is that the state of the game is conveyed to the players visually, rather than in textual descriptions. However, the limited and linear nature of the interactions remained the same. Graphic adventures were a precursor to the numerous adventure and puzzle games of the 1990s.

The "gameplay" in interactive fiction can be characterized by the discrete nature of its interactions. The players can only ever choose from a specific list of interactions in any one scene, such as typing a keyword, clicking on an object, or choosing an option from a list. Each interaction must be enumerated by the game developers and nothing outside or between these discrete actions is possible.

The players are extremely limited in what they can do at any point in time, there are no degrees of freedom, and their role in the game is simply to discover an embedded plot by finding the right set of interactions. Due to the static, specific, and linear nature of interactive fiction, along with the limited, discrete interactions available to the players, it rates *low* on each of the interaction elements (see Table 3.3).

Linear Gameplay

The next major step in gameplay involved creating persistent, continuous game worlds that players could walk around and explore. Players were given freedom to move about these worlds and could freely explore places and objects in two or three dimensions. However, despite the continuous nature of the game worlds and the resulting freedom of movement and exploration, player interactions in these worlds are still very limited. The players can only interact with each object and agent in the game world in specific, predefined ways. In order to solve each puzzle in the game, a specific, ordered set of actions must be taken. The resulting gameplay and path through the game is linear, confined, and inflexible.

The key elements of linear games are an underlying story to be discovered, puzzles to solve along the way, and a limited and predetermined set of ways to interact in the game world. The linear era of gameplay includes side-scrolling games (for example, *Sonic the Hedgehog* and *Super Mario Bros*), action-adventure games (for example, *Tomb Raider* and *The Legend of Zelda*), as well as modern role-playing (for example, *Might and Magic* series and *Wizardry* series) and first-person shooter games (for example, *Half-Life* and *Doom*). Most current games that involve some kind of storyline are essentially linear.

Action-adventure style games, such as *Bloodrayne* (see Figure 3.4), emphasize solving puzzles and navigating the environment. The players must use the moves of their characters (for example, jumping, hanging, and swimming) to solve puzzles and move between parts of the game world. Action-adventure games have a linear progression of levels, interspersed with pieces of the overarching story, and specific puzzles to be solved in each level. Puzzles include things like timing a set of jumps to reach an item on a platform and activating a set of levers in the right order to open a trapdoor. They are linear because the players must move through levels in a certain order, the storyline has a static progression, and puzzles usually have specific solutions that the players must find.

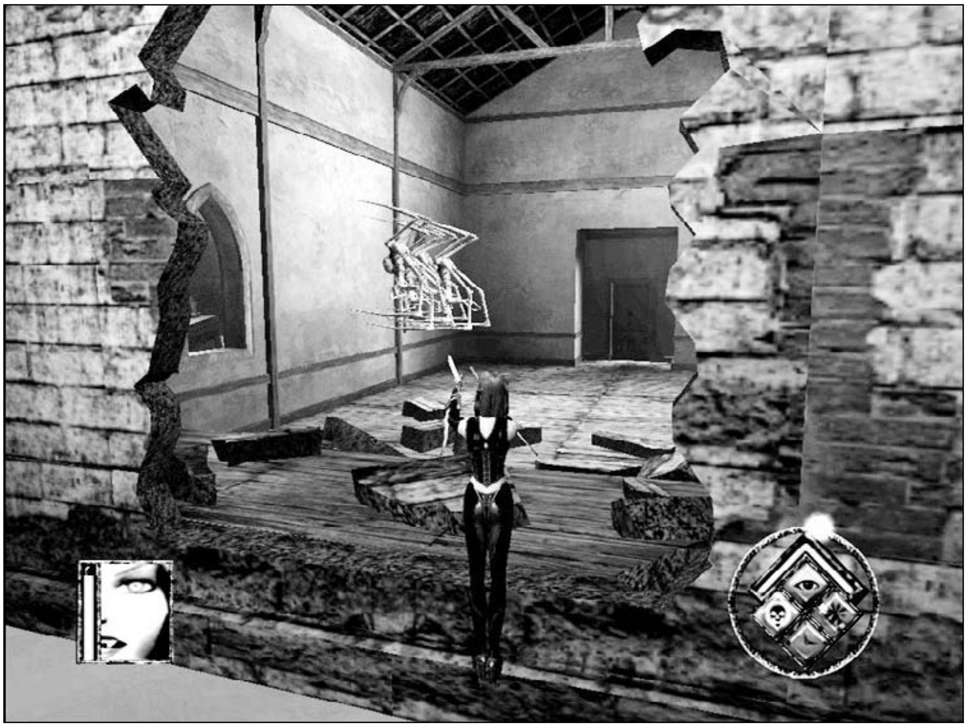


FIGURE 3.4 An action-adventure game, *Bloodrayne.* © Majesco Entertainment Company.

Another classic example of linear gameplay is in story-based first-person shooter games, such as the original *Half-Life* (see Figure 3.5). *Half-Life* has a central, linear storyline that the players must discover, by killing a series of enemies and solving various puzzles. *Half-Life* also made heavy use of scripted sequences, in which game characters would follow a set of scripted actions to challenge the players or advance the storyline. By nature, scripted sequences are linear and predefined, so they play out the exact same way every time and they do not adapt to differences in the environment. Another innovation made popular by *Half-Life* was the continuity of the game world, instead of being divided into discrete levels, which gave the players even more freedom to move about the environment. The continuous game world in *Half-Life* is similar to the hub system in the earlier first-person shooter *Hexen*.

FIGURE 3.5 A first-person shooter, *Half-Life*. © Valve Corporation. Used with permission.

As described in the key elements of interaction, first-person shooter games, role-playing games, and action-adventure games are often criticized for the lack of interactivity of many objects (scenery), inconsistencies in the game environments (for example, similar objects behave differently), and the unintuitive nature of some interactions and puzzles. Due to the specific, localized design and implementation of game objects, characters, and environments, games with linear gameplay rate *low* in consistency and intuitiveness, but allow more freedom, immersion, and have more realistic and consistent physics than their predecessors (see Table 3.3).

SANDBOX GAMES

Sandbox games are places for player experimentation. They have general goals and conditions for winning, but are open and unstructured. There is no storyline, journey of discovery, or path to be taken through the game. The players are not faced with specific challenges, conflict, or characters. Instead, they are given basic elements and a set of rules to create their own game.

Sandbox games can be classified as complex systems, and usually give rise to emergent behavior. However, whether they provide emergent gameplay is questionable, because they are more simulations than games. They are distinguished as their own category of game here due to their lack of narrative, clear goals, and challenges, and are often referred to as "electronic toys," rather than games, for these reasons. Sandbox games include city-building games (for example, *SimCity* series and *Zeus*), life-simulator games (for example, *The Sims* and *Creatures,* as shown in Figure 3.6), and other types of simulation games (for example, *Flight Simulator X* and *Trainz*).

FIGURE 3.6 Sandbox game *Creatures* is a life simulator. © Gameware Development. Used with permission.

In the *SimCity* series of games, players are given basic elements for constructing a city (for example, roads, housing, factories, schools, and so on), with the goal of creating a functional city with content inhabitants. Each type of structure has certain social, physical, and environmental effects (for example, health, education, and happiness) that influence the world and people in a given radius. Gameplay consists of placing buildings and structures in order to expand the city, attract new habitants, and improve their quality of life. The game system is complex because each structure has local effects that give rise to local and global behaviors and properties. For example, depending on the placement of buildings, the players can create different types of neighborhoods (for example, industrial areas, upper class residential areas, slums, and so on) that combine to define the structure, functionality, and effects of the entire city.

The Sims is a simulator of everyday life, and is commonly referred to as a "digital dollhouse" by its creator, Will Wright. The players use basic elements (for example, walls, doors, carpet, and so on) to construct a house, including trimmings, furnishings, and gardens. More interestingly, the players create a family of "Sims" to live in the house. Gameplay consists of instructing, or encouraging (if "free will" is enabled), the Sims to perform daily activities, such as brushing their teeth, making dinner, talking to friends, and going to work. There are bills to be paid (or the repo man comes to collect), relationships to maintain, and the needs of the Sims to meet.

Depending on the personality of the Sims, they desire varying levels of activity, social interaction, entertainment, and knowledge. If their needs aren't being met then they can become depressed (and a giant "social bunny" appears to cheer them up), overweight, fired from their jobs, or even die. There are no real objectives or conditions for winning, except to have fun and keep your Sims alive and happy. The players can have different levels of involvement in their Sims' lives, ranging from sitting back and watching them live their lives to controlling their every action.

Sandbox games have demonstrated that open gameplay and complete player freedom is possible without the confines of storytelling. Sandbox games are almost simulations, except for the somewhat loose definitions of tasks, challenges, and completion. Players create their own conflict and challenges via experimentation and extension of their sandbox, and play usually stops when they lose interest, rather than when they reach a defined goal.

In terms of the key elements of interaction, sandbox games score *high* for freedom, consistency, and intuitiveness due to their globally defined rules and object properties (see Table 3.3). However, their lack of clear goals, challenges, and narrative limits the immersion of these games. They are also usually abstract representations, which reduces the realism of their physics systems.

EMERGENT GAMEPLAY

Emergent gameplay is made possible by defining simple, global rules, behavior, and properties for game objects and their interaction in the game world and with players. Emergent gameplay occurs when interactions between objects in the game world or a player's actions result in a second order of consequence that was not planned, or perhaps even predicted, by the game developers, yet the game behaves in a rational and acceptable way.

Emergent gameplay allows the game world to be more interactive and reactive, creating a wider range of possibilities for actions, strategies, and gameplay. Local emergent gameplay occurs when a section of a game allows for new behavior that does not have knock-on effects for the rest of the game. Global emergent gameplay occurs when the simple low-level rules and properties of game objects interact to create new, high-level gameplay that alters how the game as a whole plays out.

Local emergent gameplay occurs in the games *The Sims* and *Half-Life 2* (see Figure 3.7). In *The Sims*, intelligence is embedded into objects in the environment,

FIGURE 3.7 Emergent gameplay in *Half-Life 2*. © Valve Corporation. Used with permission.

called "Smart Terrain." The objects broadcast properties to nearby agents to guide their behavior. Each agent has various motivations and needs and each object in the terrain broadcasts how it can satisfy those needs. For example, a refrigerator broadcasts that it can satisfy hunger. When the agent takes the food from the refrigerator, the food broadcasts that it needs cooking and the microwave broadcasts that it can cook food. Consequently, the agent is guided from action to action by the environment. Similarly, the game objects in *Half-Life 2* use named links, called "symbolic links" (Walker, 2004), between pieces of content to define the properties of the objects and determine how they can be affected by players and other objects.

Using this global design, the objects behave more realistically and are more interactive as they are encoded with types of behavior and rules for interacting, rather than specific interactions in specific situations. These objects afford emergent behavior and player interactions that were not necessarily foreseen by the developers.

Global emergent gameplay has been approached in games like *The Elder Scrolls IV: Oblivion* and *Vampire: The Masquerade—Bloodlines*. In *Oblivion*, there are many independent characters, organizations, and quests, each with different motivations, roles, and possible interactions with the player. The world is also expansive and filled with many enemies, animals, objects, and places players can visit. There is an extensive range of possible interactions, quests, and minor storylines, and it is unlikely that any two playing experiences will be identical. However, the game still hinges on a very linear, central storyline that the players must follow to complete the game. Although the world itself is open and varied, their choices and actions have little impact on the final outcome.

In contrast, in *Vampire*, the players' actions and decisions throughout the game impact on how their story ends. Unlike previous games, it is not merely their final dialogue choice that decides the ending, but many choices along the way. Players have a far greater sense of *agency* (the capacity to make choices and act in the world) and centrality to the game world, because their choices have consequences and they are actively changing the world and co-creating the story.

Emergent gameplay has the potential to enhance player enjoyment in terms of intuitiveness, consistency, and freedom of expression. Additionally, the physics in emergent game worlds is inherently consistent, because the laws of physics are defined globally (see Table 3.3). Emergent game systems empower the players by putting them center stage, giving them freedom to experiment, greater control, a sense of agency, and less of a feeling of uncovering a path set for them by the designers. Consequently, the game can be more satisfying and interesting for the players (see Figure 3.8).

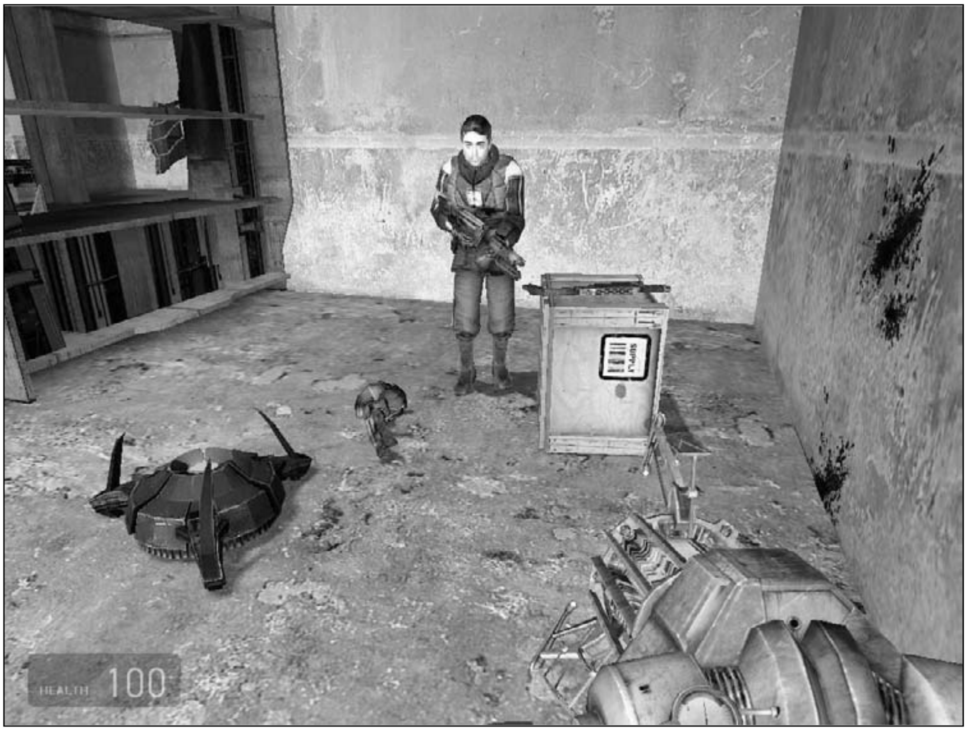


FIGURE 3.8 Game objects in *Half-Life 2* use symbolic links. © Valve Corporation. Used with permission.

Emergent games have high replayability; each time the players play the game they make different decisions, which change the game as a whole and result in different possibilities for action. The major difference between linear and emergent games is that emergent games focus on what the player wants to do, whereas linear games focus on what the designer wants the player to do (Smith, 2001).

Despite the few examples given, the emergence that has been possible in previous games has been quite limited. Games could potentially allow players to play the game in a way that was not designed or implemented by the game developer, but that works nonetheless. Emergent behavior occurs as the players use the basic elements that are provided by the game developer to create new gameplay (for example, stories or strategies). Emergence in narrative could potentially involve generating a storyline based on the interactions between the game world, characters, and objects. Emergence in gameplay could be developed to the extent of a fully emergent game world, in which there are no scripted paths, interactions, or behaviors.

Key Terms

- *Interactive fiction* requires the players to discover a prescripted story via a set of limited interactions, such as typing in key words, clicking on the interface, or choosing an option from a list.
- *Linear gameplay* involves an underlying story to be discovered, puzzles to solve along the way, and a limited and predetermined set of ways to interact in the game world.
- *Sandbox games* are almost simulations, except for the somewhat loose definitions of tasks, challenges, and completion. The players are given basic elements and a set of rules to create their own game.
- *Emergent gameplay* occurs when interactions between objects in the game world or the player's actions result in a second order of consequence that was not planned by the game developers, yet the game behaves in a rational and acceptable way.

What Players Want

Player enjoyment is the single most important goal for computer games. If players do not enjoy a game, they will not play the game. Enjoyment is a far deeper concept than "fun," which infers light and fleeting entertainment. Enjoyment pertains to a more basal pleasure that can only be achieved by being completely absorbed by something, with a sense of accomplishment and alteration at its completion.

Many different theories have been proposed to explain and analyze enjoyment in media, including disposition theory, attitude, transportation theory, cognition, and parasocial interaction. Each of these models and theories aims to analyze and understand enjoyment in terms of one specific aspect or concept. However, individually these theories are fairly narrow, and do not provide well-rounded models of enjoyment. For example, enjoyment cannot be sufficiently explained by attitude toward a particular genre (for example, science fiction), or by social context alone (for example, who an experience is shared with).

Conversely, *flow* theory is based on the premise that the elements of enjoyment are universal, providing a general model that summarizes the concepts common to all when experiencing enjoyment (for example, ability to concentrate on a task). The general, broad nature of flow theory makes it the ideal construct for a concise model of player enjoyment in games.

Enjoyment and Flow

Flow is based on Csikszentmihalyi's (1990) extensive research into what makes experiences enjoyable. Csikszentmihalyi's research consisted of long interviews, questionnaires, and other data collection over a dozen years from several thousand respondents. He began his research with people who spend large amounts of time and effort on activities that are difficult, but provide no external rewards (for example, money or status), such as composers, chess players, and rock climbers. Later studies were conducted with ordinary people with ordinary lives, asking them to describe how it felt when their lives were at their fullest and when what they did was most enjoyable. His research was conducted in many countries (for example, USA, Korea, Japan, Thailand, Australia, Europe, and on a Navajo reservation) and he found that optimal experience, or flow, is the same the world over. He also found that very different activities are described in similar ways when they are being enjoyed and that enjoyment is the same irrespective of social class, age, or gender.

Flow is an experience "so gratifying that people are willing to do it for its own sake, with little concern for what they will get out of it, even when it is difficult, or dangerous" (Csikszentmihalyi, 1990).

Flow experiences consist of eight elements:

- A task that can be completed
- The ability to concentrate on the task
- Concentration is possible because the task has clear goals
- Concentration is possible because the task provides immediate feedback
- The ability to exercise a sense of control over actions
- A deep but effortless involvement that removes awareness of worries and frustrations of everyday life
- Concern for self disappears but sense of self emerges stronger afterward
- Sense of duration of time is altered

The combination of these elements causes a sense of deep enjoyment that is so rewarding that people feel that expending a great deal of energy is worthwhile simply to be able to feel it (Csikszentmihalyi, 1990). Additionally, an important precursor to flow is a match between the person's perceived skills and the challenges associated with the task, with both being over a certain level.

Most flow experiences occur with activities that are goal-directed, bounded by rules, and that require mental energy and the appropriate skills. For example, reading is one of the most frequently enjoyed activities throughout the world (Csikszentmihalyi, 1990). Reading has a goal and requires concentration and knowledge of the rules of the written language. Reading skills begin with literacy, but also involve the ability to turn words into images, empathies with fictional characters, recognize historical and cultural contexts, anticipate plot twists, and critique and evaluate.

Throughout history, activities such as games, sports, art, and literature have been developed for the express purpose of enriching life with enjoyable experiences (Csikszentmihalyi, 1990). The key element in flow is that the activity is an end in itself—it must be intrinsically rewarding, or *autotelic*. This rings true in games because people play games (computer or other) for the experience itself; there is no external reward. Finally, every flow activity provides a sense of discovery, a creative feeling of being transported into a new reality, which is a familiar concept for game players.

GameFlow

GameFlow is a model of enjoyment in games, based on the elements of flow and research into user-experience and usability in games (Sweetser & Wyeth, 2005). *GameFlow* consists of eight elements—concentration, challenge, skills, control, clear goals, feedback, immersion, and social interaction. Each element consists of a set of criteria for achieving enjoyment in games and relate to Cziksentmilalyi's (1990) elements of flow (see Table 3.4).

TABLE 3.4 The Mapping of the Elements of GameFlow to Flow

GameFlow	Flow
The game	A task that can be completed
Concentration	Ability to concentrate on the task
Challenge, player skills	Perceived skills should match challenges and both must exceed a certain threshold
Control	Allowed to exercise a sense of control over actions
Clear goals	The task has clear goals
Feedback	The task provides immediate feedback
Immersion	Deep but effortless involvement, reduced concern for self and sense of time
Social interaction	N/A

The first element of flow, a task that can be completed, is not represented directly in the *GameFlow* elements, because it is the game itself. The remaining *GameFlow* elements are all closely interrelated and interdependent. Games must keep the player's concentration through a high work load, but the tasks must be sufficiently challenging to be enjoyable. The player must be skilled enough to undertake the challenging tasks, the tasks must have clear goals so that the player can complete the tasks, and the player must receive feedback on his or her progress toward completing the tasks. If the player is sufficiently skilled and the tasks have clear goals and feedback, then the player will feel a sense of control over the task.

The resulting feeling for the player is total immersion or absorption in the game, which causes the player to lose awareness of everyday life, lose concern for him or herself, and have an altered sense of time. The final element of player enjoyment, social interaction, does not map to the elements of flow, but is featured highly in user-experience literature on games. People play games for interaction with other people, regardless of the task, and will even play games that they do not like or if they do not like games at all.

For each element, the *GameFlow* model includes an overall goal and a set of central criteria that can be used to design and evaluate games with respect to player enjoyment (see Table 3.5).

TABLE 3.5 GameFlow Criteria for Player Enjoyment in Games

Element	Criteria
Concentration	Games should require concentration and the player should be able to concentrate on the game.
Challenge	Games should be sufficiently challenging and match the player's skill level.
Player skills	Games must support player skill development and mastery.
Control	Players should feel a sense of control over their actions in the game.
Clear goals	Games should provide the player with clear goals at appropriate times.
Feedback	Players must receive appropriate feedback at appropriate times.
Immersion	Players should experience deep but effortless involvement in the game.
Social interaction	Games should support and create opportunities for social interaction.

Concentration

For a game to be enjoyable, it needs to require concentration and the player must be able to concentrate on the game. The more concentration a task requires, in terms of attention and workload, the greater the absorption in the task. When all of a person's relevant skills are needed to cope with the challenges of a situation, that person's attention is completely absorbed by the activity and no excess energy is left over to process anything other than the activity.

CONCENTRATION

Games should require concentration and the players should be able to concentrate on the game. Here are the criteria for concentration:

- Games must provide stimuli that is worth attending to
- Games should quickly grab the players' attention and maintain their focus throughout the game
- Players shouldn't be burdened with tasks that don't feel important
- Games should have a high workload, while still being appropriate for the players' perceptual, cognitive, and memory limits
- Players should not be distracted from tasks that they want and need to concentrate on

Challenge

Challenge is consistently identified as the most important aspect of good game design. Games should be sufficiently challenging, match the player's skill level, vary the difficulty level and keep an appropriate pace. An important precursor of flow is a match between the person's perceived skills and the challenges associated with an activity, with both skills and challenges being over a certain level. If the challenges are greater than the skills, the result is anxiety and if the challenges are less than the skills, the result is apathy.

CHALLENGE

Games should be sufficiently challenging and match the player's skill level. The criteria for challenge are as follows:

Æ

- Challenges in games must match the player's skill level
- Games should provide different levels of challenge for different players
- Games should provide different levels of challenge for different players
- The level of challenge should increase as the player progresses through the game and increases his or her skill level
- Games should provide new challenges at an appropriate pace

Player Skills

For games to be enjoyable, they must support player skill development and mastery. In order for players to experience flow, their perceived skills must match the challenge provided by the game and both challenge and skills must exceed a certain threshold. Therefore, it is necessary that players develop their skills at playing a game to truly enjoy the game.

Player Skills

Games must support player skill development and mastery. The criteria for player skills are as follows:

- Players should be able to start playing the game without reading the manual
- Learning the game should not be boring; it should be part of the fun
- Games should include online help so players don't need to exit the game
- Players should be taught to play the game through tutorials or initial levels that feel like playing the game
- Games should increase players' skills at an appropriate pace as they progress through the game
- Players should be rewarded appropriately for their effort and skill development
- Game interfaces and mechanics should be easy to learn and use

Control

In order to experience flow, players must be allowed to exercise a sense of control over their actions. Players should be able to adequately translate their intentions into in-game behavior and feel in control of the actual movements of their character and the manner in which they explore their environment. Players should be able to move their characters intricately, effectively, and easily through the world and easily manipulate the world's objects, which become tools for carrying out the players' goals.

It is important that players perceive a sense of impact onto the game world and that their actions and decisions are co-creating the world they are in and the experiences they are having. Players should feel a sense of control over their character and be free to play the game and solve problems in the way that they want. In short, the players should feel like they are playing the game, not being played by it.

Control

Players should feel a sense of control over their actions in the game. The criteria for control are as follows:

- Players should feel a sense of control over their characters or units and their movements and interactions in the game world
- Players should feel a sense of control over the game interface and input devices
- Players should feel a sense of control over the game shell (starting, stopping, saving, and so on)
- Players should not be able to make errors that are detrimental to the game and should be supported in recovering from errors
- Players should feel a sense of control and impact onto the game world—their actions matter and they are shaping the game world
- Players should feel a sense of control over the actions that they take and the strategies that they use and that they are free to play the game the way that they want

Clear Goals

Games should provide the players with clear goals at appropriate times. Games inherently have an object or goal, but in order to achieve flow these goals must be clear. Games should present the players with a clear overriding goal early in the game, which is often done through an introductory cinematic that establishes the background story. The goal should be conveyed to the players in a clear and straightforward way. Also, individual levels should have several sub-goals, to help the players on the way to achieving the overall goal.

CLEAR GOALS

Games should provide the players with clear goals at appropriate times. The criteria for clear goals are as follows:

- Overriding goals should be clear and presented early
- Intermediate goals should be clear and presented at appropriate times

Feedback

Players must receive appropriate feedback at appropriate times. During flow, concentration is possible because the task provides immediate feedback. Games should use scores to tell players where they stand and players should always be able to identify their score and status in the game. In-game interfaces and sound can be used to deliver necessary status feedback. Games should also provide immediate feedback for player actions.

FEEDBACK

Players must receive appropriate feedback at appropriate times The criteria for feedback are as follows:

- Players should receive feedback on their progress to goals
- Players should receive immediate feedback on their actions
- Players should always know their status or score

Immersion

Players should experience deep but effortless involvement in a game. Immersion, engagement, and absorption are concepts that are frequently discussed and highly important in game design and research. The element of flow that describes immersion is deep but effortless involvement, which can often result in loss of concern for self, everyday life, and an altered sense of time.

Deep but effortless involvement is commonly reported by game players and people who observe them. Players become less aware of their surroundings and less self-aware than previously. Many game players report devoting entire nights or weekends to playing games without being concurrently aware of doing so or consciously deciding to do so. Enjoyable games transport the players into a level of personal involvement emotionally and viscerally, drawing the players into the game and affecting their senses through elements such as audio and narrative.

Immersion

- Players should experience deep but effortless involvement in the game
- *Criteria for Immersion*
- Players should become less aware of their surroundings
- Players should become less self-aware and less worried about everyday life or self
- Players should experience an altered sense of time
- Players should feel emotionally involved in the game
- Players should feel viscerally involved in the game

Social Interaction

Games should support and create opportunities for social interaction. Social interaction is not an element of flow and can often even interrupt immersion in games. Real people provide a link to the real world that can knock players out of their fantasy game worlds. However, it is clearly a strong element of enjoyment in games. People play games for social interaction, whether or not they like games or the game they are playing. Therefore, social interaction is not a property of the task as are the other elements of flow, but the task is a means to allow social interaction.

To support social interaction, games should create opportunities for player competition, cooperation, and connection. Game experiences should be structured to enhance player-to-player interaction and should create enjoyment of playing with others inside and outside of the game.

Social Interaction

Games should support and create opportunities for social interaction. The criteria for social interaction are as follows:

- Games should support competition and cooperation between players
- Games should support social interaction between players
- Games should support social communities inside and outside of the game

Enjoyment in Emergent Games

The *GameFlow* criteria can be divided into two categories—criteria related to game design and criteria related to gameplay. Game design relates to the creation of game elements that mostly relies on the game developer's creativity, design ability, and thorough testing. These elements are unlikely to emerge from the creation of an emergent game system. Rather, they must be specifically designed and implemented, such as the back story for a game, the goals of a level, or the game's interface.

On the other hand, emergent games can facilitate elements of player enjoyment that relate to gameplay (player interactions within the game world), such as player skills, control, and feedback. Emergent games also have the potential to enhance player enjoyment in terms of design elements that are dependent on interactions in the game world, such as concentration and challenge.

Emergent games have the potential to enhance player enjoyment by supporting the *GameFlow* elements of concentration, challenge, player skills, control, and feedback, by allowing more intuitive, consistent, and emergent interactions with the game world. Emergent games can support:

- *Concentration,* by allowing more and a greater variety of interactions with the game world, creating more interactions and effects within the game world, and simplifying content creation for game developers.
- *Challenge*, because the players have many more possibilities for strategies, the players' strategy is more likely to match their skill level as they have formulated it themselves, and the players can extend and refine their strategies as they become more skilled.
- *Player skills,* in terms of the game being easy to use and learn, because the game rules reflect real-world rules and are consistent.
- *Control,* because the players have more freedom in performing actions and interactions that they want and expect to be able to perform, the players' actions have an impact, and the players feel a sense of control over their actions and strategies and can play the way that they want.
- *Feedback,* because the game world immediately reacts to players' actions, providing implicit feedback.

Key Terms

- *Flow* is an experience so gratifying that people are willing to do it for its own sake, with little concern for what they will get out of it, even when it is difficult or dangerous.

Æ

- *GameFlow* is a model of enjoyment in games, based on the elements of flow.
- *Concentration* states that games should require concentration and the players should be able to concentrate on the game.
- *Challenge* states that games should be sufficiently challenging and match the player's skill level.
- *Player skills* states that games must support player skill development and mastery.
- *Control* states that players should feel a sense of control over their actions in the game.
- *Clear goals* states that games should provide the players with clear goals at appropriate times.
- *Feedback* states that players must receive appropriate feedback at appropriate times.
- *Immersion* states that players should experience deep but effortless involvement in the game.
- *Social interaction* states that games should support and create opportunities for social interaction.

Additional Reading

For a detailed discussion of *flow* and *GameFlow*:

- Csikszentmihalyi, M. (1990) *Flow: The Psychology of Optimal Experience.* New York, NY: Harper Perennial.
- Sweetser, P., and Wyeth, P. (2005) GameFlow: A Model for Evaluating Player Enjoyment in Games. *ACM Computers in Entertainment* 3 (3). New York, NY: ACM Press.

Future of Gameplay

In the previous section, I made the distinction between aspects of player enjoyment that are related to game design (elements crafted by game developers) and elements related to gameplay (player interactions with and in the game world). To take this distinction to the next level—game design provides the boundaries or structure of the system, whereas gameplay is the space of possibilities within those boundaries.

Games that have well-defined boundaries and structure (or game design) reduce the space of possibilities for gameplay, giving rise to more linear gameplay.

Conversely, games that have loosely defined boundaries and structure allow more freedom and possibilities for player actions, providing more open and emergent gameplay (see Figure 3.9).

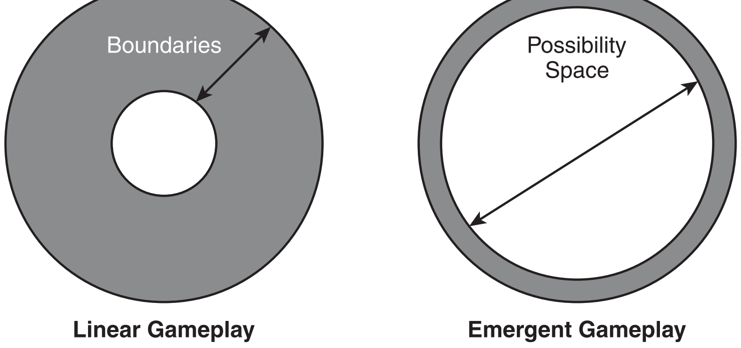


FIGURE 3.9 Increasing the boundaries and structure of a game reduces the gameplay possibility space.

The distinction between linear and emergent gameplay is not binary, and neither are the two styles mutually exclusive. Rather, there is a continuum between linear and emergent gameplay. Game systems do not need to be entirely linear or completely emergent. There are many possible levels between these extremes. The balance between emergence and linearity, or gameplay and game design, will determine the degree of the creative control the game developer possesses and the level of freedom and variation for the player.

A truly emergent system will have little to no creative control and complete freedom for the player. Conversely, linear systems provide complete creative control for the developer, but no freedom for the player. However, between these two extremes there are scripted systems with emergent elements (for example, *Half-Life 2*), which allow far more creative control for the developers than in entirely emergent systems and greater freedom and variation for the player in interacting in the game world. This middle ground provides a more balanced game in which the developer can tell a story, design challenges and tasks, and maintain a reliable and enjoyable game, whereas the player still has freedom, control, and variation.

The future of game development lies in finding the means to integrate and balance game design and gameplay and different combinations thereof to produce more enjoyable game playing experiences in a variety of game genres. There are different ways to introduce emergent gameplay into games and varying levels of freedom and emergence that can be created.

The right balance between game design and gameplay (or linearity and emergence) depends on the game that you are creating. What is right for one game will almost certainly not be right for another. Emergence definitely has a place within current games, because it can enhance player enjoyment in areas where traditional approaches have been weak (for example, giving the player more control).

The rest of this book explores different ways to incorporate emergence into games (for example, through agents, environments, narrative), investigates different levels of linearity and emergence in games and the effects on developers and players, and identifies the limitations of emergence in games. The ultimate goal is to discover how emergence can be best used in games to maximize player enjoyment.

Additional Reading

The following papers provide a more in-depth discussion of user-experience and usability in games:

- Brown, E., and Cairns, P. (2004) A Grounded Investigation of Game Immersion. *Extended Abstracts of the 2004 Conference on Human Factors in Computing Systems*. New York, NY: ACM Press, pp. 1297–1300.
- Desurvire, H., Caplan, M., and Toth, J.A. (2004) Using Heuristics to Evaluate the Playability of Games. *Extended Abstracts of the 2004 Conference on Human Factors in Computing Systems*. New York, NY: ACM Press, pp. 1509–1512.
- Federoff, M. (2002) *Heuristics and Usability Guidelines for the Creation and Evaluation of Fun in Video Games*. Unpublished thesis, Indiana University, Bloomington.
- Fullerton, T., Swain, C., and Hoffman, S. (2004) Improving Player Choices. *Gamasutra*, March 10, 2004.
- Gee, J.P. (2004) Learning by Design: Games as Learning Machines. *Gamasutra*, March 24, 2004.
- Lazzaro, N., and Keeker, K. (2004) What's My Method? A Game Show on Games. *Extended Abstracts of the 2004 Conference on Human Factors in Computing Systems*. New York, NY: ACM Press, pp. 1093–1094.
- Pagulayan, R., Keeker, K., Wixon, D., Romero, R., and Fuller, T. (2003) User-Centered Design in Games. In J.A. Jacko and A. Sears (Eds.), *The Human-Computer Interaction Handbook: Fundamentals, Evolving Techniques and Emerging Applications*. Mahwah, NJ: Lawrence Erlbaum Associates, pp. 883–905.

Summary

This chapter began your journey into emergence in games with a player-centered focus. You learned the basics of gathering feedback from players, using focus groups, playtesting, usability testing, and surveys. You also explored the key elements of player interaction in games: consistency, immersion, intuitiveness, freedom, and physics. You examined the history of games from a player interaction perspective, stepping through the eras of interactive fiction, linear gameplay, sandbox games, and emergent gameplay. You then learned about player enjoyment in games using the *GameFlow* model and the role of emergence in enhancing player enjoyment in games. You should now have an appreciation of the role of players in games, their importance and contribution, and the elements that lead to improved interaction and enjoyment. With your players in mind and their enjoyment as your focus, you will now move on to creating emergence in games.

Class Exercises

1. Think of games you have played that you would class as interactive fiction, linear, sandbox, and emergent.
   a. How does the player interact with each of these games? What constitutes the input and output?
   b. How would you rate each of these games for consistency, immersion, intuitiveness, freedom, and physics? Why?
   c. Which of these is the best and worst game that you have played? How does this correlate with your interaction ratings?

	Interactive Fiction	**Linear Gameplay**	**Sandbox Games**	**Emergent Gameplay**
Example				
Interaction				
Consistency				
Immersion				
Intuitiveness				
Freedom				
Physics				

2. Rate the importance of each of the elements of *GameFlow* on a scale of 1 to 10, for each of first-person shooter (FPS), role-playing (RPG), strategy, and sports games.
   a. How would you personally rate the importance of each element?
   b. What is your favorite type of game? How does your personal rating correspond to your rating for your favorite type of game?
   c. How are different types of games rated differently? Why?

	FPS	RPG	Strategy	Sports	You
Concentration					
Challenge					
Player Skills					
Feedback					
Clear Goals					
Immersion					
Social Interaction					

3. Consider your favorite type of game and the games of this type that you have played.
   a. How is each of the elements of *GameFlow* achieved in these games?
   b. What could be added to these games to enhance each element?
   c. Where do these games fall on the game design/gameplay continuum?
   d. What could be done to make these games more open or emergent? How would this improve the gameplay and player enjoyment?

4 Emergence in Games

In This Chapter

- Board Games
- Game Worlds
- Characters and Agents
- Emergent Narrative
- Social Emergence
- Developing for Emergence
- Emergent Games

In Chapter 3, I made the distinction between local emergence and global emergence in games. However, I'd like to take this one step further and identify three potential orders (or levels) of emergence in games. These levels can be referred to as first-order, second-order, and third-order emergence.

First-order emergence in games occurs when local interactions have knock-on or chain-reaction effects. The players' actions spread throughout the game world, affecting not only the immediate target but nearby elements of the game world as well. First-order emergence is becoming commonplace in games, especially since the advent of Valve's Source engine. Games that use property-based objects allow for a wide range of interactions and local knock-on effects.

Second-order emergence is where the players use the basic elements of the environment to form their own strategies and solve problems in new ways. Game characters might also be able to use or combine their basic actions to exhibit new behavior or strategies. These types of emergence are still local effects, because they have a limited range of effect and do not impact the game as a whole. However, they allow considerably more player freedom and creativity and change how individual parts of the game play out.

Third-order emergence pertains to the game as a whole, where the emergence occurs on a global scale. The boundaries of the game are suitably flexible to allow the players to carve new and unique paths through the game. New gameplay occurs that changes the game as a whole. The game allows for divergence in narrative, game progression, character interactions, or social systems.

Third-order emergence is the holy grail of emergence in games, but by no means the only type of value. Rather, the key is to develop emergence that will improve the player's experience of the game in some way, and never for its own sake. I whole-heartedly advocate the use of the simplest method possible to achieve the desired results. Game development is challenging enough without unnecessarily complicating matters. I do believe that, in time, games will become more realistic and emergence will eventually become the norm. However, this will be a gradual process, lead by those who have the resources and time to experiment.

This chapter samples the various ways that emergence can play a part in games. The major components discussed are game worlds, characters and agents, emergent narrative, and social emergence. Each of these sections is explored in-depth in later chapters. I also identify and discuss some of the major concerns of game developers in developing for emergence. I start with a look at board games, to see where we have come from and to gain an insight into the enormous complexity that can be achieved by systems that are simple in design.

Board Games

Although board games will not be explored in detail in this book, they are the long-standing ancestors of current computer games and much can be learned from their centuries of design, refinement, and play. Many traditional board games, such as chess, checkers, and go, contain perfect examples of emergent gameplay.

In board games, there is a finite space (that is, the board), a set of objects (that is, the pieces), and a set of simple rules that govern the gameplay. Although the structure and rules for playing these games are quite simple, the games themselves are complex. Within the confines of the game world (board, pieces, and rules), the space of possibilities is very large—the possible configurations of the board and the various ways to achieve these configurations are almost unlimited. The simple elements of these games can be combined to make complex gameplay with emergent strategies.

Chess is played on a board with eight columns and eight rows of tiles (see Figure 4.1). The game is played by two players, each with 16 pieces at the start of the game—one king, one queen, two rooks, two bishops, two knights, and eight pawns. The pieces begin the game in a prescribed arrangement and each piece has a set of allowable moves. To win the game, it is not enough to simply follow the rules, the players must anticipate their opponent, set up arrangements of pieces for future moves, and create strategies to lure, trick, and defeat their competitor.

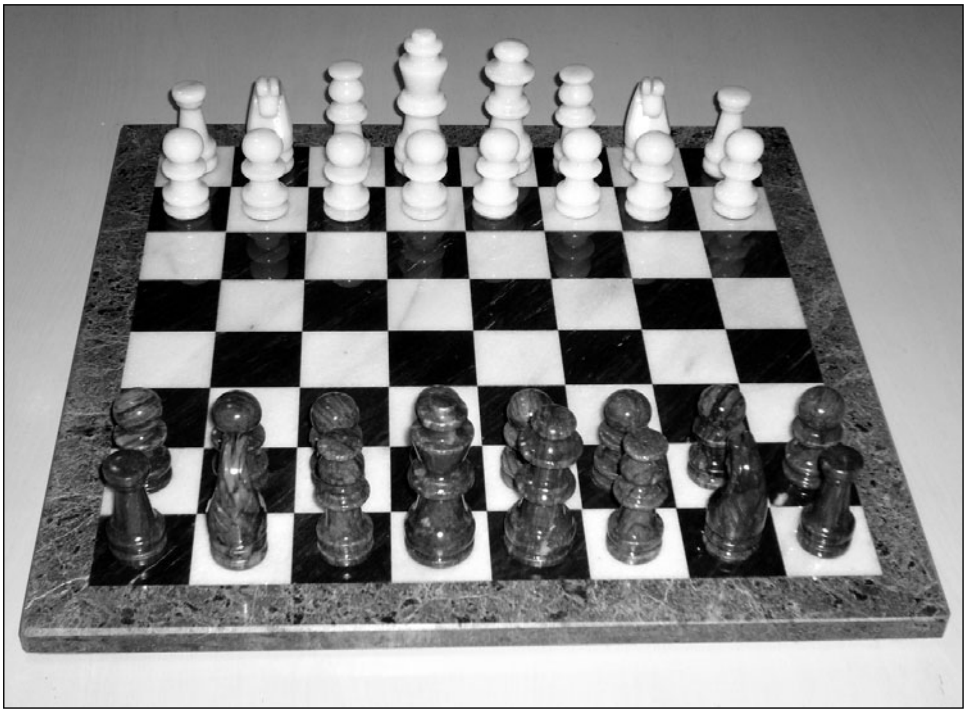


FIGURE 4.1 The game of chess.

Understanding the state of a chess game is not a simple matter of adding the values of the pieces on the board. As with all complex systems, the whole is more than the sum of its parts. The pieces on the board interact to support one another, control parts of the board, and form defensive and offensive formations. A well-structured formation of low-value pieces can easily overwhelm an opponent with higher value pieces that are poorly configured.

Chess is a game of emergence as the simple, low-level pieces and rules give rise to complex, high-level behavior and properties that are not present or predictable from the low-level components individually. Understanding the state of a chess game involves describing not only the pieces and their positions, but the emergent patterns and formations of the pieces. The patterns that are formed are not random. Rather, recognizable and recurring features exist in the high-level state of the game. For example, controlling certain formations of pawns is a commonly used pattern of play in chess. Once these formations are recognized, a player's chance of success is greatly increased.

Many common patterns of play in chess have names, such as pins, forks, skewers, undermining, overloading, and sacrificing (see Figure 4.2). Chess strategies have evolved and become more sophisticated over time, so that a chess master of the last century would be unlikely to beat a chess master of this century. Chess can even be said to have a kind of emergent narrative. In a sense, it is the classic story of good versus evil (or us versus them), with elements of battle, honor, chivalry, love, and regicide. The story of chess has been compared to that of Shakespeare's Macbeth.

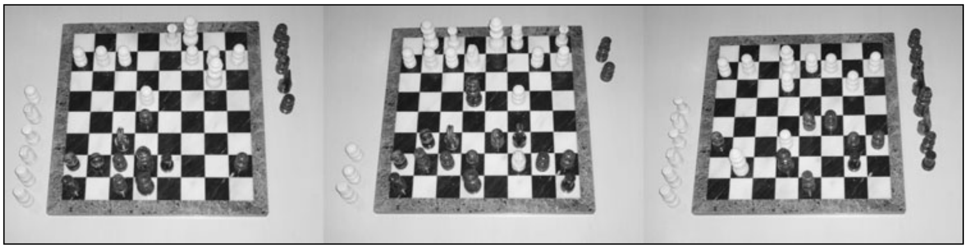


FIGURE 4.2 Patterns of play in chess—a pin (left), a fork (middle), and a skewer (right).

Similar to board games, many card games have a very simple set of elements (the cards) and a simple set of rules of play, but give rise to complex and emergent gameplay. For this reason, many board games and card games have remained popular for centuries, with endless replayability and opportunities for extending the player's repertoire of strategies. A recent popular example of a highly emergent and complex card game is *Magic: The Gathering* (see Figure 4.3).

In *Magic*, each card represents a spell with certain effects (for example, summon a creature) and rules for casting (for example, when you can play the card). The cards themselves are quite simple (for example, inflict three points of damage on a target), but when used in conjunction with other cards with varying effects, the gameplay is complex and the strategies are emergent. A large part of the game is choosing the combinations of cards that will make up your deck. Considering the effects of the cards individually makes for a poor strategy, but setting up a chain of cards that will interact, combine, and amplify each other's effects makes for very powerful gameplay.

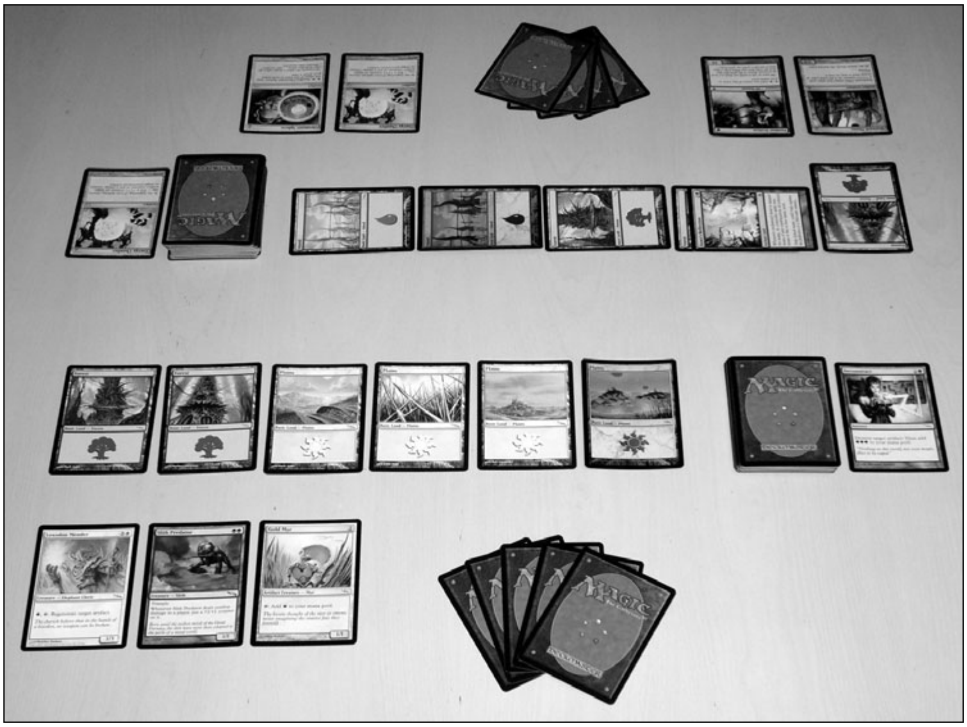


FIGURE 4.3 The card game *Magic: The Gathering.* © Wizards of the Coast, Inc. Images used with permission.

Game Worlds

Game worlds are the possibility spaces of games. The space, terrain, objects, physics, and environmental effects dictate the possibilities for actions and interactions that compose and constrain the gameplay. The elements of the game world (such as weapons, chairs, walls, and enemies) are the basic elements of gameplay, similar to the board and pieces in chess. The laws of physics and rules for interaction are the game rules, which constrain the possibility space. Within this space are the allowable actions and interactions of the player.

Interactions in the game world are the foundation of the gameplay and the types of interactions depend on the game genre. In role-playing games, interactions include talking to characters, using spells or abilities, collecting items, gaining experience, and upgrading abilities. In real-time strategy games, interactions include training units, constructing buildings, collecting resources, upgrading, attacking, and defending. In first-person shooter games, the player can run, jump, duck, hide, kick, and shoot. The gameplay is made up of how the player uses these basic interactions to solve problems, achieve goals, and advance through the game.

Creating emergent game worlds involves designing types of objects and interactions, rather than specific, localized gameplay. The properties and parameters reside at a higher level. Rather than having a specific gun able to break a specific window, there is an additional layer of abstraction that allows a gun to break anything made of glass. For example, the gun would project a bullet entity that has certain properties (for example, ballistic damage, heat, or electricity) and the glass is a stimulus-receiving entity. The system would have a set of rules about the relationship between the entities' general-case properties and when the bullet meets the glass, the game's object-property system looks up the effect of the bullet's properties on the glass entity. Therefore, the gun will work on any window (or any other stimulus-receiving object), rather than only the specified windows.

The key to creating emergent gameplay is to define a simple, general set of elements and rules that can give rise to a wide variety of interesting, challenging behaviors and interactions in varying situations. The simpler and more generalizable the rules, the easier they will be to understand (for the player and the developer), test, tune, and perfect for emergent gameplay. The simplest solution that gives the desired results is always the best. As with any emergent system, the fundamental set of rules and elements stay constant, but their situation and configuration change over time. The sensitivity of the elements to changing situations and the interaction of the elements with each other and the player are what create emergent gameplay.

Game worlds can be divided into two fundamental components—environment and objects. The environment is the space, including boundaries such as terrain, sky, and walls, as well as the physical space (for example, air in an earth-based game or water in an underwater game). The game environment in most games is inert and unresponsive to players, objects, and events. Game objects are the entities that populate the game world. There are a wide variety of objects in game worlds, which vary by game genre. Characters and agents are even types of objects, which you will read about later. Together, the environment and objects make up the game world and their properties and behavior determine the interactions that are possible and the resulting gameplay.

Environment

The environment is the central component of an emergent game system as it defines the game world and the interactions that are possible within the world. The rules that are defined for the interactions within the environment itself dictate the rules that will apply to entities that exist in the environment, such as objects and agents. Therefore, defining the rules for the behavior of the environment itself is a crucial step in developing a game world that facilitates emergent behavior.

As discussed in Chapter 3, modeling physics in games, such as gravity, momentum, and other basic laws of physics, is important to ensure realistic and consistent movement, interactions, and gameplay. Physics systems in games have become quite realistic and advanced over the last few years, which is evidenced by games such as *Half-Life 2* and *F.E.A.R.* (see Figure 4.4). However, these types of interactions are more relevant to the game objects, which you will read about in the next section, than to the environment itself.

FIGURE 4.4 Realistic physics in *F.E.A.R.* © Sierra Entertainment.

In Chapter 3, it was also noted that the physical behavior of fire, explosions, and water in games is important to players. These elements pose more of a challenge to model realistically and believably in games than the laws of mechanics. They also pertain to the environment itself, as well as the objects. Fire should burn, releasing heat and causing damage. Water should flow across surfaces, following contours, and making other substances wet. Pressure should diffuse and large pressure differences should cause explosions.

The environment in most games is inert and unresponsive to player actions. Chapter 6 discusses a framework for what I call an "active" game world, which can be used to model environmental systems, such as heat, pressure, and fluid flow in games. The Active Game World model uses simplified equations from thermodynamics, implemented with a cellular automaton. The Active Game World, based on simple interactions between cells of the environment, provides a foundation for emergent behavior to occur in game objects and agents, as well as the environment itself.

Objects

Game objects are an integral part of any game world as they compose the major source of player interactions. Objects in games are numerous and varied, including weapons (for example, guns and swords) in first-person shooter games, quest items (for example, the Holy Grail or a diary) in role-playing games, and buildings (such as barracks or factories) in strategy games. Each type of game object interacts with the game environment and the player in different ways, which gives rise to interesting possibilities for action for the player, but complicates the job of the game developer.

Some games have allowed more freedom and variation through property-based objects and rules for how the objects interact. For example, in the simulation game *The Sims*, intelligence is embedded into objects in the environment, called "Smart Terrain." The objects broadcast properties to nearby agents to guide their behavior. Similarly, the game objects in the first-person shooter game *Half-Life 2* (see Figure 4.5) use named links between pieces of content called "symbolic links" (Walker, 2004) that define the properties of the objects and determine how they can be affected by players and other objects. Using this global design, the objects behave more realistically and are more interactive as they are encoded with types of behavior and rules for interacting, rather than specific interactions in specific situations. These objects afford emergent behavior and player interactions that were not necessarily foreseen by the developers.

At the basic level, objects are the same as cells in the environment in that they both exist in the physical world and are therefore subject to the same rules of physics, such as heat transfer, fluid flow, and pressure. However, whereas all cells are uniform in structure, in that they are all sections of terrain, objects have comparatively complex physical structures. This is where the tags or labels used to create property-based game objects, such as in *The Sims* and *Half-Life 2*, come in handy.

Chapter 6 presents a framework for creating property-based game objects that can be integrated into the Active Game World model. Objects are implemented as though they are cells, using the same low-level properties based on the object's material. However, objects are also imbued with high-level properties, based on their

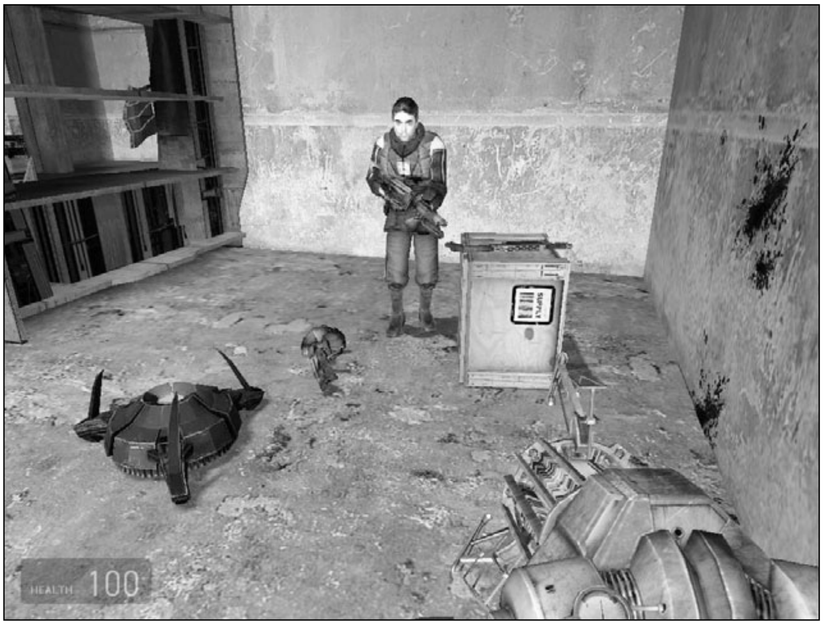


FIGURE 4.5 Objects in *Half-Life 2* use symbolic links. © Valve Corporation. Used with permission.

structure, to constrain the possible physical interactions of the objects. Objects exist within cells of the environment and can therefore be treated as additional neighboring cells for the purposes of interacting with the environment. Additionally, the high-level property tags that are attached to objects can be used to create affordances for interactions with the player and other objects. The resulting model is flexible and extensible, allowing the game world to respond consistently and realistically to a wide range of events and player actions in any situation in the game.

Key Terms

- *Game worlds* are the possibility spaces of games that define and constrain the allowable actions and interactions.
- *Environments* are the physical spaces of game worlds, including the terrain, sky, and atmosphere.
- *Objects* are the entities that populate game environment, including agents.

Characters and Agents

Characters and agents are important types of objects in game worlds; they give the game life, story, and atmosphere. Characters and agents serve many different purposes and hold many different positions in games, which contributes to making the game world rich, interesting, and complex. For example, strategy games include units (marines) that the players control and role-playing games include characters that fill a wide range of different roles in society, from kings to goblins. More than anything else in the game world, players identify with and expect lifelike behavior from game characters.

Players expect game characters and agents to behave intelligently by being cunning, flexible, unpredictable, challenging to play against, and able to adapt and vary their strategies and responses (Sweetser, Johnson, Sweetser & Wiles, 2003). However, players often find that agents in games are unintelligent and predictable. Players also believe that agents' actions and reactions in games should demonstrate an awareness of events in their immediate surroundings (Drennan, Viller & Wyeth, 2004). However, many games are proliferated with agents that do not demonstrate even a basic awareness of the situation around them. These agents often occupy the landscape as glorified pieces of scenery and behave in exactly the same way in any number of situations, ranging from rain to open gun fire. The more responsive, reactive, and dynamic the agents and characters in games, the more lifelike, believable, and challenging the game worlds will become.

Agents are a vital ingredient in creating an emergent game world. Introducing entities that have a choice of how to react to the changing environment amplifies the variation and unpredictability of a system. Reactive agents can extend emergent behavior and gameplay by adding a new level of complexity to the game world. As agents are able to choose how to react to the environment, they are able to actively change the state of the world in ways that might not have occurred without their intervention. For example, if a tank catches on fire and reacts by rolling into a group of trees, then those trees will in turn catch on fire, whereas the outcome would have been different without the active role of the tank. Also, differences between individual and types of agents, such as composition, structure, goals, personality, and so on, can add further variation and complexity. Not only can agents choose how to react to a given situation, different agents will choose to react in different ways in the same situation.

Characters and agents can be used to create emergence in games by being given an awareness of their environment and an ability to react to the changing state of the environment. The agents then become part of the living system of the game, which they sense, react to, and alter. Agents can be given the ability to respond to players and other agents, events, and conditions in their environment, as well as their own goals and motivations, by having a model of their environment and a set of rules for reacting. Characters and agents that follow simple rules for behavior, taking into account the complex environment around them, will become emergent entities in the game world.

SENSING

The agents in most games rely heavily on the prior knowledge of their designers and little on their current situation. Many agents in games, such as units in strategy games and villagers in role-playing games, do not react to the environment in any way. This behavior demonstrates a lack of situational awareness, which is an agent's dynamic mental model of its operating environment and its place in it.

Situational awareness gives an agent a sense of what is happening in its current environment, what could happen next, what options there are for action and the possible outcomes of those actions. Situational awareness is the foundation for making decisions in complex operational environments.

Giving an agent an awareness of its environment and a way to sense and model the situation is the most crucial step in creating reactive, dynamic, and emergent behavior. The more information and intelligence that can be embedded in the environment, the simpler the agents themselves can become.

The ideal framework for facilitating emergent agent behavior is to have simple agents in a complex environment. The emergence comes from the interactions between agents, between the agents and the players, and the collective interactions of the agents with the game world. In order to achieve this, the agents must be given a way to sense and model their environment. Some common approaches to sensing game environments are probing, broadcasting, and influence mapping. A framework for using each of these approaches in an emergent game system is presented in Chapter 7.

Probing

There are some games in which the agents sense and react to other agents by actively probing the environment for information. For example, the agents in *Half-Life* have sight and hearing and periodically "look at" and "listen to" the world (see Figure 4.6). Also, the game *Thief: the Dark Project* uses the same core concepts as *Half-Life*, but with a wider spectrum of states.

The agents in *Half-Life* and *Thief* periodically run through a list of rules to determine whether they sense an opponent. The agents must actively check to determine whether they can sense something at given time intervals, unlike real vision and hearing, which arrive at the senses continuously. Depending on the agents' frequency of probing the environment, it is likely that events and actions will be missed.

FIGURE 4.6 Games agents in *Half-Life* sense the environment.
© Valve Corporation. Used with permission.

Probing is quite fast and efficient if there are only a few specific things that the character is checking. However, as the number and frequency of these checks increases, the character can spend most of its time probing the environment. When the character is running a large number of checks, it is likely that most of these probes will be negative. With a lot of agents in a large environment, this can get out of hand quickly.

Broadcasting

The agents in *The Sims*, unlike *Half-Life* and *Thief*, continuously receive information from the environment. In *The Sims*, the intelligence is embedded in the objects in the environment, known as "Smart Terrain". Each agent has various motivations and needs and each object in the terrain broadcasts how it can satisfy those needs. For example, a refrigerator broadcasts that it can satisfy hunger. When the agent takes the food from the refrigerator, the food broadcasts that it needs cooking and the microwave broadcasts that it can cook food. Consequently, the agent is guided from action to action by the environment.

The behavior of the agents in *The Sims* is autonomous and emergent, based on their current needs and their environment. Whereas the agents in *Thief* and *Half-Life* use a list of rules to determine their behavior, the agents in *The Sims* use weighted sums to determine the best behavior based on the current situation.

When information is broadcast to agents, they are sent all the events that are happening in the game world and they must sort out what they need. This results in a large amount of redundancy and unused information, but the trade-off is flexibility. Agents can be set up to listen for the information they need, and discard the rest.

Influence Mapping

Influence mapping, a technique used in many strategy games, divides the game map into a grid with multiple layers of cells, each of which contains different information about the game world. For example, the layers could store data for combat strength, vulnerable assets, area visibility, body count, resources, or traversability.

The values for each cell in each layer are first calculated based on the current state of the game and then the values are propagated to nearby cells, thereby spreading the influence of each cell. This influence propagation gives a more accurate picture of the current strategic situation, because it not only shows where the units are and what they are doing, but also what they might do and the areas they potentially influence.

Each layer, or set of layers, provides information about a different aspect of the game. For example, the influence map can indicate where a player's forces are deployed, the location of the enemy, the location of the frontier, areas that are unexplored, areas where significant battles have occurred, and areas where enemies are most likely to attack in the future. When these layers are combined, they can be used to make strategic decisions about the game. For example, they can be used to make decisions about where to attack or defend, where to explore, and where to place assets for defense, resource-collection, unit-production, and research.

Influence mapping provides passive sensing of a continuous environment (as opposed to discrete entities), allows the agents' situational awareness to evolve as a function of the environment, and gives rise to reactive and emergent behavior. Unlike probing, the agent is continuously adapting its behavior to the environment (rather than probing at given time intervals) and its behavior is a function of its environment (rather than following a prescribed set of rules). The difference between influence mapping and broadcasting is that the agent is presented with a single value (calculated using the weighted sum to combine all the factors) instead of numerous messages being sent to the agent about the environment.

The sensing method that is used (probing, broadcasting, influence mapping, or other) depends on the type of agents and game world. Probing and broadcasting are frequently used in first-person shooter games and other games where individual agents react to a complex environment in real-time. Influence mapping is commonly used in games that require coordinated group behavior, such as strategy games and sports games. The computational complexity of each method in relation to the game requirements is also an important factor.

ACTING

After the agent has sensed its environment and has an understanding of its situation, it must choose an action. Even if the agent has a sophisticated world model, if it fails to act or react appropriately, it will appear lifeless and unintelligent. There are a wide range of specific actions that agents are required to take in game worlds, which vary depending on game genre. There are two major types of actions that agents are required to take—individual actions and group actions. Individual actions require the agent to behave autonomously and make decisions based on its own situation and needs. Group actions require the agent to play a role in a group of agents, which involves cooperation and coordination.

Individual

Agents that act individually are usually game characters or enemies. In first-person shooter games, a large proportion of the agents are there to fight the players. The primary actions of these agents are to run, jump, dodge, hide, and shoot enemies. In role-playing games, agents include friendly and enemy characters, as well as monsters and animals. The actions of these agents include talking, fighting, walking, and appearing to follow normal lives and routines. In strategy games, individual agents (or units) must move, attack, guard, and hold positions. Agents in sports games must move around the field or court, score goals, pass, tackle, and so on. The cars in racing games drive around the track, dodge or ram other cars, and sometimes perform stunts. The most common actions for agents in all of these types of games are movement and decision-making.

Movement

Characters spend a large amount of their time moving around, performing actions such as running away, walking around town, driving, moving to a strategic location, or charging at the players. As agents spend such a large amount of their time moving, they must do it efficiently, smoothly, and intelligently. In terms of emergent behavior, the pathfinding of individual characters is not that interesting. There are many good references on pathfinding in games, with A* being the method of choice. Deciding where to move to, on the other hand, is of more interest, especially when the agent takes into consideration the state of the environment, other agents, as well as its own goals and personality. Chapter 7 discusses how characters can use their environmental model to guide their movement.

Decision-Making

With enough information about their environment, agents can use a simple set of rules to decide how to act and react appropriately. Agent decision-making in games involves choosing an action based on goals, personality, and the current state of the game. Agents in games must decide when to run away, attack, hide, eat, talk, sleep, heal, and so on. Chapter 7 presents a simple, flexible, general-purpose framework that can be used for agent decision-making.

Group

Many games have groups of agents that must be able to interact, coordinate, and cooperate. This is particularly important in team-based games, such as strategy games and sports games. When there are two or more sides fighting or competing, the agents must cooperate in an organized way to have any chance of success. The two most important group actions in games are group movement and tactics. Emergence has a lot of potential to improve group behavior, with a focus on self-organization, rather than top-down orchestration.

Group Movement

Coordinated and fluid group movement can be achieved with a bottom-up, agent-based approach. Many movies and games have used flocking as a steering behavior for groups, schools, or herds of animals, people, and monsters. Agent-based steering behaviors are based on individual agents having a few simple rules to guide their movement in relation to their environment and other agents. These types of steering behaviors can also be extended to include goals, personality, threats, and other relevant factors. Chapter 7 discusses methods for achieving emergent group movement in games using agent-based steering behaviors.

Tactics

Tactics involve a group or team cooperating and behaving in a coordinated way in order to achieve a group goal, such as securing an area, defeating the enemy, winning a match, or making a successful play. As well as coordinated group movement, agent-based systems can be used to create emergent group tactics. If each agent considers its goals, personality, current situation, and the behavior of its team and opponents, it can make low-level decisions that will allow the high-level tactics and strategies to be emergent. Chapter 7 presents a framework for creating emergent group tactics using an agent-based approach.

Key Terms

- *Agents* are decision-making entities in games that sense and react to the game world.
- *Situational awareness* is an agent's mental model of its environment and its place in it.
- *Sensing* allows agents to monitor the changing state of their environment.
- *Probing* involves agents periodically querying the state of their environment.
- *Broadcasting* involves the environment sending game agents information about events and the state of the environment.
- *Influence mapping* provides a persistent map of the changing state of the environment, in terms of the influence of various aspects of the game world.
- *Acting* involves agents responding to changes in their environment.
- *Movement* requires agents to decide where to move to and how to get there.
- *Decision-making* involves agents choosing an action to take in their environment.
- *Group movement* requires agents to move in coordination with members of a group or team.
- *Tactics* involves agents cooperating in order to achieve a common goal.

Emergent Narrative

A game's narrative is the story that is being told, uncovered, or created as the player makes his or her way through the game. This story might take the form of a single, linear plot that is divulged to the player at selected points in time. Alternatively, it could be the deep, underlying truth of the game world that requires the player to solve puzzles and investigate the world to discover. It could also be the product of the player's interactions in the game world—the internal story that players create about their characters or challenges as they play the game. No matter the format of the narrative, it is central to enjoyment and understanding of all games, even games that do not have a story.

People in all cultures teach and learn through storytelling. From a very early age, we are told stories to not only ignite our imagination, but to teach us how to live and behave in the real world. Narrative in games frames the game in a way that players can understand and reflect upon. It is these stories that have the greatest impact on the players and that they will take with them long after they have played the game. In creating emergent narrative, the developer is tailoring the narrative to the player's experience and putting the player center stage.

NARRATIVE STRUCTURE

If you examine forms of narrative in games from the player's perspective, there are three main categories that can be identified. The first is the traditional "player as receiver" model that is drawn from other forms of storytelling, such as movies and books. In this form, the story is entirely prewritten and is simply transmitted to the player. The player receives the story and has no potential to affect the outcome or progression. A similar type of narrative is "player as discoverer," in which the story is embedded in the game world and the player must uncover the pre-existing plot. The third, and considerably different form, is "player as creator," which involves the players actively creating and affecting the story as a product of their actions and interactions. Each of these forms of narrative has been used in previous games with varying degrees of success.

Player as Receiver

Games that put the player in the role of the receiver of narrative simply deliver the plot to the player, usually in installments throughout the game. This can be done in various ways, but usually involves the player being given a piece of the story, followed by a sequence of actions or gameplay, followed by another piece of the story, and so on. In the extreme form, players are given a pre-rendered cinematic in which they watch a piece of the story unfold, followed by a discrete piece of gameplay, such as a game level. In this form, the story, at best, provides a backstory or motivation for completing the level and what the player actually does in the game has no bearing on how the story plays out. The player as receiver model is the simplest and most linear form of game narrative.

The player as receiver structure is common in first-person shooter games that use cinematics to tie together a series of game levels. For example, in the game *Painkiller* (see Figure 4.7), an introductory cinematic provides the players with a backstory that explains the main character's motivations and situation. The player then plays through a series of discrete game levels, which are interspersed with cinematics that extend the story and deepen the plot. The cinematics are all pre-scripted and pre-rendered, so they play out the same way no matter how many times the game is played. There are no alternate endings, branches, or player choices. The same format is also used in many action games, adventure games, and other level-based games. Although very simple and entirely linear, this model is used to great effect in *Painkiller* and many games like it, which is why it is so prevalent in current games.

FIGURE 4.7 A pre-rendered cinematic in *Painkiller.* © DreamCatcher Interactive Inc.

The player as receiver model is also used in many role-playing games, especially for the central storyline. Despite subplots and side-quests that might be happening at the same time, most role-playing games have a central, linear storyline. This central story is usually tied to a particular series of quests. Once the conditions for advancing the main story have been met, a cinematic or a scripted in-game dialogue sequence will play out to give the players the next installment of the story. For example, in *Diablo II*, the player is given a pre-rendered cinematic at the completion of each chapter of the game (see Figure 4.8).

A similar approach is used in many real-time strategy games, such as *Warcraft III*. In general, games that use the player as receiver structure of narrative require the players to complete a level, chapter, mission, or quest to be rewarded with the next piece of the story and advancement to the next level of the game. The story and gameplay are often not tightly intertwined, with the cinematics acting more as a reward or motivation for the action than a critical part of the gameplay.

Player as Discoverer

Games in which the player is the discoverer of the narrative are still usually very linear and scripted in nature. The pieces of the story might not occur in the same order each time, but the overarching story is linear and the outcome is predetermined. The narrative in player as discoverer games is usually more interactive than

FIGURE 4.8 *Diablo II* plays a pre-rendered cinematic at the completion of each chapter.
Diablo II® images provided courtesy of Blizzard Entertainment, Inc.

in player as receiver games. The player cannot simply wait to be told the story; they must actively try to uncover the plot. This is usually accomplished by talking to game characters, exploring, completing quests, and interacting with the game world. For this reason, the gameplay and narrative is usually more intertwined and interdependent than in player as receiver games.

Due to the interactive nature of the narrative, player as discoverer games often have branching storylines or multiple endings. The players don't just observe a piece of the story playing out; they play a role in it. This might be in the form of choosing dialogue options during a cinematic or having interactive conversations with game characters.

The player as discoverer model is used in interactive fiction (as described in Chapter 3), as well as many role-playing games. In interactive fiction, the player interacts with the game world and characters to sequentially move through the world and story. Until the player discovers what to do next, the rest of the game and story is inaccessible.

In role-playing games, such as the *Might and Magic* series, the player often needs to find the key characters to acquire information from, in order to advance the story. Depending on the player's interactions with these characters or the player's choices, the plot might branch off into different directions.

An example of a branching storyline is in the game *Wing Commander IV*. Depending on the player's choices in the cutscenes (game movies), the story plays out differently and the game has different outcomes. The player as discoverer model has not been as successful or as prevalent as the simple and effective player as receiver model.

FIGURE 4.9 Strategy game *Medieval II: Total War* does not have a defined storyline.
© The Creative Assembly. Used with permission.

Player as Creator

In the player as creator model of narrative, the players are creators or co-creators of the game's story. The story is a function of the player's actions and interactions in the game world. Narrative in player as creator games can be generated by the interactions between characters in the game world, the player's interactions in the game world, as well as any knock-on effects of these interactions. In player as creator narrative, the final destination is not important; it is the journey that counts. Players have a definite sense of agency and impact onto the game world; they are actively creating and changing the world and its story.

Games that are generally not considered to have a defined storyline fall into the category of player as creator narrative. This includes simulation games, such as *The Sims* and *SimCity*, strategy games, such as *Total War* and *Civilization*, and other open, sandbox games. In these games, the players use the basic elements of the game, such as buildings, people, and armies, to create their own stories. In

The Sims, players create a life story for their characters and in *Total War* (see Figure 4.10), players forge the history of a nation. The players are not discovering or receiving an existing plot; they are creating a new one through the act of playing the game.

FIGURE 4.10 An epic, historical battle in *Medieval II: Total War.* © The Creative Assembly. Used with permission.

Some games that do have well-defined, linear storylines can also have elements of player as creator narrative. Games that have large, open game worlds with lots of possibilities for action, such as some role-playing games, can allow players to create their own subplots. For example, in *The Elder Scrolls IV: Oblivion*, there are many optional quests that can be gained from characters throughout the world. These quests are not connected to the main storyline, do not have to be completed in a specific order, and are entirely optional. Additionally, players can go adventuring into caves or ruins and fight monsters whenever they feel like it. Players can completely ignore the main storyline, although they won't be able to complete the game by doing so. Players have a fair amount of freedom in creating a unique path for their characters through the game. Although the central storyline is linear and they will reach the same final destination, the players have a significant ability to co-create their journey through the game.

Player as creator narrative is emergent. Some, or all, of the story is a product of the player's interactions in the game world, interactions between objects or characters in the game world, and knock-on effects. The narrative is not predetermined and scripted; it emerges from interactions between entities in the game world. Emergent narrative does not need to be as complicated or as chaotic as it sounds. Chapter 8 explains a few simple ways to achieve emergent narrative in games, using the narrative elements of storyline and conversation, described in the next section.

Narrative Elements

There are two key elements that can be used to create narrative in games—storyline and conversation. Narrative is formed by telling stories about events, people, and places. Players' actions in a game can form a kind of internal narrative, but it is not until the retelling that it becomes a story.

The storyline is the overarching plot, as well as subplots, that play out in the game. As discussed in the previous section, the storyline can be received, discovered, or created by the player. The storyline is often presented in installments, such as pre-rendered or in-game cutscenes, throughout the game. These installments can recap what has happened in the previous section, reveal more depth to thicken the plot, or foreshadow what is yet to come.

Conversations are a more informal, continuous form of narrative. Players can engage in conversations with various characters throughout the game, or observe conversations between other characters, to gain small pieces of information about events, people, and places in the game. By allowing emergence in storylines and conversations in games, the developer can create emergent narrative.

Storyline

If the player's actions in the game world and interactions with objects and characters are the low-level elements of the game world, the storyline is the high-level behavior. If the story is received (as opposed to discovered or created) by the players, their actions are irrelevant to the overarching storyline, because it is imposed over their actions. There is no connection between the low-level interactions in the game world and the high-level storyline.

For a story that is discovered, the player's interactions are forced to fit the mould of the high-level behavior. This can be considered more of a top-down approach, where the interactions are determined by the high-level design, or storyline. Interactions that are incorrect or not part of the scripted path have no consequence.

In stories that are created, the low-level actions of the player, game world, objects, and characters interact to form the overarching storyline. This is where emergence can occur. The difficulty lies in designing a story system that not only enables emergence, but that makes for compelling, believable, and coherent narrative.

There are several components of storylines in games that can be used to create a compelling narrative. These components are backstory, storytelling, story creation, and post-game narrative. A backstory presents events that occurred prior to the start of the game and can be used to establish setting, character, and motivation. Storytelling is used throughout a game to impart further information about the plot or game world to the player, usually via cutscenes. Story creation is the more interactive form of storytelling in which the player performs certain actions, such as completing missions or quests, to create subplots or advance the overall plot. Finally, post-game narrative is storytelling that occurs after the game is completed, which can be used to create a story out of the player's journey through the game. Each of these components can be used to create narrative in an emergent game. Chapter 8 provides a framework for developing an emergent storyline using these components.

Conversation

Conversation is a common form of creating narrative in not only games, but other forms of storytelling, such as books and movies. The format of conversations in games is diverse, with varying levels of freedom and interactivity. Conversations can be entirely scripted, including the player's part, and simply play out for the player to observe, such as in a pre-rendered cutscene. Alternatively, characters often have conversation trees, where the player can choose a response from a limited set of options when talking to a game character. This response can be in the form of a scripted sentence, a keyword, or an emotion.

Rather than engaging in a continuous conversation, players might simply have a list of conversation topics that they can choose to ask the character about. This method often seems more like accessing a help system than having a conversation, because you are merely requesting information on a given topic. Conversation is the staple of storytelling in most role-playing games, but is also used to a lesser extent in first-person shooter and strategy games.

The character's awareness, reactions, and involvement in the conversation are also key parts of the construction of the narrative. To properly engage in a conversation, the character must have an awareness of the state of the game world, an attitude toward the player, a memory of previous interactions, their own motivations and goals, and appropriate reactions to the player's conversation choices.

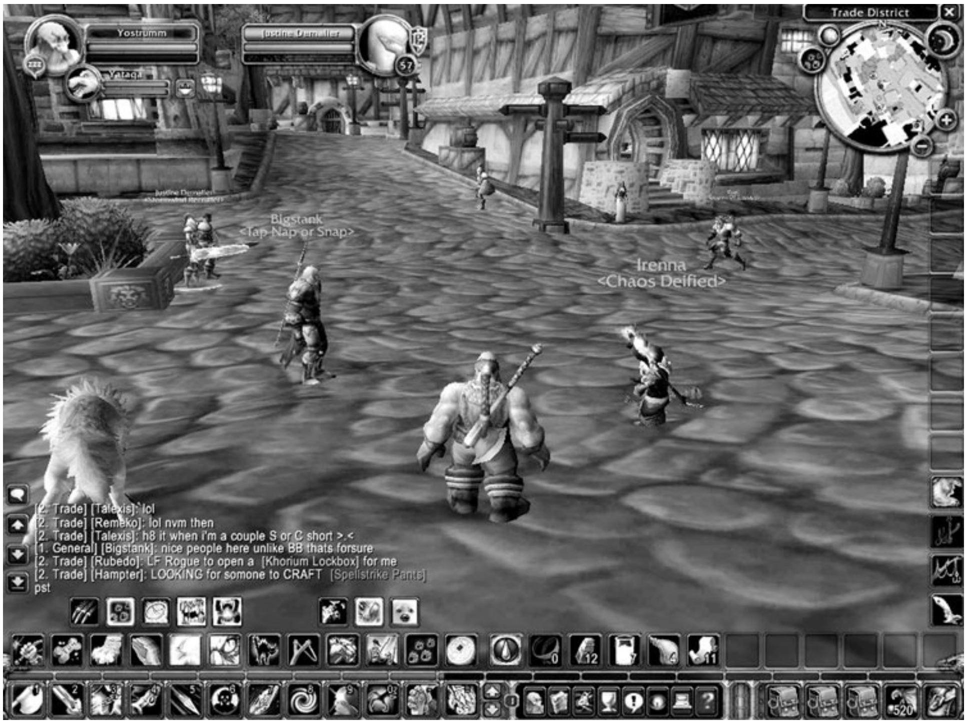
Chapter 8 describes a conversation system for enabling emergent narrative through conversation. The system involves the use of a core set of variables that affect a character's conversation and a set of rules for how the conversation is affected. The rules and variables of the system are simple, but allow emergent conversations in the context of a complex game world. A simple conversation system that is sensitive to the state of the game world, characters, and player can create emergent conversations between the player and characters, or between game characters.

Key Terms

- *Narrative* is the story that is being told, uncovered, or created as the players make their way through the game.
- *Player as receiver* narrative is delivered to the players in installments as they play the game, usually in the form of cutscenes.
- *Player as discoverer* narrative involves the players actively trying to uncover the plot, by talking to game characters, exploring, completing quests, and interacting with the game world.
- *Player as creator* narrative allows the players to create or co-create the story, which is a function of the players' actions and interactions in the game world.
- *Storyline* is the overarching plot, as well as any subplots, that play out in the game.
- *Conversations* are verbal or textual exchanges that players engage in with game characters, or observe taking place between characters.

Social Emergence

Of all the forms of emergence in games, social emergence is by far the most complex and unpredictable in current games. When millions of people come together to play popular massively multiplayer online role-playing games, such as *World of Warcraft* (see Figure 4.11) or *Lineage*, the result is comparable to the divergence and complexity of a large city. Rather than trying to find ways of creating emergent social systems in games, it is more a matter of trying to understand, model, constrain, and support the complexities that arise naturally from human interactions. In order to do this, game developers must draw on psychology, sociology, economics, and even law.



```
[2. Trade] [Talexis]: lol
[2. Trade] [Remeko]: lol nvm then
[2. Trade] [Talexis]: h8 it when i'm a couple S or C short >.<
[1. General] [Bigstank]: nice people here unlike BB thats forsure
[2. Trade] [Rubedo]: LF Rogue to open a  [Khorium Lockbox] for me
[2. Trade] [Hampter]: LOOKING for somone to CRAFT  [Spellstrike Pants]
pst
```

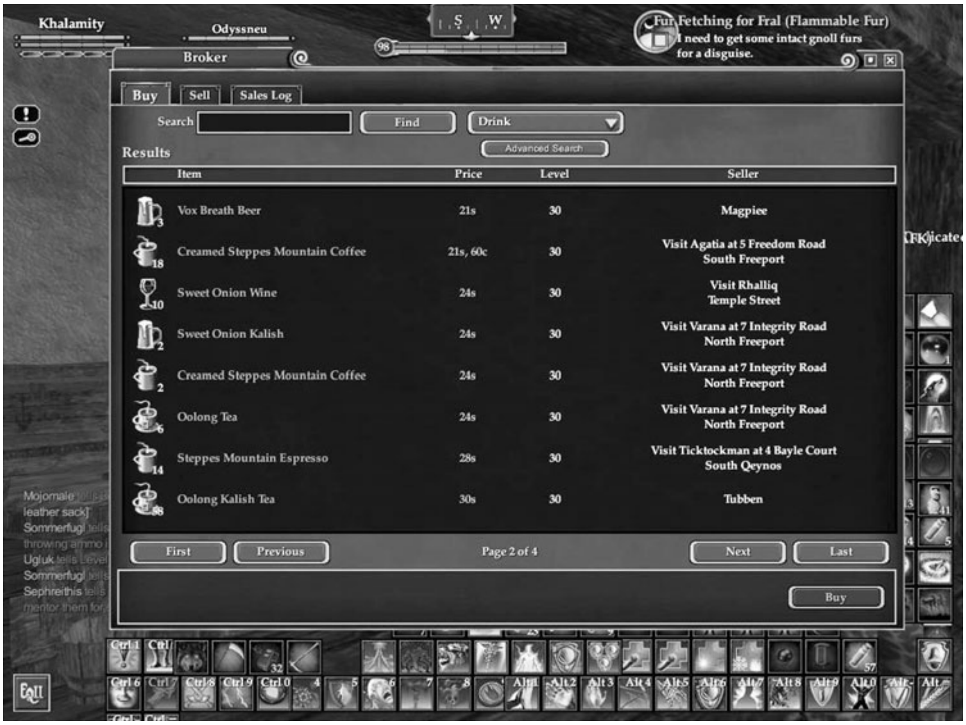
FIGURE 4.11 Millions of people come together to play *World of Warcraft.* World of WarCraft® images provided courtesy of Blizzard Entertainment, Inc.

The most prevalent forms of social emergence in games include emergent economies, social structures, and communities. Chapter 9 draws on the lessons learned from developers of major online games, as well as research into the complexities of these worlds from the perspectives of psychology, sociology, economics, and law. The result is an exploration of the major highlights and issues, as well as guidelines and suggestions for supporting and harnessing these forms of social emergence in games. Finally, you look at how artificial social networks can be created in single-player games and the emergence of artificial social communities.

Economies

The virtual economies in many massively multiplayer online games have become as complex, intricate, and difficult to manage as real-world economies. The largest virtual economies exist in popular massively multiplayer online role-playing games, such as *EverQuest* (see Figure 4.12), *Ultima Online*, *Dark Age of Camelot*, *World of*

Warcraft, *Lineage*, and *EVE Online*. Virtual economies are emergent systems that change dynamically with supply and demand, based on the trading patterns of the world's inhabitants. They share many characteristics with real-world economies, such as trading and banking, but are also subject to many of the same problems, such as inflation and gambling.



Item	Price	Level	Seller
Vox Breath Beer	21s	30	Magpiee
Creamed Steppes Mountain Coffee	21s, 60c	30	Visit Agatia at 5 Freedom Road South Freeport
Sweet Onion Wine	24s	30	Visit Rhalliq Temple Street
Sweet Onion Kalish	24s	30	Visit Varana at 7 Integrity Road North Freeport
Creamed Steppes Mountain Coffee	24s	30	Visit Varana at 7 Integrity Road North Freeport
Oolong Tea	24s	30	Visit Varana at 7 Integrity Road North Freeport
Steppes Mountain Espresso	28s	30	Visit Ticktockman at 4 Bayle Court South Qeynos
Oolong Kalish Tea	30s	30	Tubben

FIGURE 4.12 *EverQuest 2* has a large and complex virtual economy. © Sony Online Entertainment. Used with permission.

Inflation occurs in online games as the average holding per character increases over time. As a result, things continuously cost more, which causes problems for game balance. One way developers get around this problem is to create money sinks, which are things that players want to buy that don't affect gameplay. Other solutions include taxation and limiting the amount of stuff that characters can store. As games are continuously generating new currency, there must be effective sinks to maintain a stable economy. Game economies can also be destabilized by destructive actions by players, such as gold farming and hacking, which can give rise to hyperinflation.

Online role-playing games, such as *EVE Online* and *Dark Age of Camelot*, also play host to gambling and lotteries. The gambling takes the form of lotteries, betting, card games, and other real-world gambling mechanisms, using the in-game currency, but usually occurs on external Web sites. Bets and winnings are paid to the player's game accounts.

Financial institutions have begun to appear in online games that do not have game mechanics governing financial law. *EVE Online* (see Figure 4.13) was the host to the first bank in an online game world, created by the Interstellar Starbase Syndicate.

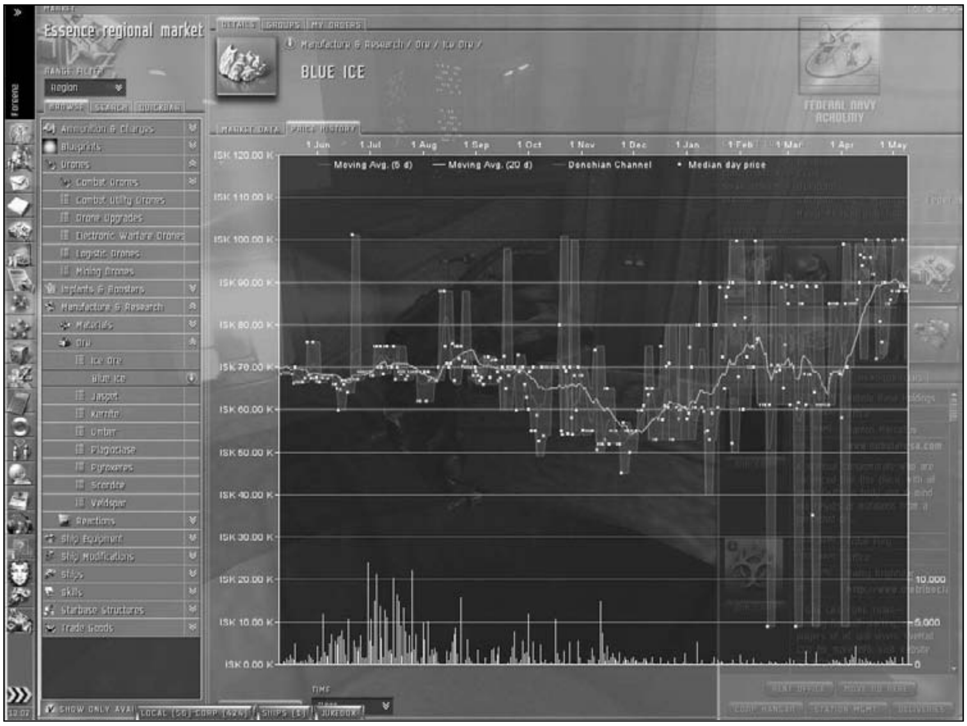
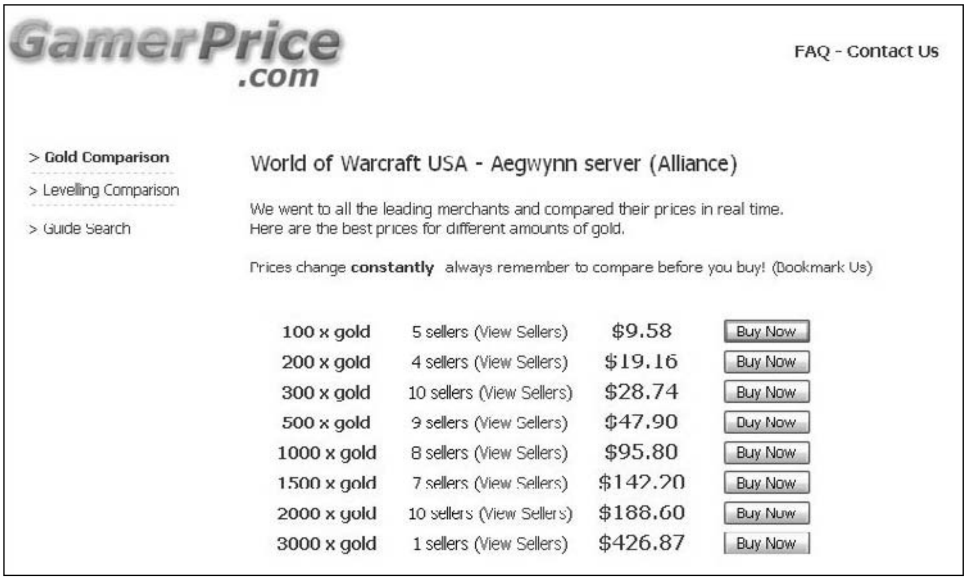


FIGURE 4.13 *EVE Online* was the first online game to host a bank. © CCP.

One of the most interesting and controversial forms of social emergence in online games is the crossover between virtual and real economies. Some players work to acquire in-game assets (such as characters and items) that they can sell for real-world money on auction or currency exchange Web sites. The crossover also works in the other direction, with players offering real-world services (such as Web site design) for in-game currency. This behavior also feeds back into the game system, because the economy and gameplay are influenced by events external to the game, which alter player motivations (to make real-world money, rather than to progress in the game) and the virtual economy of the game.

The demand to trade in-game money and the surge in popularity of online games, such as *World of Warcraft*, have given rise to a large number of external online marketplaces, auction sites, and currency converters (see Figure 4.14). The proliferation of these secondary markets has further given rise to tools for marketplace comparisons (www.gamerprice.com), which is a second-order emergent effect external to the game. It has even been suggested that secondary market sales (totaling approximately $1–3 Billion USD in 2006) may overtake subscription sales in the near future. Chapter 9 examines the emergence of economies and trading in massively multiplayer games, as well as the considerations and potential for developers.



World of Warcraft USA - Aegwynn server (Alliance)

We went to all the leading merchants and compared their prices in real time.
Here are the best prices for different amounts of gold.

Prices change **constantly** always remember to compare before you buy! (Bookmark Us)

100 x gold	5 sellers (View Sellers)	$9.58	Buy Now
200 x gold	4 sellers (View Sellers)	$19.16	Buy Now
300 x gold	10 sellers (View Sellers)	$28.74	Buy Now
500 x gold	9 sellers (View Sellers)	$47.90	Buy Now
1000 x gold	8 sellers (View Sellers)	$95.80	Buy Now
1500 x gold	7 sellers (View Sellers)	$142.20	Buy Now
2000 x gold	10 sellers (View Sellers)	$188.60	Buy Now
3000 x gold	1 sellers (View Sellers)	$426.87	Buy Now

FIGURE 4.14 *World of Warcraft* has many external marketplaces and marketplace comparison Web sites.

Social Structures

Players in massively multiplayer role-playing games often have full, developed characters with identities, skills, specializations, duties, friends, and property. These characters forge relationships, build houses, have jobs, and fulfill a role in the virtual society. Some players have characters dedicated to farming, mining, crafting and selling objects, hunting, and many other specialized activities. The individual interactions, motivations, and behavior of these players give rise to complex social structures that share many common elements with real societies.

Emergent social structures in online games include governments and political parties that form around common beliefs, desires, or goals. Virtual governments even institute laws and punish law-breakers. Virtual crimes are becoming more common-place and varied, with thefts, assaults, prostitution, and bullying. Even online mafia has emerged in some games, where powerful players threaten new players into giving them protection money, as well as carrying out organized crime. Chapter 9 discusses the emergence of social structures in games and the considerations for game developers.

Communities

Players invest large amounts of their time into games and form strong social bonds with other players over time. Communities of like-minded players come together inside and outside of game worlds, forming guilds, forums, competition ladders, and mod communities. Strong social themes of competition and cooperation are common in these communities. Status is important, and hierarchies and leaders emerge over time. Supporting the development and continuance of these communities is important for developers in sustaining interest in their games. Social bonds between players can keep them playing the game far longer than almost any gameplay mechanism.

Game communities can range from interested players coming together to discuss the game to strong social networks devoted to playing or extending the game. Guilds are common in many online games, with competition between guilds to recruit new members. The cooperation of players within a guild can allow difficult quests to be completed and better items to be acquired.

Many games have some form of online multiplayer, which allows players to compete against each other in skirmishes or tournaments, or cooperate to defeat other teams of players. Games that support non-developers in extending or modifying the game often have mod communities devoted to developing their own versions of the game for hobby or for profit in some cases. Chapter 9 discusses the various forms of game communities and what developers can do to encourage and support these communities.

Artificial Social Networks

The emergence of social structures and communities in massively multiplayer games provides insight into the power of social dynamics in games, as well as inspiration for translating these dynamics into artificial social networks for single-player games. Giving game characters the ability to form social relationships with the player, as well as other game characters, has the potential to add more life and interaction to game worlds.

Artificial social networks can be formed in games by attributing characters with status, social connections, memory of other characters, and attitude toward characters and the players. Using social networks to determine the flow of information in game worlds and character behavior in social situations can allow for emergent social interactions and character behavior. Chapter 9 presents a model for social networks in games.

Key Terms

- *Social emergence* occurs when many players interact to form complex social structures and communities.
- *Economies* are emergent in massively multiplayer games, giving rise to inflation, trading, money exchange, and banks.
- *Social structures* emerge in massively multiplayer games as the result of individual players contributing to complex societies that spawn social ladders, governments, politics, and crime.
- *Communities* form in and around games, composed of like-minded individuals that unite around common interests or goals.
- *Artificial social networks* are networks of linked non-player characters that can imitate natural social networks that emerge in games.

Developing for Emergence

Some of the important issues to consider for game developers, especially when creating new technology, are as follows:

- *Creative Control*—Level of creative control for game developers
- *Design, Implementation, and Testing*—Effort in designing, implementing, and testing the game

- *Modification and Extension*—Effort in modification and extension
- *Uncertainty and Quality Assurance*—Issues for uncertainty and quality assurance
- *Feedback and Direction*—Ease of giving feedback and direction to players

Each of these issues is described in this section and discussed with respect to developing for emergence in games. These issues will be revisited throughout the book, in relation to specific aspects of emergence in games.

Creative Control

The use of emergent systems in games can result in a loss of creative control for game developers. Using an emergent system involves defining types of interactions and behaviors, which makes it is more difficult to set up specific narrative and sequences. Consequently, controlling the flow of the game and telling a specific story is not as straightforward in an emergent system.

Games where developers manually plan and set up specific situations, interactions, and events, allow complete creative control over the game. The designers are empowered to create a specific narrative flow for the game, by defining the order and nature of the player's actions and encounters in the game. They decide what will happen and when. However, emergent systems allow a more approximate control, in that the game developer guides the player, providing boundaries for gameplay rather than dictating specifically what will happen. In an emergent system, the developer can set goals, but cannot specify how the player will get there.

Design, Implementation, and Testing

Emergent systems involve substantial initial effort in planning the rules and properties that will govern the behavior of the system. It can be difficult to decide how certain behavior should be modeled and what rules and properties best capture the behavior. Emergent systems also require substantial testing and tuning to get the rules to generate behavior that is desirable or acceptable. Getting even a simple emergent system to behave in a desired way can be difficult and involves significant tuning. With a full-scale game world, a large amount of development time would need to be spent on testing and tuning the system, to ensure it behaves reasonably, plays as intended, and to minimize undesirable exploits.

Specific game systems can also require considerable effort in planning, as well as implementing and testing. At the extreme, they involve every game element to be set up manually. Specific interactions need to be planned by the game designers and the possible courses of action that the players can take need to be manually setup. Scripted games require a great deal of time and effort by the designers, as well as vigilant manual effort to ensure consistency in the game world.

In an emergent system, the game problems can be determined and the player can find their own solution. However, in a specific system, the problem and solution must both be set and the player must find the developer's preset solution. Development of emergent systems can be more efficient as programmers can build tools that allow designers to drop objects into levels, with the properties and behavior of the object already defined. Designers can also create new objects and attribute properties to the objects using the tools.

The emergent approach definitely has benefits as games grow in size, making it impossible to predict, plan, and code everything. Games are now very large, in terms of the size of the worlds and the amount of content, and they will continue to grow. Scripting everything is already infeasible and some game developers have found that the initial outlay of effort to get an emergent game world working is a superior solution to creating the entire world manually (such as *Half-Life 2*). But so far, these games have been limited to game companies with extensive resources, experience, and time, and only the game objects have been emergent.

MODIFICATION AND EXTENSION

The minimal effort required for modification and extension is one of the major benefits of an emergent system. Once an emergent system is built successfully, the design scales well (it increases in size easily, maintaining robustness and manageability) and is easily extended. Making changes to the system (fixing bugs) has the potential to be more efficient as changes can be made to global rules and object types, rather than each particular instance of an object that needs to be changed.

Specific game systems scale poorly and do not lend themselves to extensibility. The properties and parameters of objects in specific systems can be different for each instance. Fixing bugs in the system requires each instance of a game element to be visited and reconfigured manually. Also, objects must have explicit relationships with other game elements for interactions to occur. Any changes that need to be made to the system require revision of any aspect of the game that is affected by the change.

Ease of modification and extension is an important factor in game development as it allows developers to easily add more content, create expansion packs for their games (currently a big source of revenue for games), quickly release patches to fix bugs in the games, and allow players to make modifications and create additional content. Once the initial work is done in creating and tuning an emergent system, setting up new scenarios should be a straightforward process that involves dropping in types of objects, agents, terrain, setting any desired events, and letting the system run. Conversely, modifying and extending a specific system can be difficult, time-consuming, and potentially impossible.

Uncertainty and Quality Assurance

Emergent systems introduce uncertainty, which means that the game can behave in ways that the developers had not anticipated. Although this uncertainty can give rise to desirable, emergent gameplay, it can also be undesirable if the system allows behavior that is detrimental to the game. The larger an emergent system (more entities, rules, and so on), the more complex it will become and the more variations in behavior that will be exhibited. Extensive testing is required to ensure that the game does not allow detrimental behavior. However, the emergent events can be too numerous or subtle for the development team to predict or detect during testing.

Uncertainty is an important property of emergent systems; it gives rise to new and unexpected behavior. It is also perceived as the main drawback of emergence by game developers. When human players are introduced into an emergent system, they have the ability to use the system in ways it was not meant to be used, change things in the game that the developer had not expected, and play the game in ways that could not be foreseen. Furthermore, human players seem to have a perverse drive to intentionally push the game to its limits, exploit its weaknesses, and to make it break. Consequently, game developers' fears of using emergent systems are justified, in that if a game has loopholes, exceptions, or problems then the players will not only find them, but will actively seek them out and exploit them.

Feedback and Direction

Giving feedback and direction to players is straightforward in scripted games as the developer knows when and how the player will interact with various game elements. As the desired outcome is known, it is straightforward to give players feedback on their success at performing actions or fulfilling goals.

Providing feedback and direction is non-trivial in emergent games. Players have a far greater range of possible interactions and actions and there is a lot more uncertainty in the game world. Due to the openness of emergent game worlds and the space of possibilities, players also have a greater need for feedback on the outcome and success of their actions. Players need more feedback to know that they are on the right track and that their actions are successful. Consequently, the problem in giving feedback and direction in emergent games is two-fold; players require more feedback, but the feedback is much more difficult to give.

KEY TERMS

- *Creative control* is the control and certainty that game designers have in defining and creating interactions, narrative, and game progression.
- *Design, implementation, and testing* is the effort required in planning, implementing, and testing a game system.
- *Modification and extension* is the effort that is involved in modifying and extending an existing game system.
- *Uncertainty and quality assurance* include the issues related to ensuring that a game will behave as expected and within reasonable limits to allow quality to be controlled.
- *Feedback and direction* involve giving players feedback on the success of their actions and directions on how to proceed in the game.

EMERGENT GAMES

The future of game development is toward more flexible, realistic, and interactive game worlds. Games have become increasingly more realistic visually, with graphically lifelike and detailed characters, creatures, and game worlds. However, the environments, objects, and agents in these game worlds are often static, lifeless, and afford limited interaction. Players are now seeking more realistic and interactive behavior from these game elements. Consequently, it is now necessary to search for a new approach to game design that will allow game worlds to accommodate the needs of the players, affording more flexible and interesting behavior and gameplay. Emergent games are the next step in game development.

The ways that emergence can be incorporated into games depends on the genre of the game and the level of creative control that is required. Emergence can be incorporated into games via emergent objects, agents, or entire game worlds. Alternatively, game narrative could be made emergent (as a function of the player's interactions in the game world), as could conversations with game characters and game quests, objectives, and puzzles.

Role-playing games could include emergence in the form of characters that have general rules for behavior, conversation and goals, rather than specifically scripted dialogue. First-person shooter games could include emergent objects, enemies, and buildings. Strategy games can include emergent environmental effects, as well as active and reactive buildings, units, and terrain.

Summary

This chapter has identified and discussed four key areas that hold potential for developing emergence in games—game worlds, characters and agents, narrative, and social systems. The remainder of this book will be dedicated to expanding these areas and giving specific examples and frameworks for introducing emergence into these areas of games.

Game worlds can be divided into the game environment (that is, the physical space) and the game objects (the entities that exist in the game environment). Chapter 6 examines the concept of an "active" game world, in which the environment and objects are active and reactive to players, as well as other elements of the game world. I'll present a framework for developing an Active Game World, using a cell-based world model and property-based game objects.

For agents in games to display emergent behavior, they require a way to sense and model the environment, as well as simple rules for reacting and acting in the game world. Chapter 7 discusses methods that agents can use to sense and model their environment. Subsequently, I present a method that allows individual agents to use their environmental model to guide their movement, as well as a simple, flexible, general-purpose framework that can be used for agent decision-making. Finally, I explore methods for achieving emergent group movement and group tactics in games using an agent-based approach.

Players can become the creators of emergent game narrative, by creating their own storyline and engaging in emergent conversations with game characters. Chapter 8 explains a few simple ways to achieve emergent narrative in games, using the narrative elements of storyline and conversation. I present a framework for developing an emergent storyline, as well as a conversation system for enabling emergent narrative through conversation.

Social emergence is prevalent in massively multiplayer online games, in the form of emergent economies, social structures, and communities. Chapter 9 examines the emergence of economies and social structures in games and the various forms of game communities, as well as the considerations and potential for developers. Finally, I present a model for creating artificial social networks in single-player games and the emergence of artificial social communities.

Before you enter into the specifics of each area of emergence in games, you will first take a look at some algorithms and techniques that can be used to create emergent behavior. Chapter 5 describes the design and implementation of various programming techniques from complex systems, artificial life, and machine learning. These techniques will be used as the basis for the models and frameworks presented in later chapters. You will also read about the considerations of choosing the right technique for the right application.

Class Exercises

1. Think of a board game or card game you have played.
   a. What are the components (board, cards, pieces, and so on) and rules of the game?
   b. How many possible configurations of the board or cards are there? How many different ways are there to reach each configuration?
   c. What are some of the common strategies people use for winning this game? How do you know who is winning in a given turn of this game? Is this game emergent?
2. Think of the game worlds in the games you have played.
   a. What makes up the game world? What constitutes the environment and the objects?
   b. How do you interact with the environment and objects in these game worlds?
   c. What are the limitations to your interactions? How could you make the game world more active and interactive?
   d. Have there been elements of emergence in any of these game worlds? How have you been able to use objects to create new gameplay and strategies?
3. Think of the characters and agents in a game that you have played.
   a. What actions did individual characters perform in this game and what did the characters need to be aware of in the game world to perform these actions?
   b. What group actions did agents perform in this game? What did the agents need to know about their fellow agents to carry out these group actions?
   c. Did any of these characters or agents display emergent behavior? How?
4. Think of games you have played in which the narrative put you in the role of receiver, discoverer, and creator.
   a. How was the storyline conveyed to you in each of these games?
   b. What was your role in the story? How much control or influence did you have over how the story played out? Have you seen any examples of emergent narrative in games?
   c. List all the ways you participated in conversations with characters in games.
   d. Which was the most enjoyable and believable? How would you improve conversations in games? How could emergence be used to improve conversations?

5. Think about the massively multiplayer games you have played or are familiar with.
   a. What do you think about trading and money conversion in online games? How does it affect gameplay? What could developers do to benefit from player trading?
   b. What types of emergent social structures have you seen in these games? How has this changed the gameplay?
   c. What types of game communities have you seen or participated in? What could developers do to better support or facilitate these communities?
   d. To what extent do you think artificial social networks could exist in games? How could these networks be used to improve gameplay?

This page intentionally left blank

5 Techniques for Emergence

In This Chapter

- Linear Techniques
- Approximate Reasoning
- Machine Learning
- Complex Systems
- Artificial Life
- Choosing a Technique

Chapter 4 identified four key areas of games that hold potential for emergent gameplay—game worlds, characters and agents, narrative, and social systems. The remainder of this book will be dedicated to expanding these areas and giving specific examples and frameworks for introducing emergence into these areas of games. Before you enter into the specifics of each area of emergence in games, you will first take a look at some algorithms and techniques that can be used to create emergent behavior.

This chapter discusses various programming techniques and algorithms from fuzzy logic, complex systems, artificial life, and machine learning that can be used to create emergence in games. It also outlines some traditional techniques that are prevalent in current games. The design, application, and considerations of using these techniques in games will be discussed. The basic techniques outlined in this chapter will be used as the foundation for the models and frameworks presented in later chapters. The chapter also covers the considerations of choosing the right technique for the right application.

Techniques that are given the boundaries for behavior (rather than the script) or are able to grow and change have the potential to give rise to behavior that may not have been foreseen (or expected) by the developers. Emergent behavior occurs when simple, independent rules interact to give rise to behavior that was not specifically programmed into the system. Techniques that can be used to facilitate emergent behavior come from complex systems, machine learning, and artificial life. Some examples of techniques that can or have been used in games to facilitate emergent behavior are decision trees, neural networks, flocking, evolutionary algorithms, and cellular automata.

Decision trees are algorithms with a tree-like structure that are used for learning, classification, and decision-making. Neural networks are machine-learning techniques inspired by the human brain that are used for prediction, classification, and decision-making. Flocking is an artificial life technique for simulating the natural behavior of a group of entities, such as a flock of birds or school of fish. Evolutionary algorithms are techniques for optimization and search that use concepts from natural selection and evolution to evolve solutions to problems. Cellular automata are spatial, discrete time models that are used to simulate complex systems. Each of these techniques are described in this chapter and discussed in terms of their application to games.

Before you get into the techniques for emergence, the chapter will first have a look at some traditional techniques that are commonly used in games—finite state machines and scripting. It will also discuss some techniques that span the boundary between determinism and emergence—fuzzy logic and fuzzy state machines. Each of these techniques has a place in game development, which will not quickly be surrendered to complex (and unnecessary) algorithms. The aim is to make the best use of the tools at hand to facilitate emergence in a way that will suit game development and enhance gameplay. The last part of this chapter overviews considerations that need to be made when choosing a technique and summarizes the techniques presented in this chapter.

LINEAR TECHNIQUES

Almost every commercial computer game uses scripting or state machines for some, if not all, of the game system. These techniques are simple, proven, and deterministic. They also require everything to be built into the system during development, which means that the system can only behave as it has been told to behave with no room for adaptation or unexpected behavior. Scripting and finite state machines can, however, both be used as simple components of a system that allows emergent behavior, when used in combination with other techniques or more advanced structures. This section outlines the design, application, and considerations of using scripting and finite state machines in games.

Scripting

A scripting language in a game creates a high-level interface to the game engine so that low-level code, such as C or C++, does not need to be programmed to control the objects in the game. The script can be compiled into straight code, or interpreted by an interpreter in the game engine. Scripting languages lend themselves to early prototyping and rapid content creation. The scope of a scripting language can vary significantly, depending on the problems it is designed to solve, ranging from a simple configuration script to a complete runtime interpreted language.

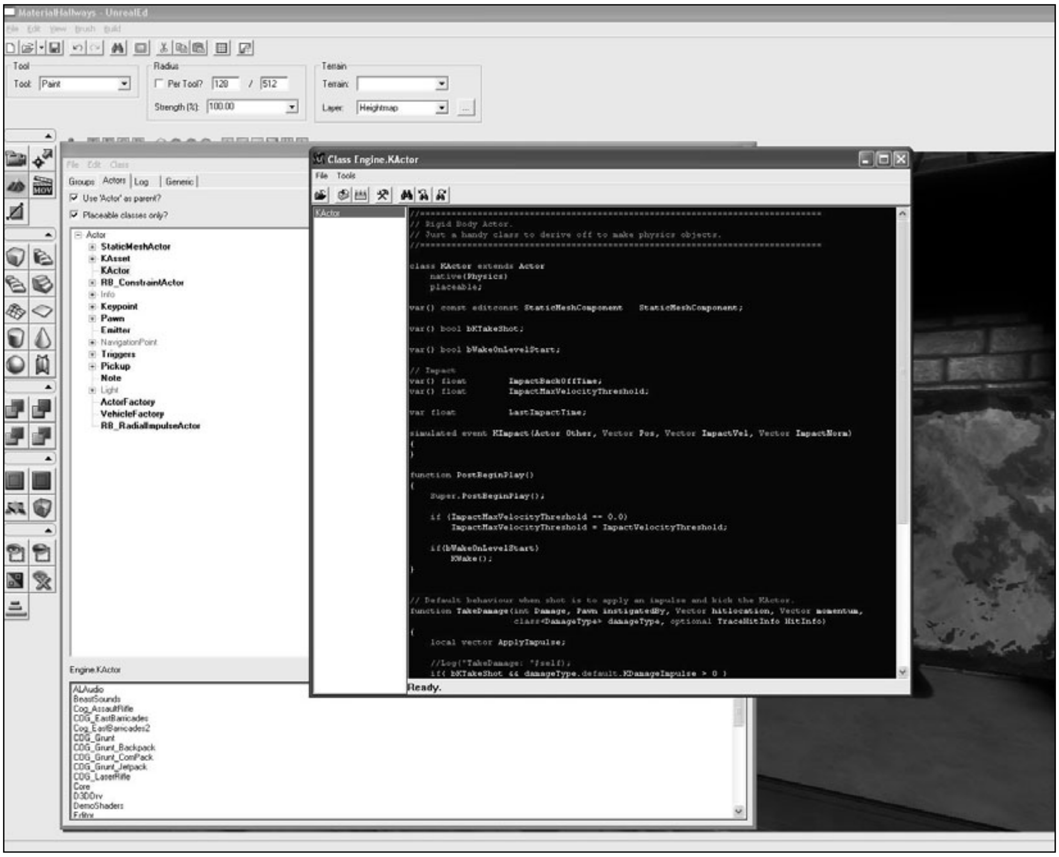
Scripting languages for games, such as *Quake's* QuakeC or *Unreal's* UnrealScript (see Figure 5.1), allow game code to be programmed in a high-level, English-like language. Scripting languages are ideal for games as they are suitable for non-programmers, such as designers, artists, and end users. Scripting allows designers and artists to implement sections of the game independently of the game programmers, and end users can make their own mods for the game. Also, scripting languages are generally separate from the game's data structures and codebase, providing a safe environment for non-programmers and end users to make changes to the game. Script can generally be reloaded without rebuilding or even exiting the game, has a fast turnaround, and is good for prototyping.

The uses of scripting languages in games vary from simple configuration files to entirely script-driven game engines. The common uses include creating events and user interfaces, storytelling, and controlling game characters and enemies. A first-person shooter game could use scripting to create a monster's AI. Alternatively, a real-time strategy game might use scripting to define how spells function or to define a quest or part of the game story. In role-playing games, scripting can be used to define simple conversation trees for a non-player character. A scripting language could even be a complicated object-oriented language that controls every aspect of gameplay.

Many commercial games use scripting to some degree and most developers report success when they customize their own scripting tools. Games that have successfully used scripting, whether it was a custom-made scripting language or an off-the-shelf language, include *Black & White*, *Unreal*, *Medieval II: Total War*, and most of the games developed by BioWare.

The game *Black & White* uses a custom scripting language to present a set of "challenges" to the player. The challenges serve to advance the storyline, give players an opportunity to practice their skills, and entertain the players. The *challenge* language allows designers to implement the logic and cinematic sequences for the challenges, as well as to experiment independently of the programmers.

Scripting is used in *Medieval II: Total War* to define the behavior of computer-controlled forces in historical battles, to guide players through the tutorial (see Figure 5.2), to control action, narration, and camera movement during in-game cinematics,



```
//=============================================================================
// Rigid Body Actor.
// Just a handy class to derive off to make physics objects.
//=============================================================================

class KActor extends Actor
    native(Physics)
    placeable;

var() const editconst StaticMeshComponent   StaticMeshComponent;

var() bool bKTakeShot;

var() bool bWakeOnLevelStart;

// Impact
var() float         ImpactBackOffTime;
var() float         ImpactMaxVelocityThreshold;

var float           LastImpactTime;

simulated event KImpact(Actor Other, Vector Pos, Vector ImpactVel, Vector ImpactNorm)
{
}

function PostBeginPlay()
{
    Super.PostBeginPlay();

    if (ImpactMaxVelocityThreshold == 0.0)
        ImpactMaxVelocityThreshold = ImpactVelocityThreshold;

    if(bWakeOnLevelStart)
        KWake();
}


// Default behaviour when shot is to apply an impulse and kick the KActor.
function TakeDamage(int Damage, Pawn instigatedBy, Vector hitlocation, Vector momentum,
                    class<DamageType> damageType, optional TraceHitInfo HitInfo)
{
    local vector ApplyImpulse;

    //Log("TakeDamage: "$self);
    if( bKTakeShot && damageType.default.KDamageImpulse > 0 )
```

FIGURE 5.1 *Unreal's* UnrealScript allows game code to be programmed in a high-level, English-like language. Copyright © 1998-2007. Epic Games, Inc. All Rights Reserved. Unreal and Unreal Editor are Trademarks or Registered Trademarks of Epic Games, Inc.

to define historical events that take place during the campaign, and for various other applications. The mod community also uses the *Total War* scripting system to redefine large portions of the gameplay to suit the specific mods that they are creating.

The games developed by BioWare using their Infinity Engine, including *Baldur's Gate*, *Baldur's Gate II, Planescape: Torment*, and *Icewind Dale*, all use a custom scripting language, called BGScript. BGScript implements a very simple syntax in which the scripts consist of stacked if/then blocks with no nesting, loops, or other complicated structures. It was designed fundamentally as a simple combat scripting language. However, it was also used for simple, non-combat creature scripting, trap and trigger scripting, conversation, and in-game movies.

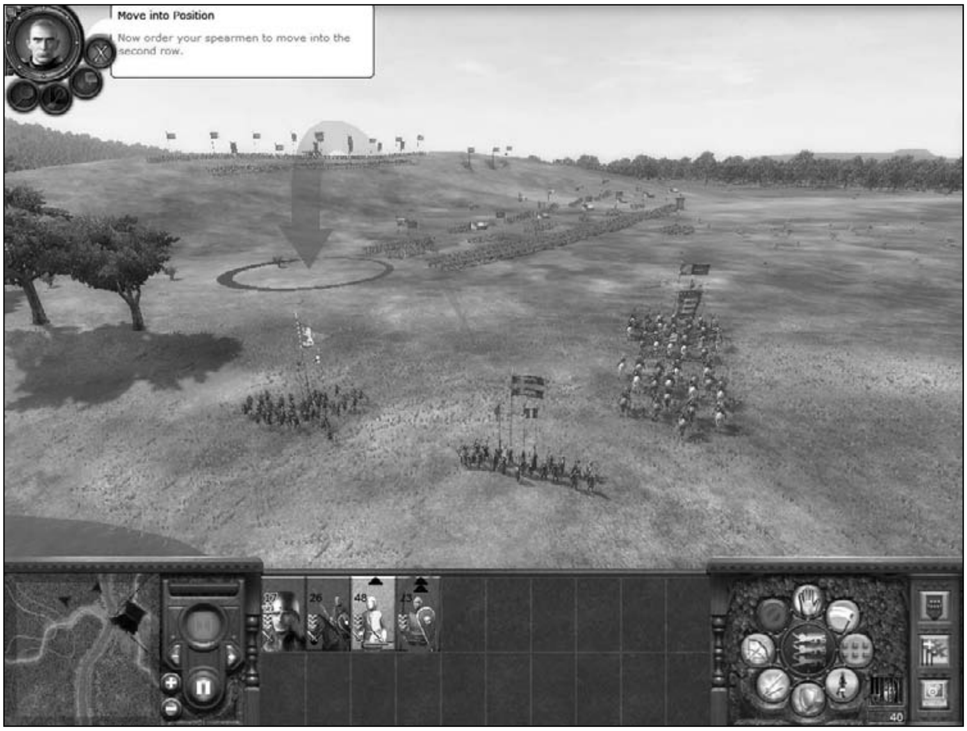


FIGURE 5.2 The tutorial in *Medieval II Total War* was created using the Total War scripting language. © The Creative Assembly. Used with permission.

BioWare's *Neverwinter Nights* uses a scripting language called NWScript. NWScript was designed to include the features from BGScript, as well as spells and pathfinding around doors. Both BGScript and NWScript were designed to be used by the end user. Also, Bioware's game *MDK2* and the LucasArts game *Escape From Monkey Island* both used the Lua scripting language, which was heavily modified by the game developers to give the desired behavior.

Scripting languages are simple, flexible, powerful, and easy to use for non-programmers. These qualities make them ideal tools for game development. However, they are also deterministic and linear. Scripting is used by game developers to hard-code character behavior, scenarios, and storylines. It is possible, however, to use scripting in conjunction with other techniques to create more complex and dynamic behavior.

ADDITIONAL READING

The following papers provide a more in-depth discussion on scripting in games:

- Barnes, J. (2002). Scripting for Undefined Circumstances. *AI Game Programming Wisdom*. Hingham, MA: Charles River Media, pp. 530–540.
- Berger, L. (2002) Scripting: Overview and Code Generation. *AI Game Programming Wisdom*. Hingham, MA: Charles River Media, pp. 505–510.
- Brockington, M. and Darrah, M. (2002). How Not to Implement a Basic Scripting Language. *AI Game Programming Wisdom.* Hingham, MA: Charles River Media, pp. 548–554.
- Poiker, F. (2002) Creating Scripting Languages for Nonprogrammers. *AI Game Programming Wisdom*. Hingham, MA: Charles River Media, pp. 520–529.
- Stripinis, D. (2001) The (Not So) Dark Art of Scripting for Artists. *Game Developer Magazine*, pp. 40–45.
- Tozour, P. (2002) The Perils of AI Scripting. *AI Game Programming Wisdom*. Hingham, MA: Charles River Media, pp. 541–547.

FINITE STATE MACHINES

A *finite state machine* (FSM) is a device that consists of a set of states, a set of input events, a set of output events, and a state transition function. In an FSM, there are a finite number of states, one of which is the current state. The transition function takes the current state and an input event and returns the new set of output events and the next state. The purpose of an FSM is to divide a game object's behavior into logical states so that the object has one state for each different type of behavior it exhibits.

The possible ways in which to use FSMs in games are endless. They could be used to manage the game world or maintain the status of the game or game objects. An FSM could be used to model unit behavior in a real-time strategy game, parse input from the human player, or to simulate the emotion of a non-player character. When making an FSM for a game, the developer needs to anticipate, plan, and test the elements on which the player's attention might possibly be focused. The more the developer can anticipate, the more immersive the environment will be for the player.

There are at least two types of state machines that need to be used in a computer game. The first deals with the game interface, including whether the game is paused, the mode in which the player is viewing the game, and what the player can and can't see. The second type of FSM deals with what is actually going on in the game. This includes the current state of the environment, objects in the level, objectives completed or failed in the mission, and other variables that are used to guide and challenge the player.

An example FSM for a game could be to represent a monster with emotional states, such as berserk, rage, mad, annoyed, and uncaring (see Figure 5.3). In each of these states, the monster would do something different to reflect its changing attitude. The FSM would be used to manage the monster's attitude and the transitions between states based on the input from the game. Different inputs could include information about the player's actions, such as whether they have come into view of the monster, attacked the monster, or run away. Also, information about the monster would also be important, such as whether the monster has been hurt or healed. These variables form the input to the FSM and, based on the input values and the monster's current attitude, the monster's attitude will change, or transition, to another state.

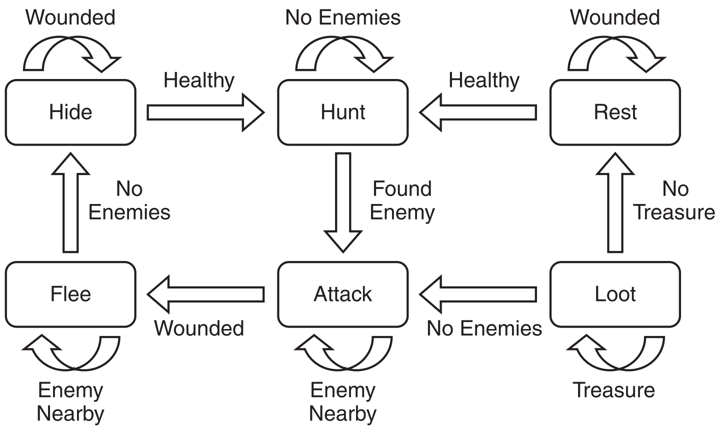


FIGURE 5.3 An example finite state machine for a game character.

Most commercial computer games make use of finite state machines, including *Age of Empires*, *Enemy Nations*, *Half-Life*, *Doom*, and *Quake 2*. *Age of Empires* uses an expert system, combined with some FSMs, for its AI. *Enemy Nations* uses a network of cooperating intelligent agents, FSMs, fuzzy state machines, and a database of goals and tasks. *Half-life* uses an AI architecture called a *schedule-driven state machine,* which is state-dependent and response-driven. Each NPC has different states and different schedules of behavior available from each state.

Quake 2 uses an FSM with nine different states for each character (see Figure 5.4). The states are standing, walking, running, dodging, attacking, melee, seeing the enemy, idle, and searching. In order to form an action, these states may be connected together. For example, in order to attack the player, the states could first go from idle to run, to allow the attacker to get closer to the player, and then switch to attack.

FIGURE 5.4 *Half-Life* uses a schedule-driven state machine for characters. © Valve Corporation. Used with permission.

In games, objects can also run more than one FSM at a time. As it can be difficult to design one FSM that controls everything, it can be quite useful for a game character to run multiple FSMs simultaneously. This is also a good way to limit the size and complexity of each component FSM. One way to structure this is to have a master FSM that makes global decisions and other FSMs that deal with components, such as movement, weapons, and conversation.

FSMs are simple to program, easy to understand and debug, and general enough to be used for any problem. They provide the simplicity of having a choice, weighing the factors, and deciding what to do in the given situation. An FSM may not always provide the optimal solution, but it generally provides a simple solution that works. Also, a game object that uses an FSM can also use other techniques, such as neural networks or fuzzy logic.

Some problems with using FSMs are that they tend to be poorly structured, put together ad hoc, and increase in size uncontrollably as the development cycle progresses. These properties tend to make FSM maintenance very difficult. Also, traditional FSMs are generally cumbersome, redundant, and not very useful for complicated systems. Therefore, FSMs in games tend to include states within states, multiple state variables, randomness in state transitions, and code executing every game tick within a state. Consequently, game FSMs that are not well planned and structured can grow out-of-hand quickly and become very challenging to maintain.

FSMs are one of the most popular techniques used in modern games, because they are easy to understand and program. FSMs are amongst the simplest computational devices and have a low computational overhead. Most importantly, they give a large amount of power relative to their complexity. These attributes make FSMs ideal for the conditions of game development, which involves limited computational resources, as well as limited development and testing time.

In general, games are primarily about creating the appearance or illusion of reality or intelligence. In many games, it comes down to what the players can see and whether they are convinced that the game and characters are behaving reasonably. Often, the use of more advanced algorithms and techniques is not possible, due to computation or other constraints, and in these circumstances a simple solution, such as an FSM, is desirable. If the simplest technique works for the problem, then the use of advanced techniques is not necessary, especially if it won't give better results.

Additional Reading

The following papers provide a more in-depth discussion on state machines in games:

- Dybsand, E. (2000). A Finite-State Machine Class. *Game Programming Gems*. Hingham, MA: Charles River Media, pp. 237–248.
- Rabin, S. (2000). Designing a General Robust AI Engine. *Game Programming Gems*. Hingham, MA: Charles River Media, pp. 221–236.
- Rabin, S. (2002). Implementing a State Machine Language. *AI Game Programming Wisdom*. Hingham, MA: Charles River Media, pp. 314–320.

Key Terms

- *Scripting languages* in a game create a high-level interface to the game engine so that low-level code, such as C or C++, does not need to be programmed to control the objects in the game.
- *Finite state machines* are devices that consist of a set of states, a set of input events, a set of output events, and a state transition function.

Approximate Reasoning

The approximate reasoning used in fuzzy logic and fuzzy state machines allows greater flexibility, variation, and non-determinism than simple linear techniques. Fuzzy logic allows ranges and continuous variables to be used deterministically. The key concepts are the fuzzification and defuzzification of variables, the ability to belong to multiple sets simultaneously and to varying degrees, as well as overlapping sets. The power of fuzzy logic lies in the ability to use a small number of fuzzy variables and rules, in place of the extensive, static states and rules in linear techniques.

Fuzzy Logic

Fuzzy logic allows intermediate values to be defined between conventional values, such as yes/no or true/false. Consequently, “fuzzy” values, such as “rather hot” or “very fast,” that are used to describe continuous, overlapping states, can be used in an exact mathematical way.

In Boolean logic, the sets are mutually exclusive and only one rule can be used. However, in fuzzy logic, every object can belong to all relevant fuzzy sets to various degrees, called the Degrees of Membership (DOM). Each object’s DOM is in the range of zero to one for each set, with the intermediate values generated by a membership function. This graduation allows a smooth overlap of the boundaries between sets. Each object’s inclusion in different sets can sum to more than 100 percent and multiple rules contribute to the output. Consequently, fuzzy logic allows decisions to be made based on incomplete or erroneous data that cannot be used in Boolean logic.

In fuzzy logic, there are operations that are equivalent to those in Boolean logic, namely intersect, unify, and negate. However, these operations are different than their Boolean counterparts. Fuzzy intersection is the minimum of each element from each set, fuzzy union is the maximum of each element from each set, and negation gives the complement (1-x) of each element in each set. Figure 5.5 shows union, intersection, and negation of two sets, A and B, in fuzzy logic.

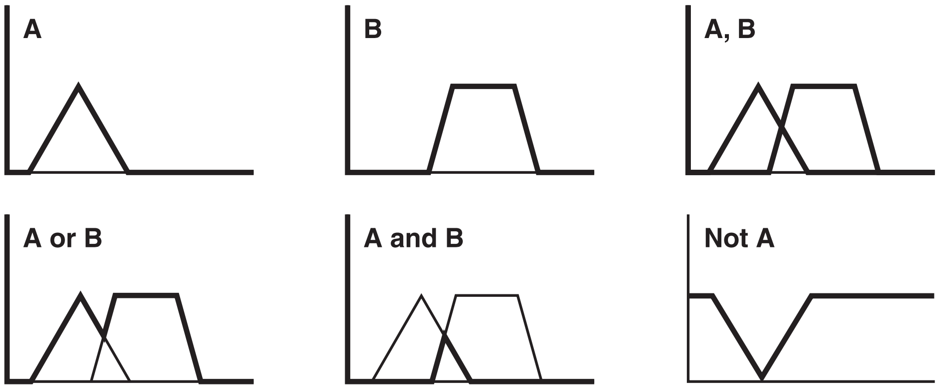
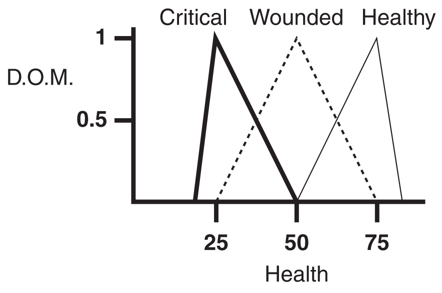


FIGURE 5.5 Union, intersection, and negation of sets A and B in fuzzy logic.

The main difference between Boolean and fuzzy logic lies in the use of Fuzzy Linguistic Variables (FLVs), which define a range of values to be used in place of crisp values. FLVs represent fuzzy concepts that are associated with a range (for example, LOW: 0 to 50, MEDIUM: 30 to 100, and HIGH: 80 to 150). The FLVs overlap slightly to represent the fuzziness of the situation, usually by about 10–50 percent. A small number of FLVs and rules can be used in place of extensive, hard-coded, Boolean rule bases.

Fuzzy rules are evaluated in a number of steps, with the final solution being produced by fuzzification and defuzzification, giving a final crisp value, another continuous value, or a fuzzy value for more fuzzy processing. Fuzzification is the transformation of an objective term into a fuzzy concept. This means that a crisp value is translated into its DOM for each category, thus allowing an FLV in a rule to be interpreted. Figure 5.6 shows an example of fuzzification.



Input (Health)	Degree of Membership		
	critical	*wounded*	*healthy*
20	1.0	0.0	0.0
30	0.7	0.3	0.0
45	0.3	0.7	0.0
60	0.0	0.6	0.4
75	0.0	0.0	1.0
80	0.0	0.0	1.0

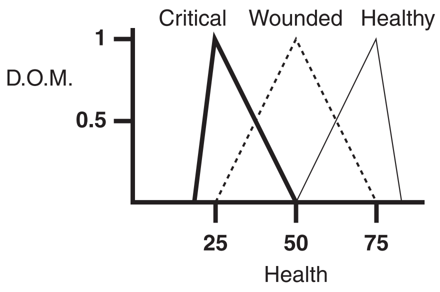
FIGURE 5.6 An example of fuzzification.

After the inputs have been fuzzified, a set of rules, similar to a truth table in Boolean logic, is applied to the values. Each rule consists of an antecedent, interpreted from the fuzzy input sets, and a consequence, interpreted from the fuzzy output set. A set of these rules that represents every combination of inputs is put into a matrix called a Fuzzy Associative Memory (FAM). More than one rule may be true as each input can belong to more than one category of fuzzy set. Figure 5.7 shows an example FAM.

		Health		
		Critical	*Wounded*	*Healthy*
Comparative Strength	*Weaker*	Flee	Flee	Attack
	Equal	Flee	Block	Attack
	Stronger	Block	Attack	Attack

FIGURE 5.7 An example fuzzy associative memory.

Each cell in the FAM contains the output in linguistic terms for the corresponding input. The output of the FAM needs to be translated back into objective terms in order for it to be used in practice. This process is called defuzzification and can be done by taking the maximum value of the outputs from the FAM or by using an averaging technique. Figure 5.8 shows an example of defuzzification.



Degree of Membership			**Output (Health)**
critical	*wounded*	*healthy*	
1.0	0.0	0.0	20
0.7	0.3	0.0	30
0.3	0.7	0.0	45
0.0	0.6	0.4	60
0.0	0.0	1.0	75
0.0	0.0	1.0	80

FIGURE 5.8 An example of defuzzification.

Fuzzy logic makes its way into most computer games, but its role in games usually doesn't exceed complex if-then-else statements, due to the complexity of creating a fuzzy logic system from scratch. A game engine can use fuzzy logic to fuzzify input from the game world, use fuzzy rules to make a decision, and output fuzzy or crisp values to the game object being controlled.

Fuzzy logic is especially useful in decision-making and behavior selection in game systems. For example, fuzzy logic can be used for enemies to determine how frightened they are of the player, for non-player characters to decide how much they like the player, for flocking algorithms to determine how close together the flock should stay, or even for events such as how the clouds would move given the wind speed and direction.

Commercial computer games that have made use of fuzzy logic include *Battle-Cruiser: 3000AD*, *Platoon Leader*, and *SWAT 2*. *BattleCruiser: 3000AD*, developed by Derek Smart, mostly uses neural networks to control the non-player characters in the game. However, in situations where neural networks are not applicable, it uses fuzzy logic. Also, the game *SWAT 2*, developed by Yosemite Entertainment, makes extensive use of fuzzy logic to enable the non-player characters to behave spontaneously, based on their defined personalities and abilities.

The power of fuzzy logic lies in the ability to represent a concept using a small number of fuzzy values, whereas, in Boolean logic, every state and transition needs to be hard-coded. Fuzzy logic is suitable to problems that are non-linear, where no simple mathematical model can solve the problem. However, it's not the best solution if there is a simple way to solve the problem or an existing mathematical model.

Fuzzy State Machines

A fuzzy state machine (FuSM) brings together fuzzy logic and finite state machines (FSMs). Instead of determining that a state has or has not been met, a FuSM assigns different degrees of membership to each state. Therefore, instead of the states on/off or black/white, a FuSM can be in the states "slightly on" or "almost off." Furthermore, a FuSM can be in both the on and off states simultaneously to various degrees. Therefore, in a game situation, a non-player character doesn't have to simply be mad at the player. Instead, they can be "almost mad," "very mad," or "raging mad" at the player, behaving differently in each situation. Thus, by using a FuSM, a character can have varying degrees of membership of a state assigned to it and these states do not have to be specific or discreet.

A FuSM's states are characterized by fuzzy sets, so that each fuzzy state is a state with possible degrees of membership (DOM) between zero and one. This differs from a crisp state, which has an implicit DOM of zero or one. A fuzzy state machine allows the system to be partially in the current state. FuSMs also have fuzzy events, which allow each state to be assigned a DOM. Similarly to an FSM, the next state is a function of both the current state and the fuzzy event. However, FuSMs can allow membership in more than one state at any given time. This means that if the system resides in one state with DOM 0.5, then it may also reside in other states.

In games, it is important that behavior is not overly predictable. However, in FSMs, the requirement of determinism prevents variable behavior from being exhibited, because they are composed of a large set of predetermined states and transitions. On the other hand, FuSMs are composed of fewer, non-deterministic transitions, allowing greater flexibility and variability with far fewer fuzzy states and transitions.

An FuSM is an easy way to implement fuzzy logic, which can allow more depth in the representation of the concepts and relationships between objects in the game world. An FuSM can increase gameplay by allowing for more interesting and varied responses by non-player characters, which leads to less predictable non-player character behavior. Therefore, the player can interact with non-player characters that can be various degrees of mad, wounded, or helpful. This variability increases gameplay by adding to the level of responses that can be developed for the non-player characters and seen by the human player. Also, an FuSM can increase replayability of a game by expanding the range of responses and conditions that the player may encounter in given situations during the game. Therefore, players will be more likely to experience different outcomes in similar situations each time they play the game.

FuSMs can be used in varying forms in different types of computer games. For example, an FuSM could be used in a role-playing game or first-person shooter for the health or hit points of a non-player character or agent. In this case, instead of the finite states healthy or dead, a range could be used for the hit points that would allow the agent to be in the fuzzy states "totally healthy," "almost healthy," "slightly wounded," "badly wounded," "almost dead," or "dead."

In a racing game, an FuSM could be used for the control process for accelerating or braking an AI-controlled car. The FuSM would allow various degrees of acceleration or braking to be calculated rather than the finite states of "throttle-up," "throttle-down," "brake-on," and "brake-off."

An FuSM is also ideal for representing non-player character emotional status and attitude toward the player or other non-player characters. Instead of simply liking or disliking the player, the non-player character could have a range of emotional states from "really liking" or "rather liking" to "slightly disliking" or "violently disliking" the player.

The games that have made use of FuSMs include *Civilization: Call to Power*, *Close Combat 2*, *Enemy Nations*, *Petz*, and *The Sims*. In *Call to Power*, FuSMs are used to set priorities for the strategic-level AI, allowing the creation of new unit types and civilizations. *Close Combat 2* uses an FuSM that weighs hundreds of variables through many formulas to determine the probability of a particular action.

As FuSMs are a combination of FSMs and fuzzy logic, they consist of fuzzy states and fuzzy transitions, rather than the usual finite set of crisp states and transitions. Consequently, FuSMs can represent a greater variation in states and transitions with far fewer variables and rules than in an FSM, where everything must be hard-coded. Most games that make use of FuSMs do so in combination with other techniques such as flocking, FSMs, or neural networks. FuSMs are ideal for controlling the behavior of game characters, giving greater variation in actions and reactions.

ADDITIONAL READING

The following papers provide a more in-depth discussion on fuzzy logic in games:

- Alexander, T. (2002). An Optimised Fuzzy Logic Architecture for Decision-Making. *AI Game Programming Wisdom*. Hingham, MA: Charles River Media, pp. 367–374.
- Dybsand, E. (2001). A Generic Fuzzy State Machine in C++. *Game Programming Gems 2*. Hingham, MA: Charles River Media, pp. 337–341.
- McCuskey, M. (2000). Fuzzy Logic for Video Games. *Game Programming Gems*. Hingham, MA: Charles River Media, pp. 319–329.
- Zarozinski, M. (2002) An Open-Source Fuzzy Logic Library. *AI Game Programming Wisdom*. Hingham, MA: Charles River Media.

KEY TERMS

- *Fuzzy logic* allows intermediate values to be defined between conventional values, allowing continuous, overlapping states to be used in an exact mathematical way.
- *Fuzzy state machines* combine fuzzy logic and finite state machines, so that instead of determining that a state has or has not been met, each state is assigned a degree of membership.

Machine Learning

A key element that distinguishes living organisms from machines is the ability to learn and adapt. Life can react to its environment, learn from its mistakes, and adapt its behavior to perform better in the future. Machine learning is concerned with giving computers that ability to recognize patterns, learn from past events, and adapt output and behaviors. Two common techniques in machine learning are decision trees and neural networks, both of which are used for pattern recognition, learning, classification, and behavior selection.

Decision Trees

Decision trees are techniques used for prediction and classification, which take the form of tree-like structures. Decision trees are used to classify examples into one of a given set of classes. These examples are composed of attribute-value pairs and can be used to describe game characters, objects, or events that the game needs to classify.

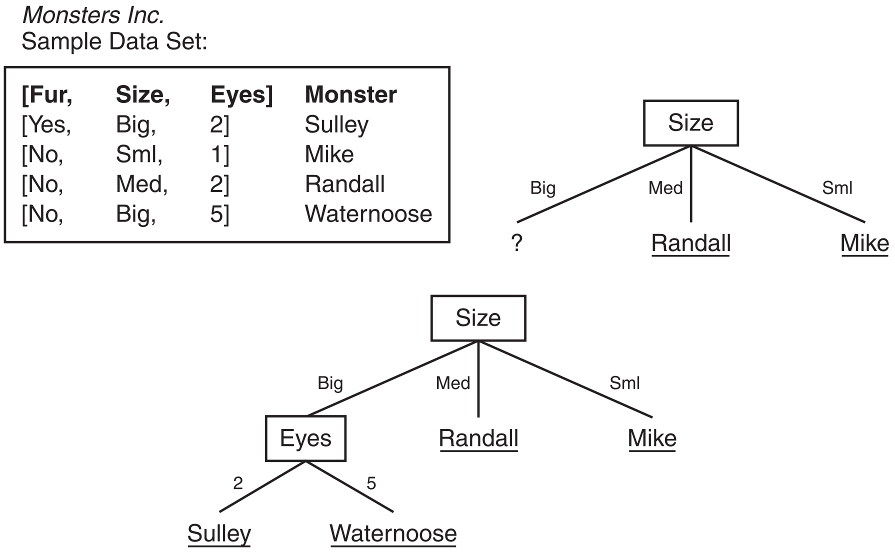
Decision trees are composed of nodes, branches, and leaves, which represent different attributes, possible values of these attributes, and possible classifications, respectively. Decision trees learn by starting at the root node and splitting, or partitioning, the data by the attribute that provides the maximum information gain (see Figure 5.9). The partitions are called *branches*, with the root node encompassing all data records. The root is split into subsets, or child branches, which may be split into sub-branches. This process of splitting the information is repeated until the branches have no more splits, resulting in classification at the leaves.

An algorithm that is commonly used for decision-tree learning is ID3. More recent incarnations of the ID3 algorithm include C4.0, C4.5, and C5.0.

Pseudo-code for the ID3 algorithm is as follows:

```
FUNCTION ID3 (example set, attribute set)
       IF all examples in example set are in the same category
       THEN create a single node tree and label with this category
       ELSE
             node = root
             WHILE example set is not empty
                   split = attribute from attribute set with maximum
                     information gain
                   node.attribute = split
                   FOR EACH value of node.attribute
                         add branch to tree as a node and label with value
                         FOR EACH branch
                               node = branch
                               tree = ID3 (example subset, remaining
                                 attributes)
                         END
                   END
             END
       END
       RETURN tree
END FUNCTION
```

A decision tree is a straightforward description of the splits found by the algorithm. Each terminal or leaf node describes a particular subset of the training data and each case in the training data belongs to exactly one terminal node in the tree. Therefore, exactly one prediction is possible for any particular data record presented to a decision tree.


Monsters Inc.
Sample Data Set:

[Fur,	Size,	Eyes]	Monster
[Yes,	Big,	2]	Sulley
[No,	Sml,	1]	Mike
[No,	Med,	2]	Randall
[No,	Big,	5]	Waternoose

FIGURE 5.9 Decision-tree learning.

After the tree has been generated, it is then used for classifying new examples (see Figure 5.10). Decision trees classify instances by sorting them down the tree from the root node to some leaf node, which provides the classification of the instance. Each node in the tree specifies a test of some attribute of the instance. Each branch descending from that node corresponds to one of the possible values for this attribute. An instance is classified by starting at the root node of the decision tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute. This process is repeated at the node on this branch, and so on, until a leaf node is reached.

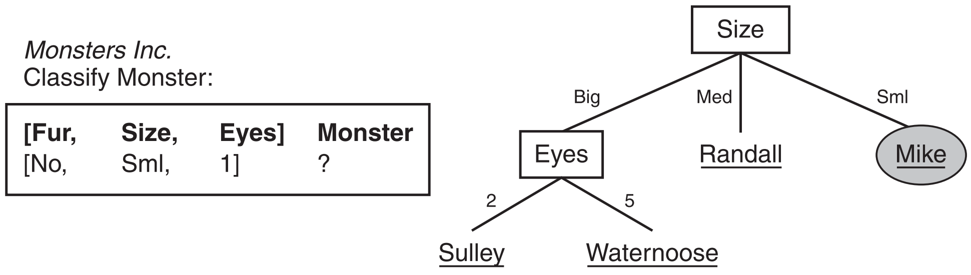


FIGURE 5.10 Decision tree classification.

Pseudo-code for a decision-tree classification algorithm is as follows:

```
FUNCTION classify (example)
       node = root
       WHILE node is not a leaf
              WHILE branch is not equal to example(node attribute).value
                 next branch
              END
              node = branch
       END
       RETURN node
END FUNCTION
```

Decision trees are appropriate for problems in which the instances can be represented as attribute-value pairs. That is, the instances are described by a fixed set of attributes and their values. The ideal situation for decision tree learning is when each attribute has a small number of possible values. Also, decision trees can only be used when the target function has discrete output values. This allows the decision tree to assign a classification to each example, chosen from two or more possible classes.

Decision trees are widely used in data mining, to find relationships in large sets of data and to predict future outcomes. However, their use in commercial computer games has been limited. Decision trees are applicable in games where classification or prediction is required. For example, a character could use a decision tree to learn which of a set of actions will most likely have the best result in different situations. This could be achieved by using the example situations during play to build up a tree and then using the tree to estimate the best action to take. Alternatively, the tree could be pre-built before shipping and simply used for prediction, rather than learning.

Another example would be to allow a character to learn about objects or other characters in its environment. The tree would be built of attributes of objects the character has encountered and their classification, or type. Then, given a new object, the character could predict what the object is and what to do with it.

The game *Black & White* allows the player to have a creature that can learn from the player and other creatures as the game progresses. Each creature has a set of beliefs based on a Belief-Desire-Intention architecture. A creature's beliefs about objects are represented symbolically as a list of attribute-value pairs and its beliefs about types of objects are represented as decision trees.

The creature has opinions about what types of objects are most suitable for satisfying different desires. The creature can learn opinions by dynamically building decision trees. The creature remembers the learning episodes and uses the attributes that best divide the learning episodes into groups. The algorithm used is based on the ID3 algorithm. For example, a creature learns what sorts of objects are good to eat by looking back at its experience of eating different types of things and the feedback it received in each case, such as how nice it tasted. The creatures try to make sense of this data by building a decision tree that minimizes entropy, which is a measure of the degree of disorder of the feedback.

Decision trees are robust in the presence of errors, missing data, and large numbers of attributes. They do not require long training times and are easier to understand than other types of models, because the derived rules have a straightforward interpretation. Decision tree learning methods are robust to errors, both in classification of the training examples and in the attribute values that describe these examples. Also, decision tree methods can be used when some training examples have unknown values.

Decision trees are generally preferred over other non-linear techniques due to the readability of their learned rules and the efficiency of their training and evaluation. Although decision trees have not been used widely in games, they are much simpler to implement, tune, and understand than other learning and classification techniques, such as neural networks. They are ideal for allowing a character to explore and learn about concepts and objects during the game.

Additional Reading

For further information on decision trees in games:

- Fu, D. and Houlette, R. (2004) Constructing a Decision Tree Based on Past Experience. *AI Game Programming Wisdom 2*. Hingham, MA: Charles River Media, pp. 567–578.

Neural Networks

Our brains are made up of about 100 billion neurons, each with up to 10,000 connections to other neurons. Each neuron has four main components: a soma, an axon, buttons, and dendrites (see Figure 5.11). The soma is the cell body of the neuron and the metabolic center, the axon carries the electrical signal from the soma, the buttons are the endings of the axon branches that release chemicals, and the dendrites receive chemical stimuli. The gap between the buttons and dendrites of adjacent neurons are called synapses, which carry chemical signals between neurons. These chemical signals are transferred by accumulation and potential difference of sodium, potassium, and chloride ions.

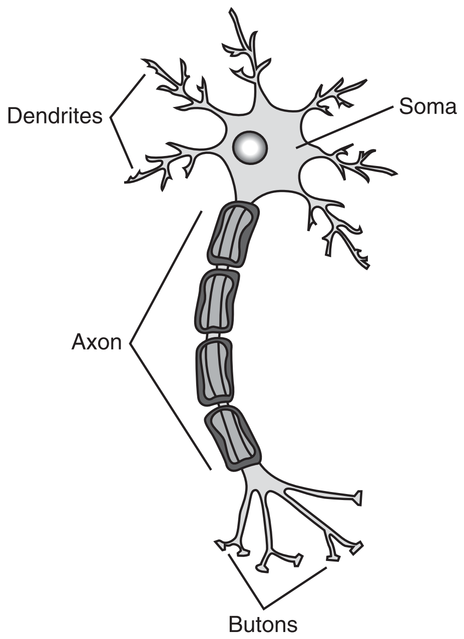


FIGURE 5.11 Neurons consist of a soma, an axon, buttons, and dendrites.

Artificial neural networks are made up of units and weights, which represent biological neurons and synapses, respectively. In an artificial neural network, knowledge is acquired from the environment through a learning process and the network's connection strengths are used to store the acquired knowledge.

Networks generally consist of an input layer, zero or more hidden layers, and an output layer. An example of a simple neural network is shown in Figure 5.12. The input layer consists of units that represent the input to the network. Each input unit has one or more weights that feed into the first hidden layer or the output layer, which provide the excitatory or inhibitory influences of the input unit. The output layer consists of one or more units that comprise the output of the system.

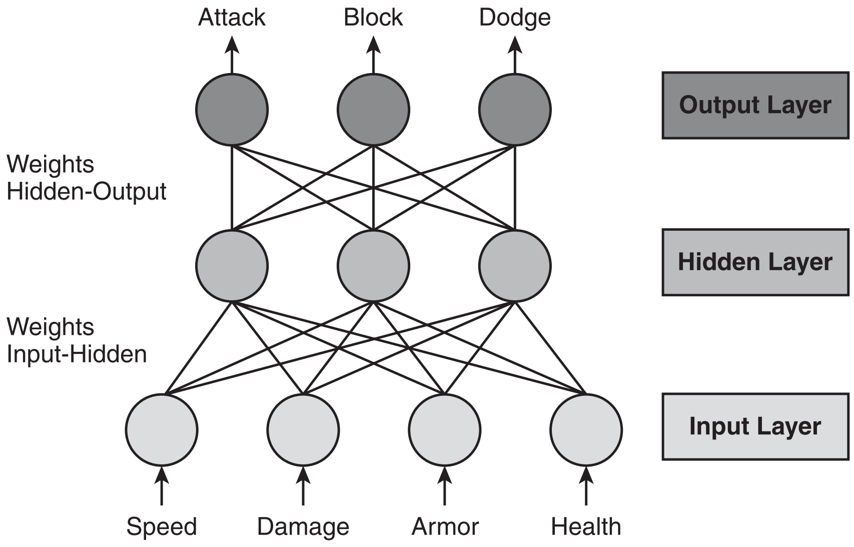


FIGURE 5.12 An example of an artificial neural network.

With one output unit, the system can be used to classify the input as being in one of two categories (for example, true/false). With multiple output units, the system can classify the input into one or more of many categories. For example, with 26 output units, the system could classify the input as being a letter in the alphabet. The hidden layers are used to extract higher order statistics. When there is no direct mapping from input to output, the hidden units provide another dimension for intermediate calculations. Each hidden unit also has an associated set of weights that feed into the next hidden layer or the output layer.

Each unit in the hidden and output layers has an adder for combining its input signals and an activation function for determining whether or not it fires. The adder sums the input values of the unit, where each input value is calculated by multiplying the input unit by its associated weight. This gives the activation of the unit as shown in Figure 5.13, where *a* is the activation, *n* is the number of units, $i_{(1-n)}$ are the inputs and $w_{(1-n)}$ are the weights.

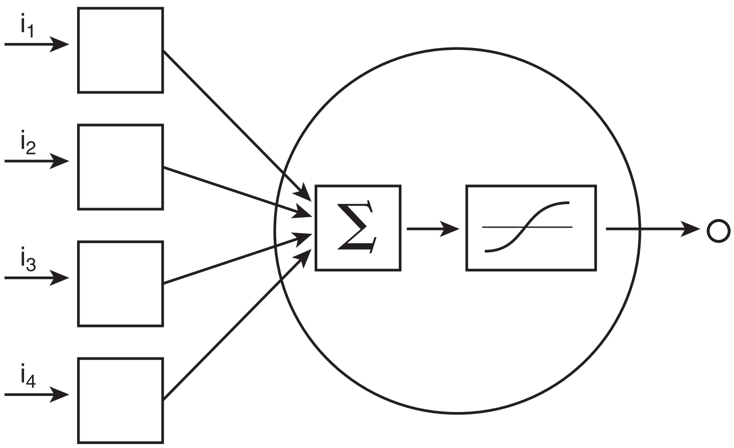


FIGURE 5.13 The activation function in an artificial neuron.

The activation function determines whether or not the unit fires and propagates information, giving the output as shown in Figure 5.14. The four most common activation functions are the threshold, piecewise-linear, sigmoid, and Gaussian functions (see Figure 5.14).

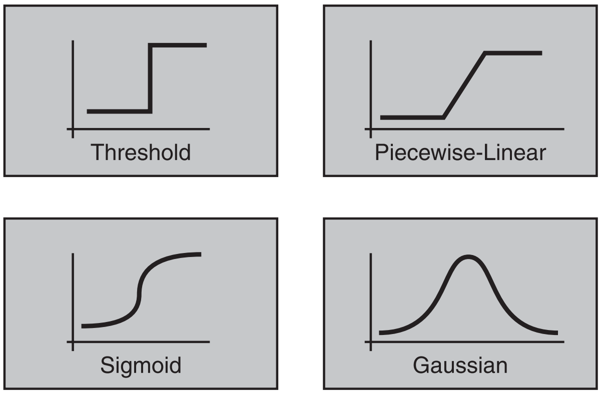


FIGURE 5.14 Common activation functions for a neural network—threshold, piecewise-linear, sigmoid, and Gaussian.

The following pseudo-code implements the propagation and activation of a feed-forward neural network, including summing the input signal to a unit and determining if it fires via the sigmoid activation function:

```
PROCEDURE feedforward( )
       // Propagate input to hidden layer
       FOR EACH hidden unit
              FOR EACH input unit
                    sum of input += this input
                          * weight between this hidden and this input
              END
              // Sigmoid activation function
              hidden activation = 1.0 / (1.0 + exp(-1.0 * sum of input))
              sum of input = 0
       END

       // Propagate hidden to output layer
       FOR EACH output unit
              FOR EACH hidden unit
                    sum of input += this hidden
                          * weight between this output and this hidden
              END
              // Sigmoid activation function
              output activation = 1.0 / (1.0 + exponential(-1.0 * sum of
                input))
              sum of input = 0
       END
END PROCEDURE
```

The most common type of neural network is the feed-forward network, so called because each layer of units feeds its output forward to the next layer. In feed-forward networks, each unit in one layer is connected to every unit in the next layer. There are two types of feed-forward networks, the single-layer and the multi-layer. In the single-layer feed-forward network, the input units map directly to the output layer. In multi-layer feed-forward networks, there are also one or more hidden layers (see Figure 5.12). There are many other types of networks that each perform well on certain problems. Some of the more popular types include Hopfield networks, Kohonen networks, and Hebbian networks (see Figure 5.15).

Neural networks can learn via supervised, reinforcement, or unsupervised learning. In supervised learning, the neural network's weights are initially set to random values and sets of inputs, called the training set, are fed into the system. For each training example, the actual output is compared to the desired output and the weights are adjusted to minimize the difference. The backpropagation learning rule is commonly used for supervised learning.

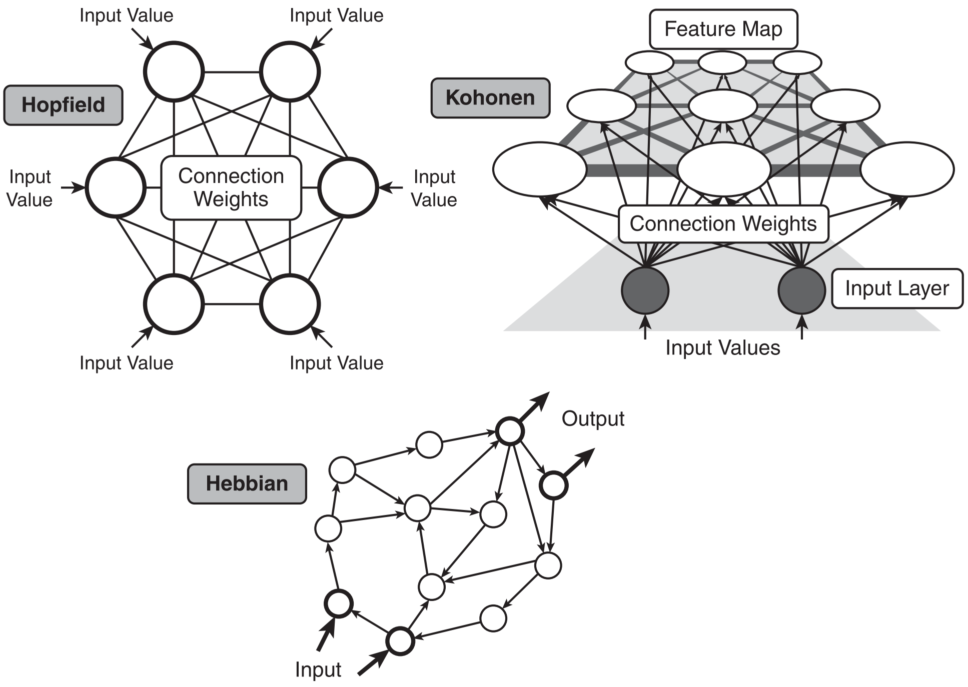


FIGURE 5.15 Hopfield, Kohonen, and Hebbian neural networks.

Backpropagation works by calculating the difference between the actual output (generated by the network) and the expected output (from the training example). The backpropagation learning rule then backpropagates this error back through the system, adjusting each weight according to its influence on the output. As learning progresses, the weights in the network are adjusted to minimize the error. The following pseudo-code implements the backpropagation learning rule:

```
PROCEDURE backpropagation( )
       // Calculate the output error
       FOR EACH output unit
              output error = expected output – actual output
       END
```

```
        // Calculate the hidden error
        FOR EACH hidden unit
               FOR EACH output unit
                      sum += output error * output weight
               END
        END
        hidden error = hidden unit * (1 – hidden unit) * sum

        // Adjust output weights
        FOR EACH hidden unit
               FOR EACH output unit
                      // Calculate change in weight
                      change in weight = (momentum * previous change in
                            weight) + (output error * Output unit)
                      // Calculate new weight
                      new weight = output weight + change in weight
               END
        END

        // Adjust hidden weights
        FOR EACH input unit
               FOR EACH hidden unit
                      // Calculate change in weight
                      change in weight = (momentum * previous change in
                            weight) + hidden error + hidden unit
                      // Calculate new weight
                      new weight = output weight + change in weight
               END
        END
END PROCEDURE
```

In supervised learning, the network is taught to mimic the teacher until the training sample has been learned satisfactorily, at which point the network functions on its own. In reinforcement learning, the network acquires feedback from the environment on its performance, such as whether it won or lost a game. Unsupervised learning involves a task-independent measure of performance, which requires the network to look for statistical regularities in a set of data in order to learn.

Artificial neural networks are flexible techniques that can be used in a wide variety of applications in games, including environmental scanning and classification, memory, and behavioral control. Environmental scanning and classification involves teaching the neural network to interpret visual and auditory information from the environment. Memory involves allowing the network to learn a set of responses through experience and then respond with the best approximation in a new situation. Behavioral control relates to the output of the neural network controlling the actions of game objects, with the inputs being various game engine variables. Neural networks can also be taught to imitate human players.

A neural network can be used to make decisions or interpret data, based on previous input and output or its success in the game. Input to the neural network represents the game state and the output is the decision or action to be performed, similar to a finite state machine. The important difference between a neural network and a state machine is that not every state needs to be predicted and encoded specifically. Instead, a neural network can make an approximation, based on the states that it already knows about. As a result, in a new situation, the network will choose an action that would have been performed in a *similar* state.

Neural networks can be trained in-game (a character can learn from its experiences) or during development (that is, the network is trained on a set of training data created by the developers). In-game learning allows the game to adapt to the player and learn different things depending on individual experiences, but requires computation time for learning. It is also possible that the game will learn undesirable things that the development team can't predict or test. Training the network during development and locking the settings prior to shipping allows thorough testing of game behavior and requires minimal in-game resources for use of the network, but learning and adaptation will not occur. Game developers have been reluctant to use in-game learning due to the possibility of unexpected and undesirable behavior and have preferred to train their networks during development and lock the settings before shipping.

Some examples of games that include neural networks for various tasks include *BattleCruiser: 3000AD*, *Black & White*, *Creatures*, *Dirt Track Racing*, and *Heavy Gear*. In *BattleCruiser: 3000AD*, neural networks are used to control the non-player characters, as well as to guide negotiations, trading, and combat. Neural networks are also used for very basic goal-oriented decision-making and pathfinding, with a combination of supervised and unsupervised learning (see Figure 5.16).

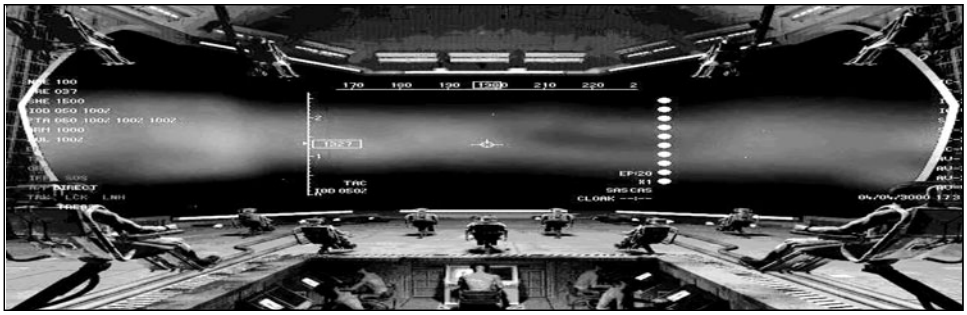


FIGURE 5.16 *BattleCruiser: 3000AD* uses neural networks to control game characters.
© 1996, 3000AD, Inc.

In *Black & White*, the player has a creature that learns from the player and other creatures. The creature's mind includes a combination of symbolic and connectionist representations, with their desires being represented as a neural network. The *Creatures* series of games includes heterogeneous neural networks, in which the neurons are divided into lobes that have individual sets of parameters. In combination with genetic algorithms, the creatures use the neural network to learn behavior and preferences over time. The game *Dirt Track Racing* uses a neural network for driving around the racetrack. Finally, *Heavy Gear* uses a neural network as part of the Mech control mechanisms, with each Mech having several specialized neural networks for particular aspects.

Additional Reading

For further information on neural networks in games:

- Champandard, A. J. (2002). The Dark Art of Neural Networks. *AI Game Programming Wisdom*. Hingham, MA: Charles River Media, pp. 640–651.
- Haykin, S. (1994). *Neural Networks: A Comprehensive Foundation*. New York: Maxwell Macmillan International.
- LaMothe, A. (2000). A Neural-Net Primer. *Game Programming Gems*. Hingham, MA: Charles River Media, pp. 330–350.

Æ

- Manslow, J. (2001). Using a Neural Network in a Game: A Concrete Example. *Game Programming Gems 2*. Hingham, MA: Charles River Media, pp. 351–357.
- Manslow, J. (2002). Imitating Random Variations in Behavior Using a Neural Network. *AI Game Programming Wisdom*. Hingham, MA: Charles River Media, pp. 624–628.
- Sweetser, P. (2004) How to Build Neural Networks for Games. *AI Game Programming Wisdom 2*. Hingham, MA: Charles River Media, pp. 615–625.
- Sweetser, P. (2004) Strategic Decision-Making with Neural Networks and Influence Maps. *AI Game Programming Wisdom 2*. Hingham, MA: Charles River Media, pp. 439–446.

Key Terms

- *Decision trees* are techniques used for prediction and classification, which take the form of tree-like structures.
- *Neural networks* are techniques for learning and decision-making, in which knowledge is acquired from the environment through a learning process and the network's connection strengths are used to store the acquired knowledge.

Complex Systems

A complex system is a system that consists of many interconnected and interdependent parts. The parts themselves may be simple or complex, but the real complexity comes from their interaction. Complex systems are made up of many simpler components, each following a set of behaviors and interacting with its local environment. The result of these behaviors and interactions is far more than what would be expected by examining a component in isolation. The collective behavior of the system is not equal to the simple sum of its parts—it is something that is dynamic, organic, and alive. The complex behavior of the system is said to be emergent, it cannot be simply inferred by the behavior of its components.

Most complex systems have a common set of properties. These properties include elements, interactions, formation, diversity, environment, and activities.

- *Elements* are the basic components of a complex system.
- *Interactions* occur between these elements to give rise to the overall complex behavior of the system.
- The system and its components are formed by some process of *formation*.
- Complex systems have a *diverse* range of behaviors and states.
- Complex systems exist within *environments* that they must respond to and interact with.
- Complex systems carry out *activities* in order to achieve certain objectives or for some purpose.

Cellular Automata

Cellular automata are commonly used to model complex systems. A traditional cellular automaton is a spatial, discrete time model, in which space is represented as a uniform grid. Each cell in the grid has a state, typically chosen from a finite set.

Time advances in discrete steps and at each time step, each cell changes according to a set of rules that represent the allowable physics of the model. The new state of a cell is a function of the previous state of the cell and the states of its neighboring cells.

Cellular automata can be represented in one, two, or more dimensions (see Figure 5.17). A one-dimensional cellular automaton consists of a single line of cells, where the new state of each cell depends on its own state and the state of the cells to its left and right. In a two-dimensional cellular automaton, each cell can have four or eight neighbors, depending on whether cells diagonally adjacent to a cell are considered neighbors.

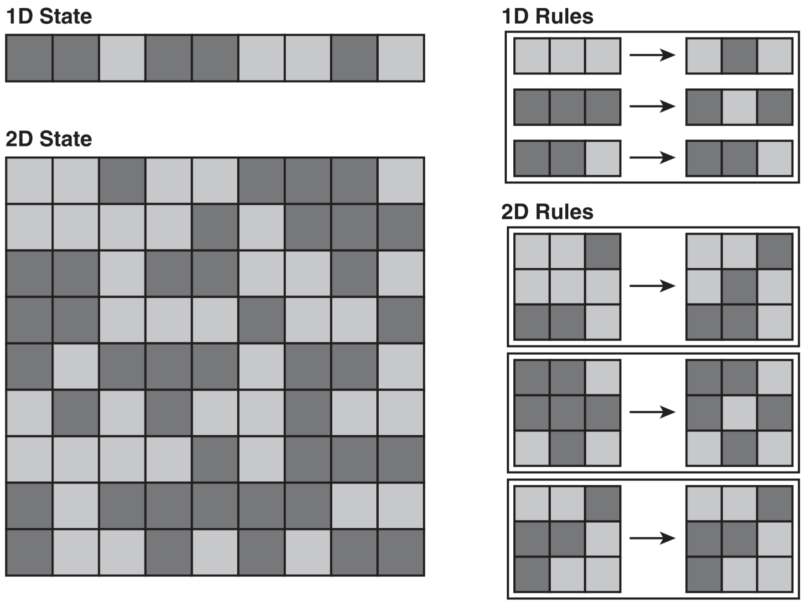


FIGURE 5.17 States and rules in one- and two-dimensional cellular automata.

Most worlds in current games are in three dimensions (3D). Games such as first-person shooters, role-playing games, and racing games play out in 3D, human-sized worlds. Other types of games, such as some strategy games and simulation games, are in two dimensions (2D) or two and a half dimensions. Depending on the types of interactions that will take place in the game world, the environment can be modeled in 2D or 3D. For human-sized games, where gravity and height are important elements, the environment needs to be modeled in 3D. For other games, 2D should be sufficient to get the desired environmental effects.

The set of rules include conditions for when the state of a cell will change, depending on the state of its neighbors. The rules apply only to a cell's immediate neighborhood, but the combination of the rules can produce complex and system-wide effects. The individual rules that are used, even in a very simple cellular automaton, can produce diverse and interesting behavior.

Cellular automata are well-suited to modeling systems and processes with a large number of identical, simple, locally interacting components. They are able to capture the complex, detailed patterns present in many complex systems via a set of simple rules for local interactions. Each of the examples of complex systems in biology, physics, and society discussed in Chapter 2 can be modeled using cellular automata, including the crystallization of snowflakes, self-organization in ant colonies, and the flow of traffic on streets and highways (see Figure 5.18).

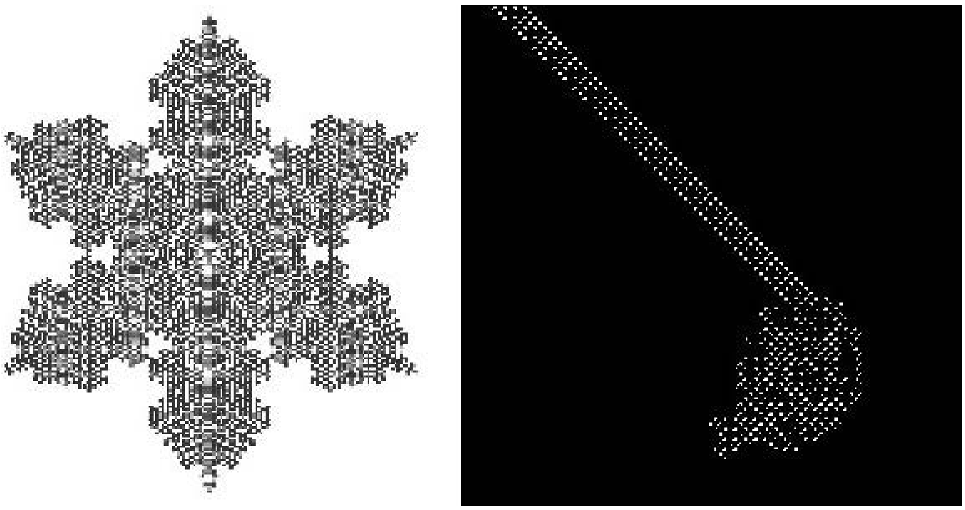


FIGURE 5.18 Cellular automata for snowflakes and ant colonies.

The most common example of complex behavior in cellular automata is Conway's *Game of Life.* The Game of Life is simulated on a nine by nine grid of cells, using a Moore neighborhood (that is, each cell has eight neighboring cells). The grid is first seeded with cells that are either "alive" or "dead." Subsequently, the following three rules are applied simultaneously every time step (see Figure 5.19):

- Birth—A dead cell becomes alive if three of its neighbors are live
- Death—A live cells dies if it has a maximum of one live neighbor (isolation) or if it has more than three live neighbors (overcrowding)
- Survival—Live cells survive if they have two or three live neighbors

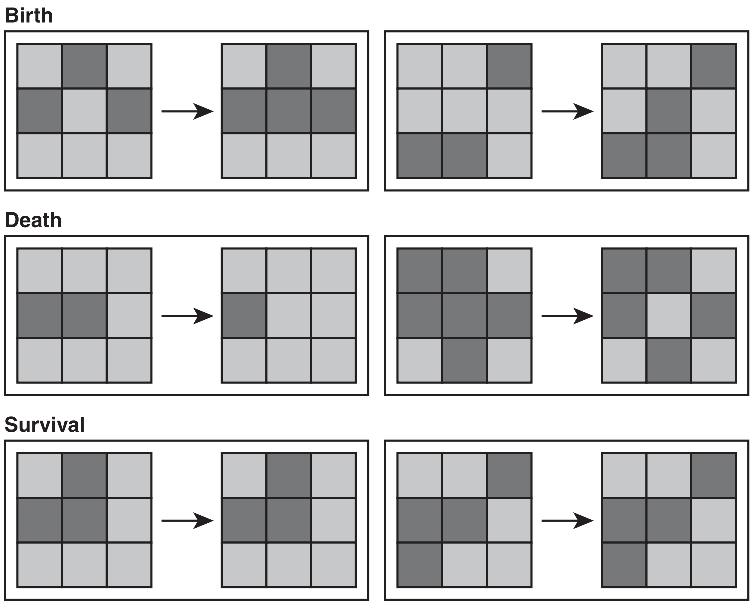


FIGURE 5.19 Rules for birth, death, and survival in the *Game of Life*.

The following pseudo-code implements birth, death, and survival in the *Game of Life*:

```
PROCEDURE game_of_life ( )
       FOR EACH cell
              count = 0
              FOR EACH neighbor of cell
                     IF neighbor is alive THEN
                            count = count + 1
                     END
              END
              IF cell is alive THEN
                     IF count = <= 1 or count > 3 THEN
                            set cell to dead next iteration
                     END
              ELSE
                     IF count = 3 THEN
                            set cell to alive next iteration
                     END
              END
       END
END PROCEDURE
```

Various complex patterns emerge in Conway's *Game of Life,* including gliders (creatures that appear to walk across the grid) and glider-guns (stationary entities that appear to create gliders) (see Figure 5.20).

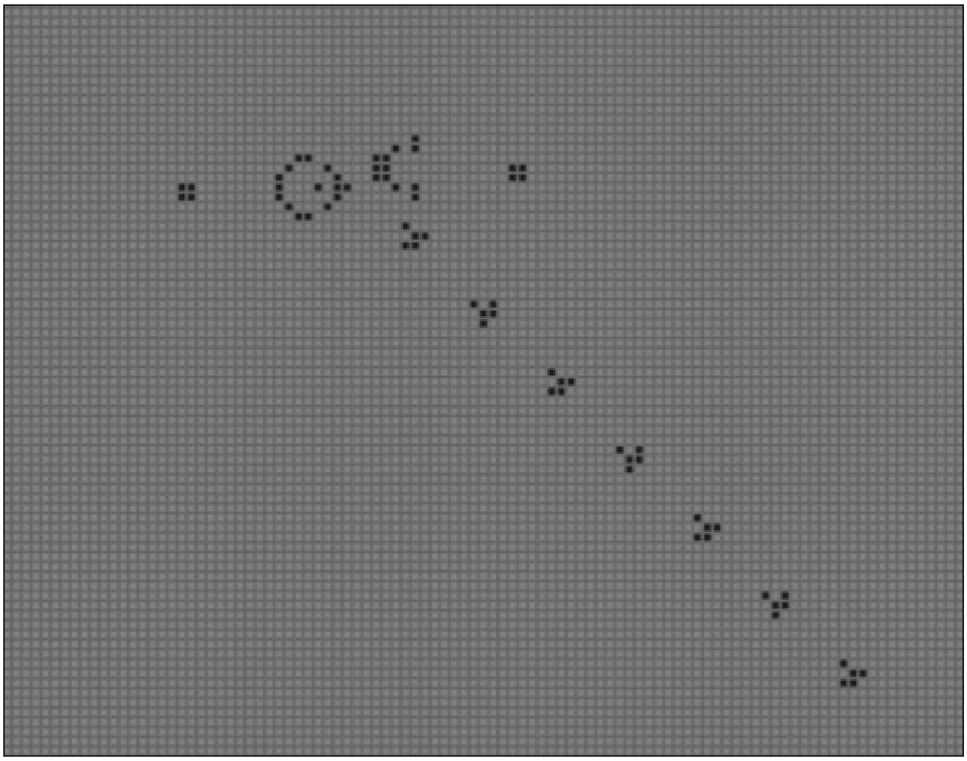


FIGURE 5.20 Emergent patterns in Conway's *Game of Life*—gliders and a glider-gun.

Cellular automata can be used to model environments and environmental effects in games. The use of cellular automata to model game worlds can lead to more dynamic and realistic behavior of many game elements, such as fire, water, explosions, smoke, heat, and social systems. Cellular automata are used in the *X-Com* series of games for environmental modeling, including fire, smoke, dust, gas, and destructible terrain.

The *SimCity* series of games also use cellular automata to model the dynamic state of the simulated cities, including social and physical effects, as well as the development of buildings over time as a function of their neighborhood.

Additional Reading

For further information on cellular automata in games:

- Bar-Yam, Y. (1997) *Dynamics of Complex Systems*. Reading, MA: Addison-Wesley.
- Forsyth, T. (2002) Cellular Automata for Physical Modeling. *Game Programming Gems 3*. Hingham, MA: Charles River Media, pp. 200–214.
- Ilachinski, A. (2001) *Cellular Automata: A Discrete Universe*. Singapore: World Scientific.

Key Terms

- *Cellular automata* are spatial, discrete time models, in which space is represented as a uniform grid.
- *The Game of Life* is a cellular automaton that simulates life, using simple rules for birth, death, and survival.

Artificial Life

Artificial life is the attempt to understand life as it is by examining life as it could be. In artificial life, the way information is organized is as important to life as the physical substance that embodies the information. Life is studied by using artificial components to capture the behavior of living systems. If the artificial components are organized in a way that captures the organization of the living system, the artificial system will also exhibit the same higher-level behavior as the living system.

Systems in artificial life have five general properties:

- A set of simple instructions about how individuals interact
- No master or director that directs the actions of the individuals
- Each instruction defines how individuals respond to their local environment
- No rules that direct the global behavior of the system
- Behaviors on higher levels than the individuals are emergent

Systems in artificial life include a large number of individuals, or agents, that are independently interacting with their local environment and each other.

Through the multitude of simple, local interactions that occur, the collective manages to acquire properties, dynamics, and global behaviors that are not present or predictable on the scale of an individual. The behavior that occurs often seems organized and directed, as though there were some higher power directing the movement of the individuals. The result is complex, self-organizing, and adaptive systems that carry out surprisingly complex and intricate tasks and behaviors.

Flocking

Flocking is an artificial life technique for simulating the natural behavior of groups of entities that moves in herds, flocks, or swarms. Flocking was devised as an alternative to scripting the paths of each entity individually, which was tedious, error-prone, and hard to edit, especially for a large number of objects. Flocking is based on particle systems, which are used to represent dynamic objects that have irregular and complex shapes. Particle systems consist of collections of large numbers of individual particles and have been used to model fire, smoke, clouds, and the spray and foam of ocean waves.

In flocking, the aggregate motion of the simulated flock is created by a distributed behavioral model like that in a natural flock. Each bird in the flock is an individual that navigates according to its local perception of its environment, the laws of physics that govern this environment, and a set of programmed behaviors. Flocking assumes that a flock is simply the result of the interaction between the behaviors of individual birds.

Flocking is an example of emergence; the interaction of simple local rules gives rise to complex global behavior. In flocking, the complex yet organized group behavior comes from the interaction between the simple behaviors of individual boids. Mixing the non-linear component behaviors of the boids gives the emergent group dynamics a chaotic aspect. However, the negative feedback provided by the behavioral controllers tends to keep the group dynamics ordered, resulting in life-like behavior.

In flocking, the generic simulated flocking creatures are called boids. The basic flocking model consists of three steering behaviors (see Table 5.1), separation, alignment, and cohesion, which describe how an individual boid maneuvers based on the positions and velocities of its nearby flockmates. In separation, each member of a flock tries to keep a minimum distance from its neighboring flockmates. It helps to prevent boids from crowding together, while ensuring a lifelike closeness. Each boid of a flock tests how close it is to its nearby flockmates and then adjusts its steering to obtain the desired distance. Alignment involves each member attempting to go in the same direction as its neighbors. Each boid looks at nearby flockmates and adjusts its steering and speed to match the average steering and speed of its neighbors. In cohesion, each member tries to get as close as possible to its neighbors. Each boid examines its neighbors, averages their positions and adjusts its steering to match. Figure 5.21 illustrates the three steering behaviors.

TABLE 5.1 Steering Behaviors in Flocking

Steering Behavior	**Rule**
Separation	Maintain a minimum distance from other boids
Alignment	Match the velocity of nearby boids
Cohesion	Move toward the perceived center of nearby boids

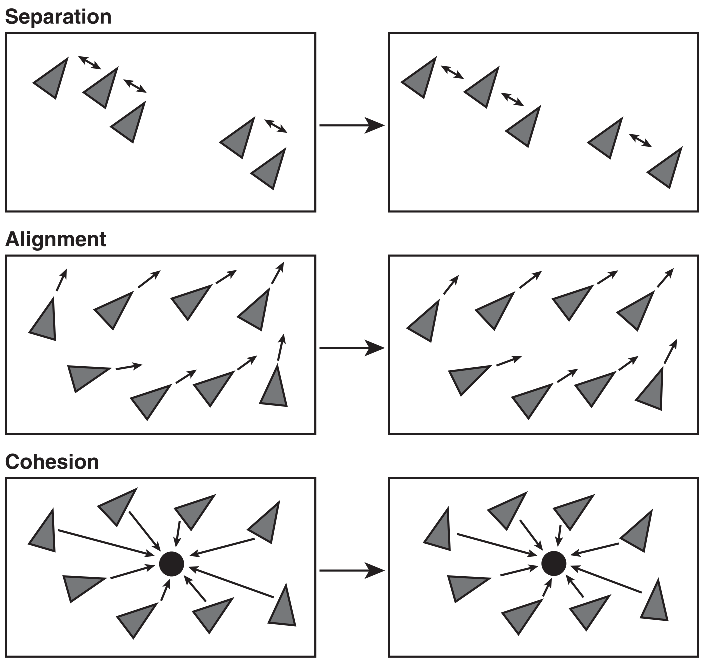


FIGURE 5.21 Rules for separation, alignment, and cohesion in flocking.

The following pseudo-code implements the three steering behaviors of separation, alignment, and cohesion:

```
PROCEDURE flocking ( )
       FOR EACH boid
              sum of mass = sum of mass + position
              perceived velocity = perceived velocity + velocity
       END

       center of mass = sum of mass / number of boids
       average velocity = perceived velocity / number of boids

       FOR EACH boid
              new position = position + separation(boid)
                     + alignment(boid) + cohesion(boid)
       END
END PROCEDURE

FUNCTION separation (this boid)
       separation = 0
       FOR EACH boid
              IF boid is not equal to this boid THEN
                     // minimum separation = 10
                     IF abs(position - boid.position) < 10 THEN
                            // double the separation from boid
                            separation = separation – (position -
                              boid.position)
                     END
              END
       END
       RETURN separation
END FUNCTION

FUNCTION alignment (this boid)
       // move 10% closer to average velocity
       RETURN (average velocity – velocity) / 10
END FUNCTION

FUNCTION cohesion (this boid)
       // move 1% closer to center of mass
       RETURN (center of mass – position) / 100
END FUNCTION
```

A fourth steering behavior, avoidance, can be added to allow a flock to react to predators and obstacles. Avoidance makes each member keep a certain distance from obstacles or members in other flocks, such as predators. This provides a boid with the ability to steer away from obstacles and avoid collisions. Each boid looks ahead some distance and determines whether a collision with some object is likely and adjusts its steering accordingly.

Flocking is a stateless algorithm, because no information is maintained from update to update. Each member in the flock revaluates its environment at every update cycle, which reduces the memory requirements and allows the flock to be purely reactive, responding to the changing environment in real time.

Each boid in the flock has direct access to the whole scene's geometric description. However, flocking only requires the boid to react to flockmates in its local neighborhood, which is characterized by a distance from the center of the boid and an angle from the boid's direction of flight. The flockmates that are outside this local neighborhood are ignored. This neighborhood is the region in which flockmates influence a boid's steering.

There are several constraints that restrict how boids can move and react, namely perception range, velocity, and environment. The perception range is the distance that the boid can look around to detect flockmates, obstacles, and enemies. A flock with a larger perception range is more organized and better at avoiding enemies and obstacles. Whereas a smaller range results in a more erratic flock with groups of boids splitting off more often. The velocity refers to the boids' ability to keep up with their flockmates by how fast they can move and turn. The flock's environment can also impose constraints, such as a size limit or many obstacles or predators.

In simulated flocking (see Figure 5.22), the boids initially move together rapidly to form the flock. As they are flocking, the boids at the edge of the flock either increase or decrease their flying speed to maintain the integrity of the flock. Each boid in the flock makes minor adjustments to its heading as the flock winds its way around. The boids fluidly flock around obstacles in their path, which may temporarily divide the flock, but they are soon reunited. Each boid only perceives its neighbors and their actions and reacts accordingly. However, the collective movement of the boids closely resembles real flocking, even though there are no rules that dictate the behavior of the flock as a whole.

Flocking can be used in games for unit motion and to create realistic environments the players can explore. In a real-time strategy or role-playing game, groups of animals can be made to wander the terrain more realistically than with simple scripting. Similarly, flocking can be used for realistic unit formations or crowd behaviors. For example, groups of archers or swordsmen can be made to move realistically across bridges or around obstacles, such as boulders. Alternatively, in first-person shooter games, monsters can wander the dungeons in a more believable fashion, avoiding players and waiting until their flock grows large enough to launch an attack.

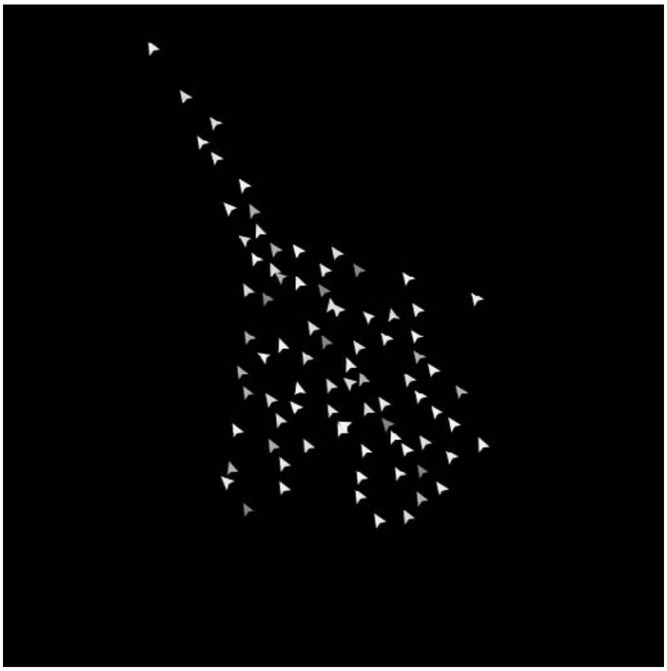


FIGURE 5.22 Simulated flocking.

Many games have successfully used flocking to simulate the group behaviors of monsters and animals, including *Half-Life*, *Theme Hospital*, *Unreal*, and *Enemy Nations*. *Half-Life* uses flocking to simulate the squad behavior of the marines, who run for reinforcements when wounded, lob grenades from a distance, and attack the players with dynamic group tactics. *Theme Hospital* uses flocking to simulate the hustle-and-bustle of patients, doctors, and staff in a hospital. *Unreal* uses flocking for many of the monsters, as well as other creatures, such as birds and fish. *Enemy Nations* uses a modified flocking algorithm to control unit formations and movement across a 3D environment.

Flocking has also been used to simulate crowds of extras and flocks of animals in feature films. The movie *Batman Returns* made use of flocking algorithms to simulate bat swarms and penguin flocks.

Flocking is currently used in games where there are groups of animals or monsters that need to simulate lifelike flock behavior. It is a relatively simple algorithm and only composes a small component of a game engine. However, flocking makes a significant contribution to games by making an attack by a group of monsters or marines realistic and coordinated. It therefore adds to the suspension of disbelief of the game and is ideal for role-playing or first-person shooter games that include flocks, swarms, or herds.

Additional Reading

For further information on flocking in games:

- Reynolds, C. (1987) Flocks, Herds, and Schools: A Distributed Behavioral Model. *Computer Graphics* 21 (4), pp. 25–34.
- Woodcock, S. (2000) Flocking: A Simple Technique for Simulating Group Behavior. *Game Programming Gems*. Hingham, MA: Charles River Media, pp. 305–318.
- Woodcock, S. (2001) Flocking with Teeth: Predators and Prey. *Game Programming Gems 2*. Hingham, MA: Charles River Media, pp. 330–336.

Genetic Algorithms

A Genetic Algorithm (GA) is a technique for optimization and search, which evolves a solution to a problem, in a similar way to natural selection and evolution. A GA includes a population of possible solutions to a problem, referred to as chromosomes, as well as processes that evaluate each chromosome's fitness and select which chromosomes will become parents. The chromosomes that are selected to be parents take part in a process similar to reproduction in which they generate new offspring by exchanging genes. The new offspring also have a chance that they will mutate, similar to natural mutation. As the cycle continues over time, more effective solutions to the problem are evolved.

A simple GA initializes the population with random solutions and evaluates each trial solution by a fitness function. It then erases the poor solutions and copies the remaining good solutions to replace the ones that were erased. Finally, it mixes and matches those copies in a similar way to genetic mutation and recombination. This process is repeated until the desired level of performance is met or until the time limit expires (see Figure 5.23).

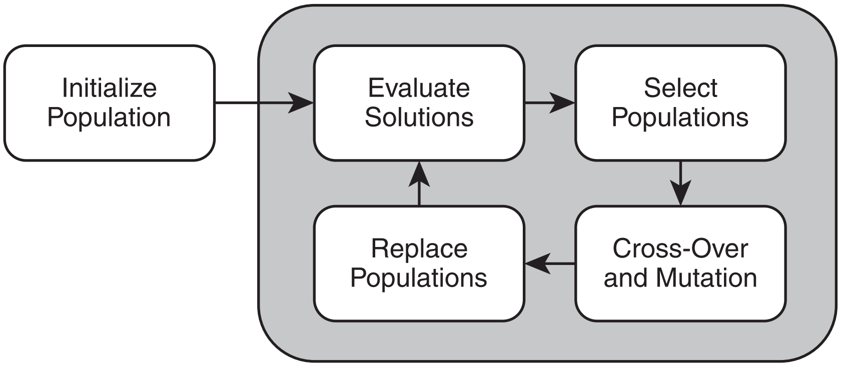


FIGURE 5.23 The flow of an evolutionary algorithm.

It must be possible to represent a single solution in a single data structure that is appropriate to the problem. For example, if optimizing a function of real numbers, then real numbers should be used in the chromosome. Also, the representation should be minimal but completely expressive. It needs to be able to represent any solution to the problem and be designed so that it cannot represent infeasible solutions. Different types of representations include numeric representation, such as an array of real numbers or a string of bits that map to real numbers, a sequence of items implemented as a list or array, or even a tree structure.

A fitness function is used to evaluate the candidate solutions. The choice of fitness function is dependent on the problem. The closer a chromosome is to solving the problem, the higher the fitness score it is given. After the fitness score has been assigned to each chromosome, a method is needed for choosing which solutions will become parents and the number of offspring they will produce. Some of the methods for assigning offspring include Roulette Wheel Selection, Tournament Selection, Linear Ranking, and Stochastic Remainder Selection (see Figure 5.24).

In Roulette Wheel Selection, the probability of being selected to be a parent is proportional to the chromosome's fitness. Each solution is assigned a piece of the "roulette wheel" that is proportional to its fitness value. For each new offspring that is to be created, the roulette wheel is "spun" to determine which of the possible solutions will become a parent. Tournament Selection involves randomly choosing two trial solutions from the population. The better of these two solutions is selected to be a parent, where each solution's probability of being selected as a parent is proportional to their fitness. Linear ranking is where the trial solutions in the population are sorted and given a linearly decreasing number of offspring. Stochastic Remainder Selection involves normalizing the fitnesses so they sum to give the size of the population and the average equals one. This normalized fitness is then used as the expected number of offspring.

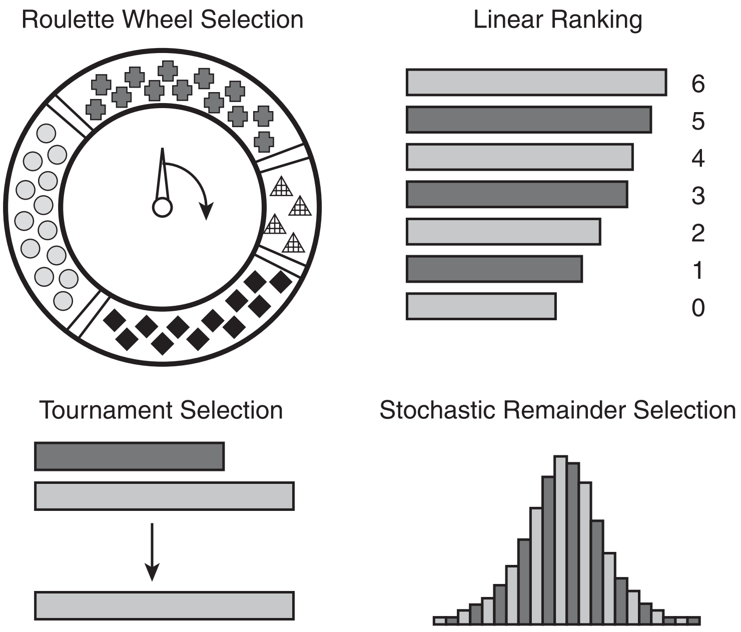


FIGURE 5.24 Methods for assigning offspring in a genetic algorithm.

The following pseudo-code implements the Roulette Wheel Selection algorithm:

```
PROCEDURE roulette_wheel_selection ( )
       // two parents to generate one offspring
       // need to assign twice the number of offspring as parents
       number of offspring = population size * 2
       total fitness = sum of all parent fitness values
       FOR 1 to number of offspring
              // spin the wheel
              r = random number between 0 and total fitness
              sum = 0
              p = first parent
              // find the winner
              WHILE sum < r
                     sum = p's fitness + sum
                     next p
              END
              p gets an additional offspring
       END
END PROCEDURE
```

As well as generating new solutions, it is also necessary to decide how many of the good solutions to copy unchanged. Elitism refers to the copying of the best solutions to keep for the next generation. At one extreme is generational elitism, which is where all parents are replaced by the offspring. At the other extreme is steady state, where only one offspring is created at a time. The most common approach is to copy the best few solutions unchanged to the next generation.

Once it is decided which solutions will become parents, which will be copied unchanged, and which will be erased, it is necessary to determine how to evolve the parent solutions for the next generation. Copies are made of the selected parents to replace the erased non-parents. However, making identical copies won't search new trial solutions so the copies need to be slightly different. This is accomplished through crossover and mutation. The aims are to search the regions of the space that seem to be promising, because good solutions have already been found there, and to search unexplored regions, in case good solutions are located where the GA has not yet looked. However, these aims are mutually exclusive.

The Two-Armed Bandit Problem is an analogy for the process of exploring trial solutions in unexplored regions and explored regions that seem promising. In this problem, there are two slot machines, one which pays with probability less than 50 percent and the other pays with more than 50 percent, but it is unknown which is which. Therefore, they both must be tried and more trials assigned to the one that seems better with increasing probability. This generalizes to the k-armed bandit problem, which is a classical problem in statistical sampling.

In mutation, a small number of random changes take place as too many or too big can be disruptive. However, if there is not enough mutation then the population fills with clones of the same few solutions. In mutation, the variables that need to be tuned are mutation probability per variable, which is how likely each variable is to mutate, and mutation step size, which is the size of the mutation. Figure 5.25 illustrates mutation in a bit string.

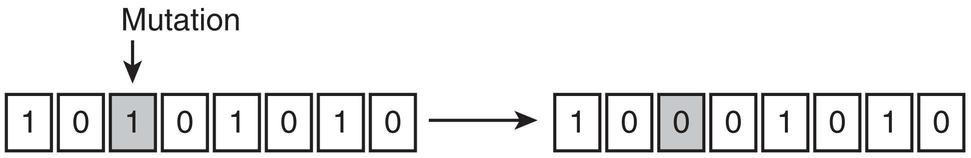


FIGURE 5.25 Mutation in a bit string.

Most problems represent solutions as a vector of numbers, in which the options for mutation include range mutation, Gaussian mutation, and Cauchy mutation. In range mutation, the new value is uniformly random from a set range. In Gaussian mutation, the standard deviation is used and most changes are small.

Finally, Cauchy mutation is similar to Gaussian mutation, but the tail of the bell curve never goes to zero, allowing a few very big changes. The following pseudo-code implements range mutation:

```
PROCEUDRE mutation ( )
       // Mutation chance = 1 in 1000
       FOR EACH offspring
              FOR EACH gene in chromosome
                     r = random number between 1 and 1000
                     IF r == 1 THEN
                            // mutate in range of 1 to 5
                            m = random number between -5 and 5
                            this gene of this offspring += m
                     END
              END
       END
END PROCEDURE
```

Crossover (or recombination) takes two or more solutions and generates one solution. The aim is to use features from both parents and avoid disruption by keeping features intact. If the solutions are represented as an array of numbers then crossover simply involves cut-and-paste. This can be done with one-point crossover, n-point crossover, or in extreme cases, uniform crossover, in which each variable can come from either parent (see Figure 5.26). A random number is generated and if it is over the specified threshold, then the chromosomes crossover at randomly chosen or specified points.

The following pseudo-code implements one-point crossover, using a randomly generated number to determine the crossover point:

```
PROCEDURE crossover ( )
       // Crossover takes two parents and generates one offspring
       WHILE offspring < population size
              p1 = randomly choose parent one
              WHILE number of offspring of p1 = 0
                     randomly choose another p1
              END
              // we found a parent with offspring
              p1 becomes a parent and loses one offspring
              p2 = randomly choose parent two
              WHILE number of offspring of p2 = 0
                     randomly choose another p2
              END
              // we found a parent with offspring
              p2 becomes a parent and loses one offspring
```

```
                i = randomly choose a position along the chromosome
                // parent genes combine to form offspring
                FOR EACH gene in the offspring's chromosome
                       IF this gene's index < I THEN
                             offspring (this gene) = p1 (this gene)
                       ELSE
                             offspring (this gene) = p2 (this gene)
                END
         END
END PROCEDURE
```

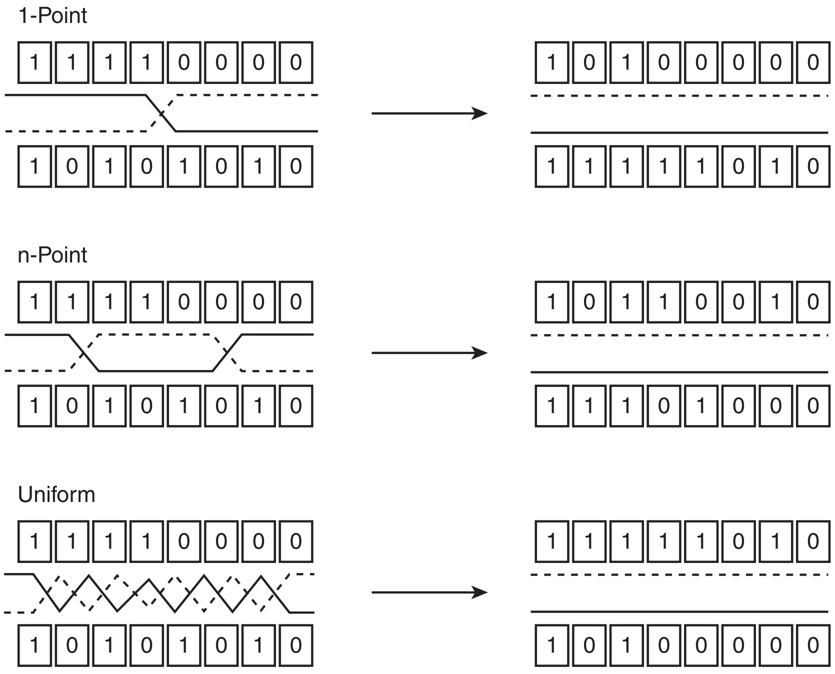


FIGURE 5.26 One-point, n-point, and uniform crossover.

GAs can be used to find non-intuitive, unpredictable solutions to problems in games. A GA could be used in a real-time strategy game to adapt the AI's strategy to exploit the player's weaknesses or to define the behavior of individual units. A role-playing game or first-person shooter could use a GA to evolve behaviors of characters. For example, a GA could take the creatures in the game that have survived the longest and evolve them to produce future generations. This would only need to be done when a new creature is needed. GAs could also be used in games for pathfinding, in which the chromosome could represent a series of vectors and the fitness function could be the distance the sum of vectors is away from a target point.

Computer games that have used GAs include *Cloak, Dagger, and DNA*, the *Creatures* series, *Return Fire II,* and *Sigma*. *Cloak, Dagger, and DNA* uses GAs to guide the computer opponent's play. It starts with four DNA strands, which are rules governing the behavior of the computer opponents. As each DNA strand plays, it tracks how well it performed in every battle. Between battles, the user can allow the DNA strands to compete against each other in a series of tournaments, which allows each DNA strand to evolve. There are a number of governing rules for DNA strand mutation and success, and the player can edit a strand's DNA ruleset. The *Creatures* series of games uses a GA to evolve the creatures (see Figure 5.27).

FIGURE 5.27 *Creatures* uses a GA to evolve the creatures. © Gameware Development. Used with permission.

Considerations that need to be made when designing a GA for a game include the many parameters that need to be tuned, such as choice of a suitable representation, population size, number of generations, choice of a fitness function and selection function, and mutation and crossover parameters.

There are many advantages to using a GA, because they are a robust search method for large, complex, or poorly understood search spaces and non-linear problems. A GA is useful and efficient when domain knowledge is limited or expert knowledge is difficult to encode, because they require little information to search effectively. They are useful when traditional mathematical and search methods fail.

On the down side, a GA is computationally expensive and requires substantial tuning to work effectively. In general, the more resources they can access the better, with larger populations and generations giving better solutions. However, a GA can be used offline, either during development or between games on the user's computer, rather than consuming valuable in-game resources.

Additional Reading

For further information on genetic algorithms in games:

- Laramee, F. D. (2002) Genetic Algorithms: Evolving the Perfect Troll. *AI Game Programming Wisdom*, Charles River Media, pp. 629–639.
- Buckland, M. (2004) Building Better Genetic Algorithms. *AI Game Programming Wisdom 2*. Hingham, MA: Charles River Media, pp. 649–660.
- Sweetser, P. (2004) How to Build Evolutionary Algorithms for Games. *AI Game Programming Wisdom 2*. Hingham, MA: Charles River Media, pp. 627–637.

Key Terms

- *Flocking* is an artificial life technique for simulating the natural behavior of groups of entities that move in herds, flocks, or swarms.
- *Genetic algorithms* are techniques for optimization and search, which evolve solutions to problems, in a similar way to natural selection and evolution.

Choosing a Technique

This chapter provides the basics of various techniques that can be used in games for both simple and complex behaviors. Game development is about creating an experience for the player and how this is achieved is secondary. The right technique should be chosen based on the needs of the situation, including desired results and limitations of the game developer. Research is often about experimenting with new techniques or extending existing techniques to new situations. However, development is about producing the best results within development constraints. In development, you should always start with the problem that needs to be solved, and engineer a solution with the tools at hand.

In this book, you are investigating ways to push current technologies to new levels of realism and immersion. However, the remainder of the text will still adhere to a problem-based approach. You will draw on the basic techniques that have been discussed in this chapter, constructing architectures that draw on multiple techniques in order to achieve the desired results. There are two main components that you need to bear in mind when constructing these frameworks—development considerations and desired gameplay. The resulting solutions are a trade-off between what you want, what you know, and what is possible and achievable.

Development

Chapter 4 outlined some of the primary concerns of game developers when developing new technology and discussed emergence with respect to these issues. The primary concerns of developers include:

- *Creative control*—Level of creative control for game developers
- *Design, implementation, and testing*—Effort in designing, implementing, and testing
- *Modification and extension*—Effort in modification and extension
- *Uncertainty and quality assurance*—Issues for uncertainty and quality assurance
- *Feedback and direction*—Ease of giving feedback and direction to players

Linear techniques tend to give more creative control, whereas emergence involves a more general, open-ended approach. Linear techniques can involve a large amount of time in planning, implementing, and testing specific rules and scenarios. However, emergent systems are risky as they are relatively unproven and unknown techniques, which involve significant testing to ensure stability. Emergent techniques are more conducive to modification and extension, because they are general systems that are extensible by nature. There is certainly more room for uncertainty in emergent systems. However, specific rules and linear techniques can give rise to many exceptions and individual cases to test. Finally, feedback and direction is generally more straight-forward in linear systems as the player's path and possible actions are easier to plan and predict.

When devising a solution to a problem, developers need to consider available programming and testing time, future uses of the technology by the development team and end users, and how much control over the gameplay and story is required by the designers. These constraints will dictate the possible methods that can be used. If the project involves developing a simple, linear game in the shortest time possible, then simple and proven techniques is the obvious choice. However, if you are trying to create more freedom or flexibility and push gameplay to the next level, then you need to explore new options.

GAMEPLAY

Chapter 3 identified the key elements of interacting in games: consistency, immersion, intuitiveness, freedom, and physics:

- Consistency relates to objects behaving in a consistent manner, enabling players to learn the rules of the game and to know when and how they can interact.
- Immersive games draw players into the game and affect their senses and emotions through elements such as audio, graphics, and narrative.
- Intuitiveness is about meeting player expectations, in terms of how they would expect to be able to interact with game objects and solve problems in the game world.
- Players want to be free to express their creativity and intentions by playing the game in the way that they want.
- The physical elements of the game world, such as gravity, momentum, fire, and water, should behave in a way that the player expects.

You also learned how gameplay has progressed over the life of computer games, from very scripted and limited interaction in interactive fiction and linear games to open and immersive play in sandbox and emergent games. This progression has occurred due to a combination of advances in technology and a drive for more realistic and immersive games from players.

The interactivity required in a computer game is an important consideration when choosing the algorithms and techniques that will form the basis of your gameplay or artificial intelligence. How important are each of the key elements of interaction in the game? The type of game that you are creating will also determine the relevance of each issue. Realistic physical modeling is not as important in a turn-based strategy game as it is in a first-person shooter. Immersion is likely to be more important in a role-playing game than in a racing game. Also, the more open and unrestricted the game environment, the more important consistency and intuitiveness become. Large game worlds with many options and interactions need to be intuitive and consistent or players will become confused and lost.

Chapter 3 also discussed the key concepts of enjoyment in games and the eight elements of the GameFlow model: concentration, challenge, player skills, control, clear goals, feedback, immersion, and social interaction:

- Games should require concentration and the player should be able to concentrate on the game
- Games should be sufficiently challenging and match the player's skill level
- Games must support player skill development and mastery
- Players should feel a sense of control over their actions in the game
- Games should provide the player with clear goals at appropriate times
- Players must receive appropriate feedback at appropriate times
- Players should experience deep but effortless involvement in the game
- Games should support and create opportunities for social interaction

Each of these elements contributes to player enjoyment in games, but has varying importance in different games. When choosing models and techniques for your game, it is important to consider how these models will allow you to facilitate the player's enjoyment, in terms of each GameFlow element, and which elements are important and relevant to your game. Concentration, challenge, and player skills are highly important to strategy games, feedback and immersion are vital in first-person shooter games, and clear goals and control are central in role-playing games. Social interaction is not relevant to all games, but it will have a significant impact on your choice of models if it is relevant to your game.

SUMMARY

Each of the techniques outlined in this chapter have advantages, disadvantages, and applications that they are more suited towards. Table 5.2 gives a summary of each of the techniques with respect to these qualities, as well as a list of some of the games that have made use of the technique. Each of these techniques can be used for various purposes in games, separately or in combination with other techniques. Chapters 6 to 9 will investigate different problems in games that could benefit from emergence. Specific models and frameworks are created to solve these problems, drawing on various techniques, in order to facilitate greater enjoyment, immersion, realism, and emergent gameplay in games.

TABLE 5.2 Technique Comparison

Technique	Advantages	Disadvantages	Applications	Games
Finite State Machine	Simple	Can be poorly structured	Manage game world	*Age of Empires*
	General	Poor scaling	Manage objects and characters	*Half-Life*
	Use in conjunction with other techniques	Need to anticipate all situations		*Doom*
	Computationally inexpensive	Deterministic		*Quake*
Scripting	Simple	Deterministic	Events	*Black & White*
	Can be used by non-programmers	Need to anticipate all situations	Opponent AI	*Unreal*
	Safe environment		Tell the story	*Baldur's Gate*
			Automate tasks	
			Conversation trees	
Fuzzy Logic	When no simple solution	Not good when there is a simple solution	Decision making	*SWAT 2*
	When expert knowledge is needed	Complicated to build from scratch	Behavioral selection	*Call to Power*
	Non-linear problems		Input/output filtering	*Close Combat*
	More flexible and variable		Health of NPC	*Petz*
			Emotional status of NPC	*The Sims*
Decision Trees	Robust to noise and missing values	Needs tuning	Prediction	*Black & White*
	Readable		Classification	
	Efficient training and evaluation		Learning	

Æ

Technique	Advantages	Disadvantages	Applications	Games
Neural Networks	Flexible	Needs tuning	Memory	*Black & White*
	Non-deterministic	Choosing variables is difficult	Pattern recognition	*BC3K*
	Non-linear	Complicated	Learning	Creatures
		Resource intensive	Prediction	*Heavy Gear*
			Classification	
			Behavioral control	
Flocking	Purely reactive	Limited applications	Unit motion	*Half-Life*
	Memory requirements		Groups of animals and monsters	*Unreal*
	Realistic and lifelike			Enemy Nations
Genetic Algorithms	Robust search method	Resource intensive	Optimization	*Cloak, Dagger & DNA*
	Effective in large, complex, poorly understood search spaces	Slow	Learning	Creatures
	Non-linear	Needs a lot of tuning	Developing game strategies	*Return Fire II*
	Non-deterministic	Complicated	Evolve behavior	
			Pathfinding	

Class Exercises

1. What would you like to improve in games that you have played? What have you found to be too linear, limited, or synthetic?
   a. How has this aspect of gameplay been limited in terms of consistency, immersion, intuitiveness, freedom, or physics?
   b. How has this aspect of gameplay affected your enjoyment? Which elements of GameFlow do you find are decreased by this aspect of gameplay? How could your enjoyment be increased by changing this aspect?
   c. How could this aspect of gameplay be changed to be more open, emergent, or realistic? How would you like to improve it?

2. How well can linear techniques, such as scripting or state machines be used to improve your chosen aspect of gameplay?
   a. How have these techniques been used in the past for this feature in games?
   b. Are the techniques currently used for this feature a limiting factor?
   c. Could linear techniques be extended or improved to make this feature better?
   d. What are the considerations for gameplay in using linear techniques to implement this feature?
   e. What are the considerations for developers in using linear techniques to implement this feature?
3. Could you use techniques from machine learning, complex systems, or artificial life to improve your chosen gameplay feature?
   a. Which non-linear techniques would be most suited to this problem? Why?
   b. How could non-linear techniques be used to improve this aspect of gameplay?
   c. How would using non-linear techniques for this problem improve gameplay? Would it make the game more interactive? Would it create emergent gameplay? How would it increase your enjoyment of the game?
   d. What would be the considerations for developing these techniques? Have they been used for this purpose before? How much planning, implementation, and testing would be involved? Is the effort expended worth the gain in gameplay?

6 Game Worlds

In This Chapter

- Active Game Environment
- Property-Based Objects
- Emergent Game Worlds

Game worlds are the possibility spaces of games. The space, terrain, objects, physics, and environmental effects dictate the possibilities for actions and interactions that compose and constrain the gameplay. The elements of the game world (such as weapons, chairs, walls, and enemies) are the basic elements of gameplay, similar to the board and pieces in chess. The laws of physics and rules for interaction are the game rules, which constrain the possibility space. Within this space are the allowable actions and interactions of the player.

Interactions in the game world are the foundation of the gameplay. The gameplay is made up of how the player uses the basic interactions to solve problems, achieve goals, and advance through the game. The key to creating emergent gameplay is to define a simple, general set of elements and rules that can give rise to a wide variety of interesting, challenging behaviors and interactions in varying situations. The simpler and more generalizable the rules, the easier they will be to test, tune, and perfect for emergent gameplay.

Game worlds can be divided into two fundamental components—environment and objects. The environment is the space, including boundaries such as terrain, sky, and walls, as well as the physical space (such as air in an earth-based game or water in an underwater game). Game objects are the entities that populate the game world. There are a wide variety of objects in game worlds, which vary by game genre. Together, the environment and objects make up the game world and their properties and behavior determine the interactions that are possible and the resulting gameplay.

This chapter presents a framework for creating emergent game worlds and player interactions in those game worlds. Game worlds can be divided into the game environment (the physical space) and the game objects (the entities that exist in the game environment). The environment in many games is inert and unresponsive to players, objects, and events. This chapter examines the concept of an "active" game world, in which the environment and objects are active and reactive to players, as well as other elements of the game world.

The Active Game World framework uses a cell-based environment model and property-based game objects. In this chapter, the framework is used to create an environmental system for use in a strategy game, which models heat, pressure, and fluid flow.

The Active Game World implements simplified equations from thermodynamics with a cellular automaton. An active environment, based on simple interactions between cells of the environment, provides a foundation for emergent behavior to occur in game objects and agents, as well as the environment itself. Property-based game objects can be integrated into the active environment to create an Active Game World.

Objects in the Active Game World are implemented as though they are cells, using the same low-level properties, based on the object's material. Objects are also imbued with high-level properties, based on their structure, to constrain the possible physical interactions of the objects. The resulting Active Game World model is flexible and extensible, allowing the game world to respond consistently and realistically to a wide range of events and player actions in any situation in the game.

Active Game Environment

The environment is the central component of an emergent game system as it defines the game world and the interactions that are possible within the world. The rules that are defined for the interactions within the environment itself dictate the rules that will apply to entities that exist in the environment, such as objects and agents. Therefore, defining the rules for the behavior of the environment itself is a crucial step in developing a game world that facilitates emergent behavior.

As discussed in Chapter 3, modeling physics in games, such as gravity, momentum, and other basic laws of physics, is important to ensure realistic and consistent movement, interactions, and gameplay. The physical behavior of fire, explosions, and water in games is important to players. Fire should burn, releasing heat and causing damage. Water should flow across surfaces, following contours, and making other substances wet. Pressure should diffuse and large pressure differences should cause explosions.

An active environment, based on simple interactions between cells, provides a foundation for emergent behavior to occur in game objects and agents, as well as the environment itself. One technique that can be used as a foundation for an active game environment is *cellular automata.*

This section discusses the design and implementation of the Active Game World, based on cellular automata, which models basic elements of the environment, such as heat, pressure, and fluid. You'll begin with simplified equations from thermodynamics, implement a two-dimensional cellular automaton and basic strategy game environment, tune the rules and properties until reasonable observable behavior is achieved, and test the system's behavior with possible strategy game scenarios.

STRATEGY GAMES

Cellular automata can be used to model game environments in a variety of game genres, each with individual constraints. However, a strategy game was chosen to demonstrate the Active Game World for several reasons. First, the abstract nature of strategy games means that the rules and properties can be more abstract. More specifically, a strategy game is conducted on a map that represents a world or a large region, with each cell representing an area that covers several kilometers. On this scale, there is no need to model effects such as ripples in water or drops of rain causing splashes. Rather, it is the large-scale effects, such as forest fires and dams bursting that are important.

Second, the environmental interactions are more likely to directly impact on gameplay in a strategy game. As the interactions are inherently on a larger scale, it is more likely that they will have a more significant effect on gameplay. For example, a forest burning down can give the players a way into an opponent's base or destroy a needed resource of wood.

Third, due to the abstract nature of the game map, the world needs to be represented in two-dimensions only, which means the cellular automaton needs to be represented in two-dimensions only. In games such as first-person shooters, the entire three-dimensional world would need to be represented. However, in a strategy game, the important interactions are occurring only on the surface of the world. The system may need to take the height of the surface into consideration, but there is still only one plane of cells that require calculations to be performed.

Fourth, strategy games are almost always divided into grids, called influence maps. As influence maps are already widely used in games, a system that uses influence maps is likely to be easier to implement and integrate into existing systems.

Fifth, strategy game maps are generally much smaller than maps in other types of games. Strategy games usually have one static map that represents the world, whereas other types of games have multiple cities or regions that continually need to be loaded as the player moves into each region.

For these reasons, strategy games have been selected as an ideal environment to demonstrate the Active Game World, because they are far simpler, have more obvious effects, and will involve far fewer issues in implementing and incorporating into current games.

Physical Modeling with Cellular Automata

Most approaches to modeling real-world phenomena in virtual worlds aim to develop accurate, error-free models. These models are usually developed for the purposes of simulating natural disasters (such as forest fires) or visually realistic effects (such as smoke or fluid flow), using complex equations. Equations and models that are commonly used in these applications include Navier-Stokes equations, Euler equations, and the Stable Fluids algorithm.

However, these computationally expensive, complex methods are not needed in game worlds, where the emphasis is on credible and acceptable behavior, rather than accurate and error-free simulation. Game worlds only need to approximately model reality for the purposes of entertainment.

In traditional cellular automata, space is represented as a uniform grid and each cell in the grid has a state, typically chosen from a finite set. In a game environment, the cells in the cellular automaton can include data for a variety of game variables and are represented by continuous values, rather than finite values. In the Active Game World, variables such as the heat, pressure, fluid, and terrain of each cell are tracked and stored in the cellular automaton.

Not only does the cellular automaton store these values, but it also uses equations to determine how the variables change over time. Similar to traditional cellular automata, cellular automata in games update each time step by applying rules for how neighboring cells should interact. In the Active Game World, this involves exchanging heat, pressure, fluid, and other relevant elements, according to a set of rules for the physics of the system.

The rules and variables that the cellular automaton requires depends on what is being modeled in the game environment. For example, to model fire, the cellular automaton will need rules for heat exchange and burning and variables such as temperature, burning temperature, flashpoint, and specific heat capacity. To model fluid flow, the cellular automaton will need rules for how fluid should be exchanged between cells and variables for height, fluid, absorption, and so on.

Forsyth (2002) has identified ways in which environmental processes can be simplified for games using cellular automata and has formulated some example equations for these processes in human-sized (first-person shooter) games. In this section, Forsyth's equations for heat, pressure, fluid flow, and fire are summarized.

Heat

There are three different mechanisms for transmitting heat through the environment: conduction, convection, and radiation. In conduction, neighboring cells pass heat to each other until they reach the same temperature (see Figure 6.1). Convection models the process of heat rising and radiation models the phenomenon that hot objects emit light. Convection and radiation are not relevant to this strategy game model, because it works on a single plain and the emission of light is not important. In the Active Game World, only conduction will be used for modeling heat.

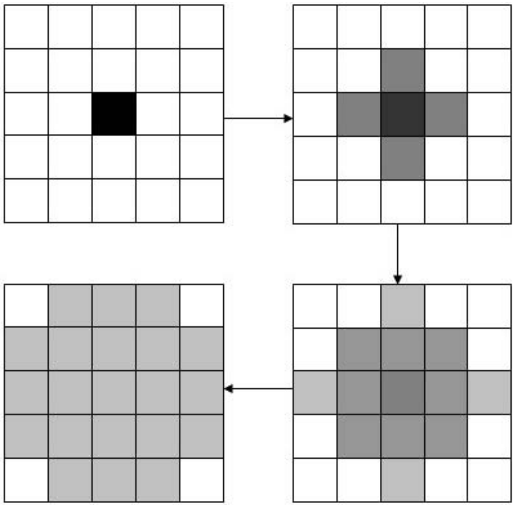


FIGURE 6.1 Conduction.

Pressure

Pressure diffusion calculates the difference in pressure between a cell and its neighbor and divides that amount by the number of neighbors (see Figure 6.2). Consequently, pressure flows from areas with higher pressure to areas with lower pressure, at a rate corresponding to the difference in pressure, until equilibrium is reached.

Fluid Flow

Pressure diffusion can be used as a basis for modeling fluid flow (see Figure 6.3). If the fluid is made compressible, the fluid at greater depth seems to have greater pressure (more fluid stored in the same space). The compression property allows water to

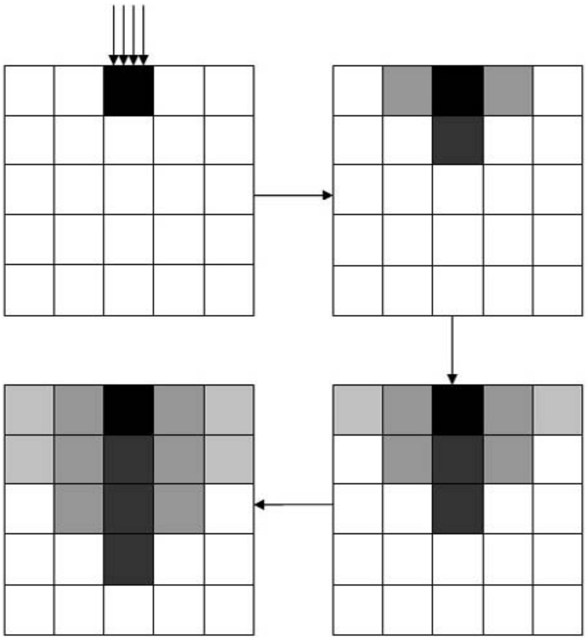


FIGURE 6.2 Pressure diffusion.

behave realistically in three dimensions. In the Active Game World, you are working in two dimensions, so the use of fluid compression to simulate depth is not necessary.

Fire

The process of burning materials is extremely complex. The best results are given by using a function that shows the amount of heat energy that is released per unit of time when a material burns at a certain temperature. In order to model burning in real-time, the materials need to be reduced to their main characteristics. Two key variables, maximum burning rate and burning temperature, can be tweaked to allow the burning of any material to be simulated. See Figure 6.4.

Developing a System

The preceding equations provide a foundation for developing an emergent game environment based on cellular automata. However, to create an emergent game world, they must be developed into a full world model that includes structure, as well as means for integrating the individual components and updating the game world.

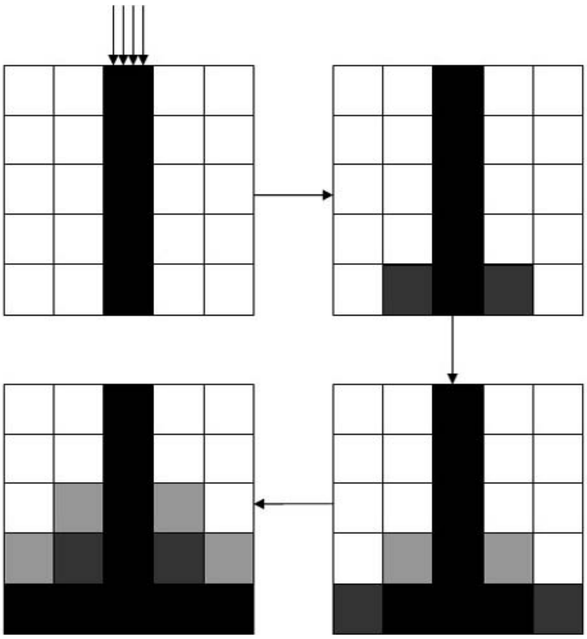


FIGURE 6.3 Fluid flow.

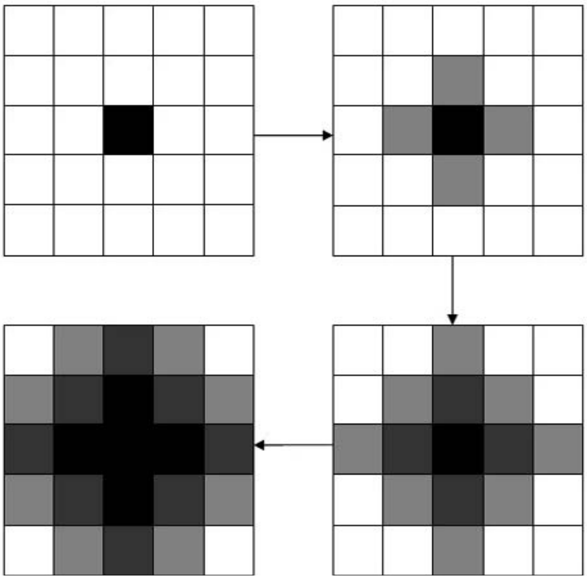


FIGURE 6.4 Burning.

For the Active Game World example, the equations will be developed into complete algorithms and adapted to strategy games, as opposed to human-sized games. Extensive tuning is also required to ensure the behavior of the system meets the requirements. The development and tuning of the Active Game World for a strategy game is discussed in the following sections.

ADDITIONAL READING

For further information on physical modeling with cellular automata in games:

- Forsyth, T. (2002) Cellular Automata for Physical Modeling. *Game Programming Gems 3*. Hingham, MA: Charles River Media, pp. 200–214.

KEY TERMS

- *Conduction* involves neighboring cells passing heat to each other until they reach the same temperature.
- *Convection* models the process of heat rising.
- *Radiation* models the phenomenon that hot objects emit light.
- *Pressure* flows from areas with higher pressure to areas with lower pressure, at a rate corresponding to the difference in pressure, until equilibrium is reached.
- *Fluid flow* works the same as pressure diffusion, except fluid is compressible so that the fluid at greater depth seems to have greater pressure.
- *Fire* is the amount of heat energy that is released per unit of time when a material burns at a certain temperature.

ACTIVE ENVIRONMENT STRUCTURE

The Active Game World model presented in this chapter can be viewed as a hierarchy with three levels (see Figure 6.5). The top level (Level 1) is the behavior of the system that is observable by the players, such as the effects of fire, damage, and fluid flow. The player can see that a cell is on fire or that it is damaged, because these are observable effects.

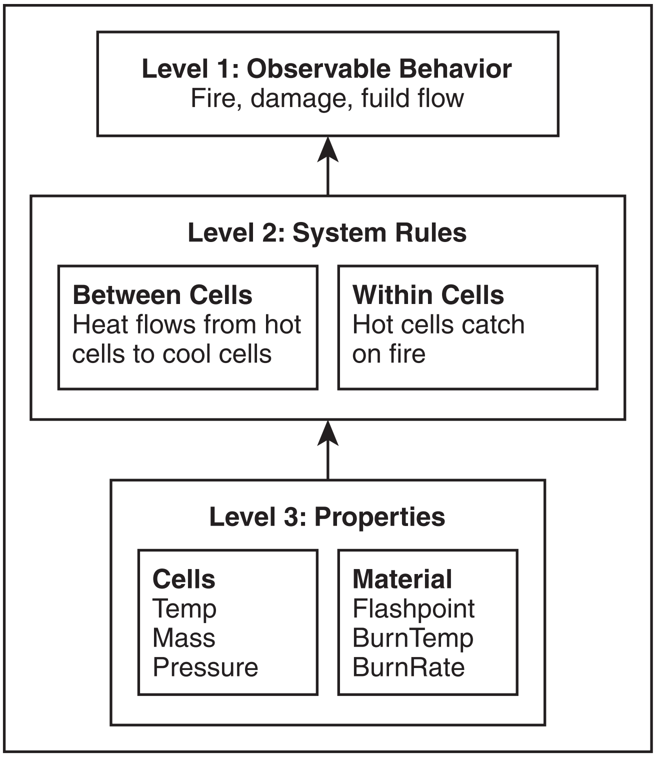


FIGURE 6.5 Active Game World model.

The second level (Level 2) consists of the simple rules that give rise to the visible, complex, top-level behavior. There are two types of rules at the second level of the system. First, there are rules that define the interactions between neighboring cells, such as the spread of heat from one cell to another. An example rule for interactions between cells is that heat flows from a hot cell to a cooler cell. At the second level, there are also rules for interactions within a cell, such as the burning of a cell. An example rule for interactions within a cell is that hot cells catch on fire.

Finally, the third level (Level 3) of the hierarchy contains the properties of the cells, which determine how cells act and react in accordance with the rules at the level above. For example, each cell is made of a material that has a certain flashpoint, burning temperature, and burning rate that determines how hot it needs to be to catch on fire, how fast it burns, and how long it will burn.

The multi-leveled design of the Active Game World model allows the system to be systemic, facilitates emergent top-level behavior, and lends it to future extension.

The implementation of an emergent system is an iterative process. First, the rules need to be implemented and approximate starting values set for parameters (see the "Properties" section). Subsequently, rules and variables must be tuned through testing in various game scenarios (see the "Observable Behavior" section).

Each system in the Active Game World was added incrementally (fluid, heat, and pressure) and first tested individually and then together. During development, it was found that some systems needed significant changes to suit the strategy-game environment and achieve desired behavior (such as fluid flow) and that others could be integrated without alteration (such as heat). The properties, rules for interactions between cells, and rules for interactions within cells are explained in the following sections, including their design, implementation, and considerations.

Properties

Level 3 of the Active Game World hierarchy (see Figure 6.5) contains the properties of the cells. This level contains the data structures for the cellular automaton and materials of the system. The system consists of a grid of cells and each cell has a set of properties. Included in the set of properties is the material (or terrain) of the cell. Each material has its own set of properties that govern its behavior.

The main data structure in the system is the grid for the cellular automaton. The grid consists of 100 cells (10 by 10), each of which represents a piece of a strategy game map of arbitrary size. In commercial strategy games, the size of the cells that are used in structures such as influence maps is arbitrary and there is a tradeoff between accuracy and efficiency. If the cells are too large, the influence map will miss important features and if the cells are too small then there is redundant information and substantial memory is used. Usually, the cells are made fairly large, approximately big enough to fit 10–20 standard units side by side, and from there the cell size is tuned to obtain optimal results.

Each cell in the Active Game World model is a record with a set of 12 associated properties, including the terrain type (material), temperature, mass, damage, wetness, height, fluid, and pressure. The following code implements the data structure for the cellular automaton:

```
// Structure of a cell
struct cell{
       int Material;
       float Temp;
```

```
        float NewTemp;
        float Mass;
        float NewMass;
        float Burn;
        float Damage;
        float NewDamage;
        float Wetness;
        float Height;
        float Fluid;
        float NewFluid;
};
// Active Game World cells
int numX, numY, buffer;
numX = 100;
numY = 100;
buffer = 2;
cell cells[numX+buffer][numY+buffer];
```

There is also a set of materials, which contains information about all the materials in the system. Each material has a set of properties, including flashpoint, burning temperature, maximum burning rate, specific heat capacity (SHC), and maximum fluid level before overflow (see Table 6.1). The values were initially estimated and then tuned until satisfactory behavior was achieved. The following code implements the data structure for the materials:

```
// Structure of a material
struct material{
        float FlashPoint;
        float BurnTemp;
        float MaxBurn;
        float SHC;
        float BurnRate;
        float MaxFluid;
};
// Active Game World materials
int numMaterials;
numMaterials = 3;
material materials[numMaterials];
```

TABLE 6.1 Cell Material Properties

		Values		
Property	**Description**	**Water**	**Grass**	**Woods**
Flashpoint	Ignition temperature of the material	99999	99	2000
BurnTemp	Multiplier for the temperature that the material burns at (amount of heat released)	0	3	5
BurnRate	Multiplier for the rate that the material burns at (rate of consuming fuel)	0	20	10
MaxBurn	Maximum rate that the material can burn at	0	200	300
SHC	Specific heat capacity—the amount of energy required to heat up this material	1	100	100
MaxFluid	Maximum amount of fluid a cell can hold	60	60	60

Rules for Interactions between Cells

Level 2 of the hierarchy contains the rules of the cellular automaton, which define the interactions between cells. An important process for maintaining the cellular automaton is the update procedure. Apart from the update process, there are rules for the environmental systems that use the cellular automaton to spread their effects, including heat, fluid flow, and pressure. The following sections discuss the equations, algorithms, and pseudo-code for heat, pressure, and fluid flow in the Active Game World model.

Updating the Cellular Automaton

In the update process, the new values for fluid, temp, damage, and pressure are substituted for the old values in the cellular automaton. Any rain that is currently falling is added to the fluid value. The following code implements the update procedure:

```
for (int x=0; x < numX+buffer; x++)
{
        for (int y=0; y < numY+buffer; y++)
        {
              cells[x][y].NewFluid += Rain[x][y];
              cells[x][y].Fluid = cells[x][y].NewFluid;
              cells[x][y].Temp = cells[x][y].NewTemp;
              cells[x][y].Damage = cells[x][y].NewDamage;
              cells[x][y].Pressure = cells[x][y].NewPressure;
       }
}
```

Finally, the buffer is cleared. The *buffer* is made up of the outermost layer of cells in each direction and is used to simulate the effects of the system moving out into the world beyond the game world. For example, heat is released into the buffer each turn and at the beginning of each turn the buffer is reset to zero. The following code implements the buffer clearing:

```
if ((x==0) || (x>numX) || (y==0) || (y>numY))
{
       cells[x][y].Temp = 0;
       cells[x][y].Damage = 0;
       cells[x][y].Burn = 0;
       cells[x][y].Fluid = 0;
}
```

Key Terms

- *Level 1* is the behavior of the system that the player can observe.
- *Level 2* consists of the simple rules that give rise to the visible, complex, top-level behavior.
- *Level 3* of the hierarchy contains the properties of the cells, which determine how cells act and react in accordance with the rules at the level above.
- *Properties* include the data structures for the cellular automaton and materials of the system.
- *Buffer* is made up of the outermost layer of cells in each direction and is used to simulate the effects of the system moving out into the world beyond the game world.

Heat

The algorithm for conduction described in Forsyth (2002) was used as a basis for heat diffusion in the Active Game World. In the Active Game World, each material has a specific heat capacity, which determines how much heat is required to raise the temperature of that material. Materials with a high specific heat capacity require more energy to raise their temperature.

In order to diffuse heat in the Active Game World, the heat capacity of the cell, *HCCell*, is first calculated. The heat capacity is equal to the specific heat capacity of the material, *material(cell).SHC*, multiplied by the mass of the material in the cell, *cell.Mass*. The heat capacity for the neighbor, *HCNeigh*, is calculated in the same way.

```
HCCell = material(cell).SHC * cell.Mass
HCNeigh = material(neigh).SHC * neigh.Mass
```

The energy flow, *EnergyFlow*, between the cell and its neighbor is equal to the difference in temperature between the cell, *cell.Temp*, and its neighbor, *neigh.Temp*. The energy flow value is converted from heat to energy by multiplying it by the heat capacity of the transmitting cell. The energy flow is then multiplied by a constant, *ConstantEnergyFlowFactor*, to control the speed of the cell update.

```
EnergyFlow = cell.Temp – neigh.Temp
EnergyFlow *= HCCell
EnergyFlow *= ConstantEnergyFlowFactor
```

Subsequently, the new heat values for the cell, *cell.NewTemp*, and neighbor, *neigh.NewTemp*, are calculated by dividing the energy flow by the heat capacity for each cell.

```
neigh.NewTemp += EnergyFlow / HCNeigh
cell.NewTemp -= EnergyFlow / HCCell
```

To reduce oscillations (heat moving back and forth between the same two cells), the heat of neighboring cells is distributed evenly if the neighbor cell has more heat than the cells as a result of the heat transfer.

```
TotalEnergy = (HCCell * cell.Temp) + (HCNeigh * neigh.Temp)
AverageTemp = TotalEnergy / (HCCell + HCNeigh)
cell.NewTemp = AverageTemp
neigh.NewTemp = AverageTemp
```

The diffusion of heat is increased in the direction that the wind is blowing and reduced against the wind, proportional to the speed of the wind.

Convection and radiation were not modeled in the Active Game World as the scale of the strategy game environment means that these processes are likely to go unnoticed. In a first-person game, where the environment is on a “human-sized” scale, convection and radiation are likely to be more important.

A possible enhancement to the Active Game World would be to model convection to some extent. In a strategy game environment, the most appropriate application of convection would be to allow heat to transfer faster uphill by a small amount. However, such a subtle difference may not be noticed by the player or add to the gameplay in any way.

The following pseudo-code implements the diffusion of heat in the Active Game World:

```
PROCEDURE heat (currentCell, neighborCell)

       // get the heat capacities of the cell and the neighbor
       HCCell = material(currentCell).SHC * currentCell.Mass;
       HCNeigh = material(neighborCell).SHC * neighborCell.Mass;

       // calculate the difference in temp between the cell and neighbor
       EnergyFlow = currentCell.Temp – neighborCell.Temp;

       // convert from heat to energy
       EnergyFlow *= HCCell;

       // multiply by a constant for cell update speed
       EnergyFlow *= ConstantEnergyFlowFactor;

       // heat doesn’t flow directly against wind
       IF (neighborCell isn’t directly against wind) THEN
              // cell has higher heat than neigh
              IF (EnergyFlow > 0) THEN
                     // heat flow into neighbor
                     neighborCell.Temp += EnergyFlow / HCNeigh;
                     // heat flows from cell
                     currentCell.Temp -= EnergyFlow / HCCell;
              END

              // detect and kill oscillations
              IF ((EnergyFlow > 0) AND (neighborCell.Temp <
              currentCell.Temp))
              THEN
                     // find average temp of cell and neigh
                     TotalEnergy = (HCCell * currentCell.Temp)
                            + (HCNeigh * neighborCell.Temp);
                     AverageTemp = TotalEnergy / (HCCell + HCNeigh);
```

```
                              // set cell and neigh to average temp
                              currentCell.Temp = AverageTemp;
                              neighborCell.Temp = AverageTemp;

                              // increase heat flow directly with wind
                              IF (neighborCell is directly with wind) THEN
                                     currentCell.Temp /= (1 + (windspeed *
                                       wind_const));
                                     neighborCell.Temp *= (1 + (windspeed *
                                       wind_const));
                              END
                       END
                END
         END PROCEDURE
```

Fluid Flow

The method suggested for fluid flow by Forsyth (2002) is not well suited to a strategy game environment, because it is designed to simulate the effects necessary in a "human-sized" game, such as a first-person shooter. For example, it is more suited to water flowing into a container, such as a bucket, and would not be viable for simulating large bodies of water, such as a river in a strategy game.

Due to the difference in the scale of a human-sized game and a strategy game, a new approach was needed for the Active Game World. A new algorithm was developed that was similar to the heat and pressure diffusion algorithms, with the addition of terrain height. For heat and pressure, the terrain can be treated as though it is flat. However, for fluid flow, it is necessary to know the contours of the landscape, because fluid has different rules for flowing downhill, uphill, and on level ground.

The main principle modeled in the Active Game World's fluid-flow algorithm is the flowing of the fluid from one cell to its neighboring cells. Once the fluid in a cell, *cell.Fluid*, exceeds a certain amount (dependent on the maximum amount of fluid the material in the cell can hold, *material(cell).MaxFluid*), the fluid then flows into the surrounding cells. The flow of fluid between cells is affected by the relative height of the cell, *cell.Height*, and its neighbors, *neigh.Height*.

```
IF (currentCell.Fluid > (material(cell).MaxFluid
       * (neighborCell.Height / currentCell.Height)))
```

Fluid flows faster to lower cells and less fluid is accumulated in a cell before it starts flowing to a lower neighbor. When flowing uphill, more fluid is accumulated in the cell, until the cell overflows to its higher neighbors. Fluid flows at a slower rate uphill. The steeper the slope, the more effect it has on the flow of the fluid, in terms of the rate of the *flow* and the amount of fluid that is accumulated before flowing to the neighbors.

```
flow = flow * (currentCell.Height / neighborCell.Height)
```

Fluid flows faster when there is a greater difference between the amount of fluid in a cell, *cell.Fluid*, and the amount in the neighboring cells, *neigh.Fluid*, because there would be greater pressure. The difference is divided by four as each cell has a maximum of four neighboring cells, providing a good approximation to how the fluid will be divided into the neighbors.

```
flow = (currentCell.Fluid – neighborCell.Fluid) * 0.25
```

Another property of fluid is that it wets. Material that is wet has a lower temperature, as well as being harder to ignite and slower to burn. The wetness in a cell also reduces slowly overtime, as the moisture evaporates, and gradually repairs damage over time.

Currently, there is only one type of fluid that is modeled in the Active Game World. However, a possible enhancement to the Active Game World would be to model different types of fluids. Different fluids flow at different rates, depending on properties such as their viscosity. However, the need to model fluids other than water would be more relevant to human-sized games, rather than strategy games. In a strategy game, it is unlikely that there will be large bodies of fluid other than water, but a possible example would be lava or fuel, which would flow very differently to water.

The following pseudo-code implements fluid flow in the Active Game World:

```
PROCEDURE fluid (currentCell, neighborCell)

    // neighborCell lower than currentCell
    IF ((neighborCell.Height < currentCell.Height)

          // currentCell has the max fluid it will hold,
          // modified by the slope – easier to flow downhill
          AND (currentCell.Fluid > (material(currentCell).MaxFluid
                 * (neighborCell.Height / currentCell.Height)))) THEN

          // flow equals the difference between fluid in currentCell
          // and neighborCell divided by four
          flow = (currentCell.Fluid – neighborCell.Fluid) * 0.25;

          // flow is increased proportionally to slope
          flow = flow * (currentCell.Height / neighborCell.Height)
                 * flow_const;
```

```
        // flow cannot be less than zero
        IF (flow < 0) THEN flow = 0;

        // update currentCell and neighborCell with flow
        currentCell.Fluid -= flow;
        neighborCell.Fluid += flow;

        // currentCell.Fluid cannot be less than zero
        IF (currentCell.Fluid < 0) THEN currentCell.Fluid = 0;
END

// neighborCell higher than currentCell
ELSE IF ((neighborCell.Height > currentCell.Height)

        // currentCell has the max fluid it will hold,
        // modified by slope — harder to flow uphill
        AND (currentCell.Fluid > (material(currentCell).MaxFluid
               * (neighborCell.Height / currentCell.Height)))) THEN

        // flow equals difference between the fluid in currentCell
        // and neighborCell divided by four
        flow = (currentCell.Fluid — neighborCell.Fluid) * 0.25;

        // flow is decreased proportionally to slop
        flow = flow * (currentCell.Height / neighborCell.Height)
        / flow_up_const;

        // flow cannot be less than zero
        IF (flow < 0) THEN flow = 0;

        // update currentCell and neighborCell with flow
        currentCell.Fluid -= flow;
        neighborCell.Fluid += flow;

        // currentCell fluid cannot be less than zero
        IF (currentCell.Fluid < 0) THEN currentCell.Fluid = 0;
END

// neighborCell on same level
// fluid in currentCell must exceed max fluid cell can hold
ELSE IF (currentCell.Fluid > material(currentCell).MaxFluid) THEN
```

```
            // flow equals difference between currentCell and neighborCell
            // divided by four
            flow = (currentCell.Fluid – neighborCell.Fluid) * 0.25;

            // flow cannot be less than zero
            IF (flow < 0) THEN flow = 0;

            // update currentCell and neighborCell with flow
            currentCell.Fluid -= flow;
            neighborCell.Fluid += flow;

            // cell fluid cannot be less than zero
            IF (currentCell.Fluid < 0) THEN currentCell.Fluid = 0;
        END
    END PROCEDURE
```

Pressure

The Active Game World extends the simple pressure diffusion equations presented by Forsyth (2002) to include a model for explosions. In the Active Game World, the rate of diffusion of pressure, *PressureFlow*, is determined by the difference in pressure between a cell, *cell.Pressure*, and its neighbor, *neigh.Pressure*. As with fluid, the pressure flow is divided by four, because each cell has a maximum of four neighbor cells, providing a good approximation.

```
    PressureFlow = currentCell.Pressure – neighborCell.Pressure
    neighborCell.Pressure += PressureFlow * 0.25
    currentCell.Pressure -= PressureFlow * 0.25
```

If there is a sufficient difference in pressure between two adjacent cells then an explosion occurs, due to the rapid change in pressure. An explosion occurs when a material produces a large amount of air in a short period of time.

```
    pressure_ratio = currentCell.Pressure / neighborCell.Pressure
```

When an explosion occurs in the Active Game World, heat is released in an amount proportional to the ratio of pressure between cells (the size of the explosion).

```
    currentCell.NewTemp += (explosion_const * pressure_ratio) * 0.25
```

If there is enough heat, a fire is started (see the "Fire" section) and damage is caused to the surrounding cells. Additionally, explosions cause damage due to high absolute pressures, as well as high pressure differences. Therefore, the Active Game

World models the effects of high absolute pressure in cells. A cell with a high enough pressure causes damage to itself and its contents, irrespective of the pressures of the surroundings cells.

A possible enhancement to the Active Game World would be to have the pressure system affecting the wind, so they are part of the same force rather than different forces acting independently. This integration would be particularly useful when objects are introduced into the system. When an explosion occurs, small objects and debris should be picked up and carried by the force of the explosion.

The following pseudo-code implements pressure diffusion in the Active Game World:

```
PROCEDURE pressure (currentCell, neighborCell)

       // if there is more pressure in cell than in neighbor
       IF (currentCell.Pressure > neighborCell.Pressure) THEN
              // pressure ratio between currentCell and neighborCell
              pressure_ratio = currentCell.Pressure / neighborCell.Pressure;

              // if pressure ratio is more than explosion ratio then explode
              IF (pressure_ratio > explosion_ratio) THEN
                           // release heat proportional to pressure ratio
                           currentCell.Temp += (explosion_const *
                             pressure_ratio) * 0.25;
              END

              // pressure difference between currentCell and neighborCell
              PressureFlow = currentCell.Pressure – neighborCell.Pressure;

              // pressure diffuses to neighborCell
              neighborCell.Pressure += PressureFlow * 0.25;
              currentCell.Pressure -= PressureFlow * 0.25;

              // detect and remove oscillations
              IF ((PressureFlow > 0) AND
                     (neighborCell.Pressure < currentCell.Pressure)) THEN

                     // calculate the average pressure of currentCell
                     // and neighborCell and distribute evenly
                     TotalPressure = currentCell.Pressure
                            + neighborCell.Pressure;
                     AveragePressure = TotalPressure / 2;
                     currentCell.Pressure = AveragePressure;
                     neighborCell.Pressure = AveragePressure;
```

```
                END
         END
END PROCEDURE
```

Rules for Interactions within Cells

Level 2 of the Active Game World hierarchy (see Figure 6.5) also includes rules for interactions that occur within a cell. Similar to the rules for interactions between cells, the rules for interactions within cells interact with Level 3 of the hierarchy—the properties. However, the rules for interactions within cells are specific to what is happening within an individual cell, irrespective of what is happening in neighboring cells, and include rules for fire, wind, and rain. The following sections discuss the equations, algorithms, and pseudo-code for fire, wind, and rain in the Active Game World.

Fire

The simulation of fire in the Active Game World is based on the equations provided by Forsyth (2002). In the Active Game World, if the temperature of a cell, *cell.Temp*, exceeds the flashpoint of the material in the cell, *material(cell).Flashpoint*, the cell ignites. The rate that the cell burns at depends on the burning rate of the material in the cell, *material(cell).MaxBurn*, and the temperature of the cell. The wetness of the cell, *cell.Wetness*, affects the rate that it burns and how difficult it is to ignite.

```
Temp = cell.Temp – (material(cell).Flashpoint + cell.Wetness)
Burn = (1.0 –((0.25 * Temp)/material(cell).MaxBurn)) * Temp
```

As a cell burns, damage is caused to the cell, *cell.NewDamage*, proportional to the temperature of the cell. As a cell becomes more damaged, it burns slower until all the fuel in the cell is used up and it can burn no longer.

```
cell.Damage += (Temp * material(cell).BurnRate) – cell.Wetness) *
  burn_const
Burn -= cell.Damage
```

As the cell burns, the fire releases heat and the cell heats up proportional to the burning rate of the cell and burning temperature of the material, *material(cell). BurnTemp*.

```
cell.NewTemp += Burn * material(cell).BurnTemp
```

Different materials burn at different rates and have different flashpoints. Three materials are modeled in the Active Game World: water, grass, and wood. Water cannot burn, grass is easy to ignite and burns quickly, and wood is harder to ignite and burns longer as it provides more fuel per cell.

The following pseudo-code implements fire in the Active Game World:

```
PROCEDURE fire (cell)

       // temperature is the difference between the temp of the cell and
       // the flashpoint of the material in the cell
       // and the wetness of the cell
       tempDifference = cell.Temp – (material(cell).FlashPoint +
         cell.Wetness);

       // damage the cell
       IF (tempDifference > 0) THEN
           cell.Damage = ((tempDifference * material(cell).BurnRate)
                  - cell.Wetness) * burn_const;
       END

       // convert to actual burning value
       IF (tempDifference > (material(cell).MaxBurn * 2)) THEN
              Burn = material(cell).MaxBurn;

       ELSE IF (tempDifference> 0) THEN
               Burn = (1.0 – ((0.25 * tempDifference) /
                 material(cell).MaxBurn)) * tempDifference;
       END

       // burn cannot exceed MaxBurn
       IF (Burn > material(cell).MaxBurn) THEN
              Burn = material(cell).MaxBurn;
       END

       // reduce burn by amount of damage in cell (less fuel to burn
       // when damaged)
       Burn -= cell.Damage;

       // burn cannot be less than zero
       IF (Burn < 1) THEN Burn = 0;

       // Heat the cell up from the burning
       cell.Temp += Burn * material(cell).BurnTemp;
       cell.Burn = Burn;

END PROCEDURE
```

Wind

In the Active Game World, wind is a global value that is comprised of two components, speed and direction. Wind blows from one of four directions, north, south, east, or west. The speed of the wind is set to an arbitrary strength of ten. The only effect of the wind in the Active Game World is that it modifies the spread of heat. Heat spreads slower against the wind and spreads faster in the direction that the wind is blowing, depending on the speed of the wind, *windspeed*.

```
IF (neighborCell is with wind) THEN
       currentCell.NewTemp /= 1 + (windspeed * wind_const)
       neighborCell.NewTemp *= 1 + (windspeed * wind_const)
END

IF (neighbor is against wind) THEN
       currentCell.NewTemp *= 1 + (windspeed * wind_const)
       neighborCell.NewTemp /= 1 + (windspeed * wind_const)
END
```

Wind currently affects the entire grid in the same way (there is a uniform wind speed and direction for the entire grid). A possible enhancement to the Active Game World would be to have local wind effects, as opposed to the current global wind effects.

Rain

In the Active Game World, rain affects fluid flow by adding fluid to the cells where it is raining. In turn, cells that contain fluid are wet by the fluid, increasing their wetness value. The effects of wetness are described in the section on fluid flow. In the update cycle, the amount of rain in a cell is added to the fluid in the cell.

Key Terms

- *Rules for interactions between cells* are rules for the environmental systems that use the cellular automaton to spread their effects, including heat, fluid flow, and pressure.
- *Rules for interactions within cells* are specific to what is happening within an individual cell, irrespective of what is happening in neighboring cells, and include rules for fire, wind, and rain.
- *Wind* is a global value that is comprised of two components, speed and direction.
- *Rain* affects fluid flow by adding fluid to the cells where it is raining.

Prototypes of the Active Game World

Originally, the Active Game World was implemented in DirectX and visualized in two dimensions (2D). The 2D prototype included three types of terrain: grass, water and woods, which were represented as light green, blue, and dark green, respectively (see Figure 6.6). The observable effects in the system were also visualized, including fire, fluid flow, rain, and damage caused by heat.

Each of the effects was visualized as different colored pixels, where fire was red, fluid was blue, rain was light blue, and damage was black. The number of pixels for each effect in each cell was equal to the magnitude of the effect. For example, as a cell became more damaged there were more black pixels to illustrate the damage. The code and demo for the 2D prototype can be found on the accompanying CD-ROM.

ON THE CD

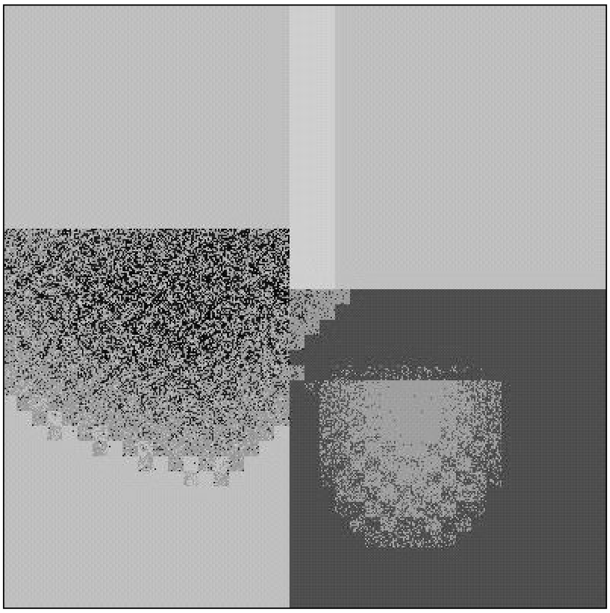


FIGURE 6.6 Active Game World prototype in 2D.

Subsequently, the Active Game World was ported into a three-dimensional (3D) game engine, the Auran Jet (www.auran.com/jet/overview.htm), to provide a realistic game-like environment. Similar to the 2D representation, the 3D representation included the same three terrain types, water, woods, and grass. The observable effects are modeled in the 3D system by the use of sprites (2D objects), including a rain cloud, water, fire, and damage (see Figure 6.7). The 3D representation also visualizes contour of the terrain.

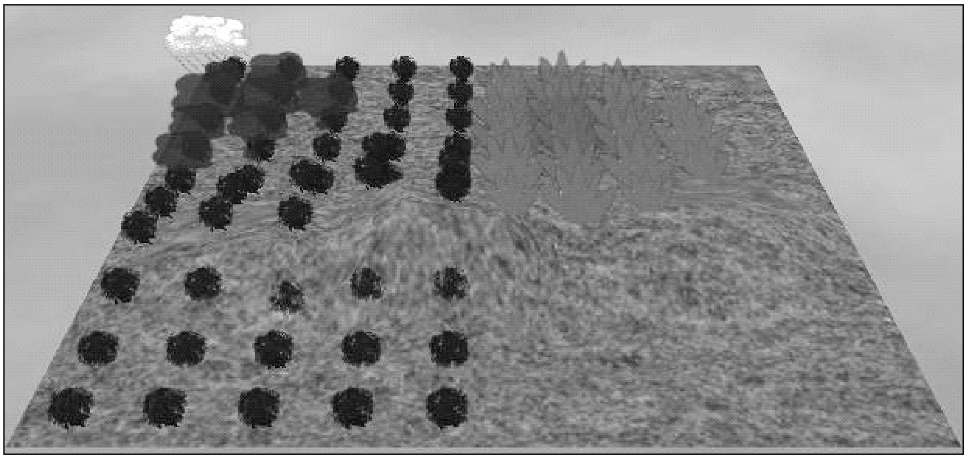


FIGURE 6.7 3D Active Game World.

Observable Behavior

In games, scenarios and sequences of actions are often specifically scripted and coded. The advantage to using cellular automata and a systemic approach, as used in the Active Game World, is that the observable behavior of any object or situation in the game can be dynamic and emergent, without needing to be scripted individually for each object or situation. As a result, the game engine is flexible and responds consistently and realistically to a wide range of actions that players may take and events in any situation in the game.

This section examines four scenarios that are possible situations for a strategy game and evaluates the performance of the Active Game World and a conventional scripted system in terms of the observable behavior. In order to tune the behavior of an emergent game system, a series of scenarios such as these need to be run, and the parameters and rules tuned until the game behaves as expected. The four scenarios used for evaluation and tuning are as follows:

- Heat and fire
- Fluid and wetness
- Pressure and explosions
- Integrated system with each of the previous components

ON THE CD

Demos for each of these scenarios in the 3D Active Game World can be found on the CD-ROM.

Scenario 1: Heat and Fire

Consider an example where a fire starts in a forest or woods. To appear natural, the fire needs to burn, cause damage to the woods, and spread through the woods, all at a believable rate. Subsequently, the fire is blown north to grasslands by wind.

As grass has different properties than woods, the fire should behave differently when interacting with the grass, it will burn out faster as the grass provides less fuel, it will spread faster as the grass is easier to ignite and it will release less heat as it burns. Finally, the fire runs into a river that runs across the map and the fire spreads no farther since the river cannot be ignited, although some heat can be passed across it (see Figure 6.8).

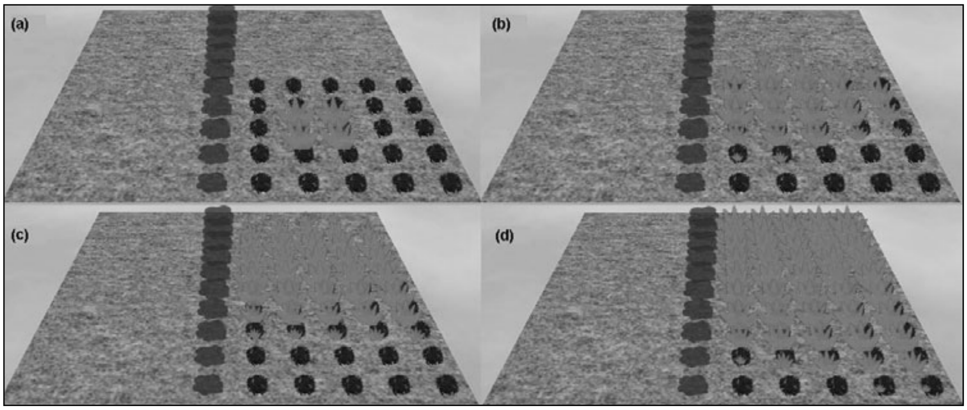


FIGURE 6.8 A fire starts in a forest (a), wind spreads the fire north to grasslands (b–c), and the fire is blocked by a river (d).

Scripted System Evaluation

In a game where the heat and fire scenario is scripted, the rate at which the fire burns, spreads, and damages would need to be specifically coded. Also, the rate of the fire spreading from the woods to the grass would need to be specified, as would the information that the fire must stop burning when it reaches the river. For example, the script would include instructions such as burn each cell of woods for x seconds, spread fire to neighboring woods cells after y seconds, spread fire from woods to grass after z seconds and so on.

If the scenario were changed slightly, such as increasing the density of the woods, the variables that are dependent on the density of the woods, such as the rate at which the fire spreads in woods, would need to be changed. It becomes apparent that a system with a specifically coded, static architecture is not robust to even small changes in the heat and fire scenario. Each time a change is made, each variable that is associated with the change would need to be updated by hand, or the system would not work.

Furthermore, extending this scripted system would become difficult and awkward. For example, if a new type of terrain were added, such as scrub, there would need to be new variables for fire moving from woods to scrub, grass to scrub, scrub to woods, and scrub to grass, as well as all the variables for burning scrub. It is clear that the number of instructions needed to run the heat and fire scenario would rapidly grow out of control with even a few small extensions.

Active Game World Evaluation

In contrast to scripted systems, when the heat and fire scenario is run in the Active Game World, the resulting behavior depends on the underlying properties of the terrain, which are used by the equations for heat and fire to determine the behavior of the system dynamically.

The approach that is taken when scripting is to attempt to encode the behavior directly. For example, the woods burn for five seconds because it is told to burn for five seconds. However, the Active Game World uses lower-level information so that the visible behavior of the system will be emergent. With the Active Game World, the woods burn for five seconds because the system rules consider its flashpoint, specific heat capacity, rate of burning, amount of fuel, temperature, and so on, and calculates at each time step whether it is still burning. Due to the extra level of complexity, the Active Game World works on any material in any scenario.

It may seem more complex to make these calculations, but there is one set of formulae that applies in any situation with any material and the only difference is the properties of each material. Therefore, once the set of formulae is coded, the burning of any material can be simulated as long as the values of the properties are available, making it a data-driven system. There is no awkward expansion of the system, no limit to the number of materials that can be added, and a great deal of flexibility in handling new situations, which reduces the difficulty and cost of quality assurance.

Scenario 2: Fluid and Wetness

The second scenario presents a case study with fluid and wetness, in which rain falls on the map in a position that is at the top of a hill. A certain amount of the water is able to soak into the ground at the top of the hill (depending on the terrain), but after some rain has fallen the water starts to run down the hill. The harder the rain, the faster the water runs down the hill. Also, the steeper the slope, the faster the water runs down the hill.

At the bottom of the hill is a valley, which rises back up into a hill on the other side. When the water reaches the valley it starts to build up and runs outward to fill the level ground in the valley. With sufficient rainfall, the water will fill the valley and start to flow back up the hills on either side of the valley (see Figure 6.9).

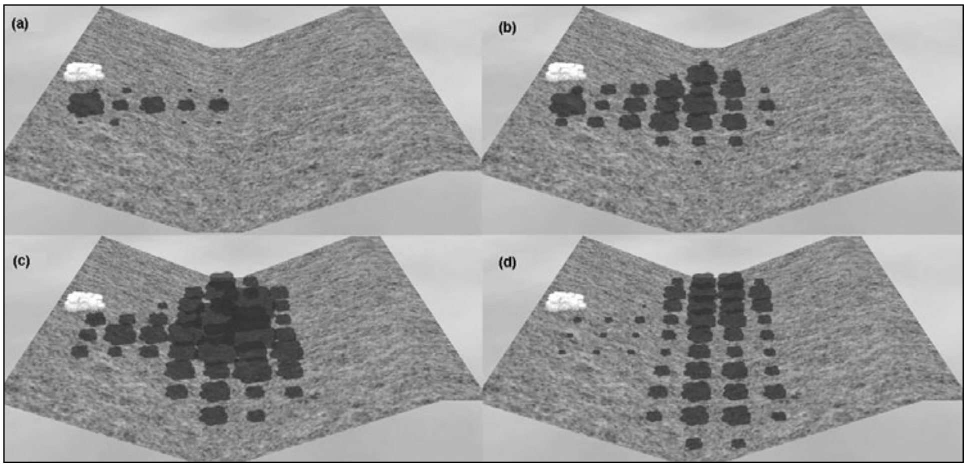


FIGURE 6.9 Rain falls at the top of a hill (a) and the water runs down the hill (b) and spreads out in the valley below (c–d).

Scripted System Evaluation

When scripting the fluid and wetness scenario specifically, the script would need to encode information for where the water will run, how fast it will run and other behaviors such as how deep it will pool in different areas. Scripting the behavior of the water specifically would be relatively simple, but would rely on the rain falling in the same position and at the same rate and the contours of the landscape always being the same.

If the rain were to fall from a different position, or if the contours of the landscape were to change even slightly, the behavior of the water would be completely different and a new script would be needed. So, if the fluid and wetness scenario was an event in a game that always played out the same way, a script would be fine. However, if rain was a random event that occurred in the game, it could not be foreseen where it would fall or what path it should follow.

Active Game World Evaluation

In the Active Game World, the position of the rain, the speed of the rain and the contour of the landscape are not a problem. The Active Game World makes calculations every time step for which way the water will flow, how fast it will flow and how deep it will pool. The Active Game World uses its formulas and the data it is provided, such as the relative height of the neighbors of each cell and the amount of water in each cell, and it dynamically calculates the behavior of the water at each time step.

As a result, the Active Game World can accommodate changes in the environment and simulate the flow of water down the hill for different positions and speed of the rain and for different contours of the hill. The Active Game World uses its rules and properties to determine the behavior of the water in real-time, dependent on the current situation, rather than having the behaviors of the water pre-scripted.

Scenario 3: Pressure and Explosions

The third scenario presents a case study with pressure and explosions, in which an explosion occurs somewhere on the map. The high absolute pressure of the explosion causes immediate damage in the vicinity of the explosion. The area that is affected by the pressure depends on the magnitude of the explosion.

In addition to the damage from pressure, energy is generated as a result of the difference in pressure between the explosion and the surrounding area. As a result of the energy (heat) that is released, a fire is started in the area around the explosion. The fire then spreads and causes damage to a wider area around the initial explosion (see Figure 6.10).

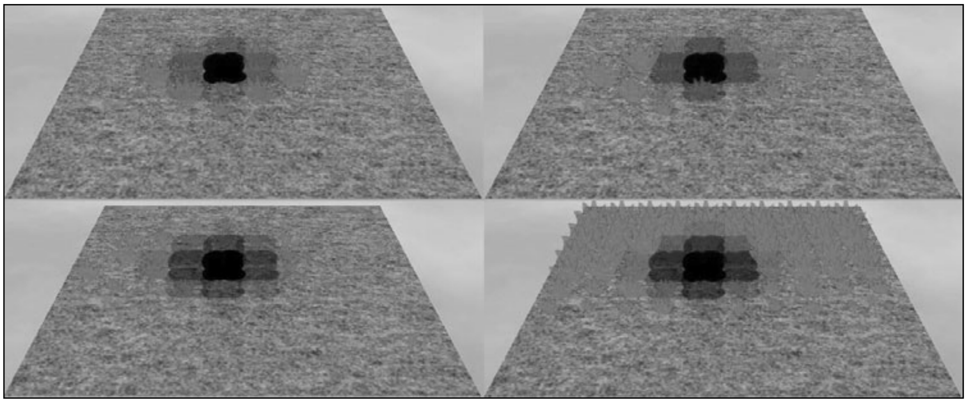


FIGURE 6.10 An explosion occurs, the high absolute pressure causes immediate damage and a fire starts as a result of the heat released by the explosion. Each panel shows different-sized explosions.

Scripted System Evaluation

An explosion is relatively simple to simulate in the conventional manner, and is commonly scripted in many commercial games. However, there is a difference between a scripted explosion occurring that is triggered by a bomb going off and an explosion naturally occurring as there is suddenly a large difference in pressure between two cells.

With the scripted example, there must always be a high-level trigger for the explosion, such as a bomb, a grenade, or some other discrete explosion event. On the other hand, if there are conditions under which an explosion naturally occurs, such as a large difference in pressure brought about by piercing a gas cylinder, the explosion will naturally occur when these conditions are met. Therefore, there does not need to be a discrete event, rather the explosion will occur because an explosive device has just caused a large difference in pressure or for unforeseen reasons.

Active Game World Evaluation

In the Active Game World, an interesting occurrence is the second degree effect of the explosion, namely the fire that is started as a result of the rapid increase in heat around the explosion. This second degree effect could be manufactured in a scripted explosion, but would not reach the same level of realism as the Active Game World, which can dynamically take into consideration the type of terrain, objects in the area, and other important factors.

Depending on the magnitude of the explosion, there would be different amounts of heat generated and different effects of the fire. Although this variation could be scripted, it would not have the flexibility of the Active Game World. For example, if an explosion occurs in an area containing highly flammable material, the behavior of the Active Game World would be significantly different, but a scripted explosion would consider only the magnitude of the explosion and burn an area around the explosion accordingly.

Scenario 4: Integrated System: Heat, Fluid, and Pressure

The fourth scenario is a combination of the previous three case studies and illustrates the real power of the Active Game World. An explosion occurs in the woods and the high pressure causes damage to the immediate area. As a result of the energy released from the explosion a fire is started, which then spreads through the woods. A wind then blows the fire west into a neighboring grassy valley, which contains a small village.

It has been raining at the top of the hill next to the valley and the water flows down the hill into the valley and puts out the fire. The fire recedes into the woods and burns until the woods are burnt out and there is no more fuel (see Figure 6.11).

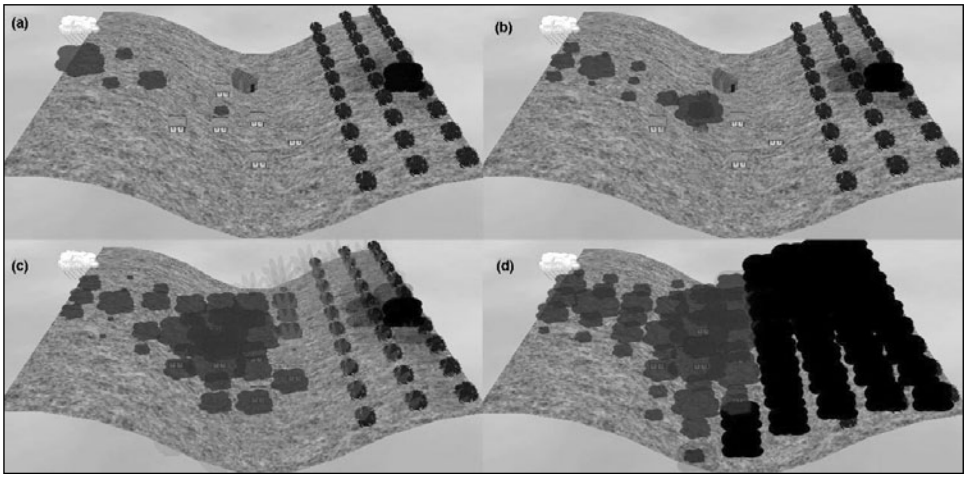


FIGURE 6.11 An explosion in the woods causes immediate damage and starts a fire (a). The fire is blown west by the wind (b) and extinguished by the water flowing down the hill from the rain falling above (c–d).

Scripted System Evaluation

It would take a great deal of effort to specifically script the complex behavior of the environment in the integrated scenario, as well as careful attention to detail. Every small change to the scenario would result in different desired behavior, requiring the scenario to be rescripted. Therefore, specifically scripting a scenario as complex as the fourth scenario would require substantial initial effort as well as significant ongoing effort to update and maintain the system.

Active Game World Evaluation

Each element in the Active Game World looks after its own behavior, the heat spreads, the fire burns, the water flows and so on, dependent on its own set of rules and the properties and state of the materials it is acting on. As these elements simultaneously work independently, they impact on each other and give rise to emergent, complex behavior that was not specifically programmed into them. The observable complex behavior in the scenario arises from the independent behavior of the elements, making the whole more than the sum of its parts.

The use of cellular automata means that the complex behavior arises as a result of the individual elements interacting with each other. The Active Game World accommodates the possible variations in a complex scenario, such as different types of terrain, the effect of wind, the contours of the terrain, the interactions of heat, pressure, and fluid, without the need to recode for each specific scenario.

The complex situation described in the integrated scenario demonstrates the benefit of the emergence of the Active Game World. Rather than the complex behavior having to be specifically scripted, it just happens as a result of all the components of the system working together.

Active Environment Summary

In the first part of this chapter, you developed and tested a set of properties and rules for a cellular automaton that can be used to model basic environmental effects in a strategy game environment. To create the environment for the Active Game World, you started with simplified equations from thermodynamics and developed three complete, independent systems: heat, fluid flow, and pressure, with local effects of rain, explosions, and fire.

Each of the major systems, heat, fluid, and pressure, demonstrated various advantages over conventional scripted approaches. For heat and fire, the major advantage was that any number of terrain types could be added to the system and no additional scripting or calculating would be required. In contrast, in a conventional system, the script would need to be updated for each new material and transition between materials.

In the rain and fluid flow system, the advantage was found in the contours of the terrain. As the Active Game World used the underlying rules and properties to calculate the speed and direction of the flow, any possible terrain contours could be accommodated, whereas a scripted system would need to have the contours of the terrain specified in advance.

For the pressure and explosions system, advantages were that the explosions could occur naturally due to a difference in pressure between two cells, rather than needing an event to trigger the explosion. Also, a fire was started as a second degree effect of the explosion, if sufficient heat was generated by the explosion to ignite the material.

Each of these sub-systems had advantages over conventional scripting methods, which were mostly related to the Active Game World being able to dynamically accommodate changes or variations to the environment that a scripted system would be unable to do. However, the real power of the Active Game World became apparent when the heat, pressure, and fluid flow systems were combined.

In combining these systems, the Active Game World was able to demonstrate the emergent interactions that take place in a complex scenario that would otherwise need to be scripted specifically, such as water flowing downhill to put out a fire that had been blown there from an explosion to the east. Additionally, due to the real-time simulation of the environmental rules, variables in emergent game worlds can be tuned on-the-fly while the game is running. Tuning conventional approaches in real-time is substantially more difficult and can even be impossible in some cases.

Property-Based Objects

Game objects are an integral part of any game world as they compose the major source of player interactions. Objects in games are numerous and varied, including weapons (such as guns and swords) in first-person shooter games, quest items (such as the Holy Grail or a diary) in role-playing games, and buildings (such as barracks and factories) in strategy games. Each type of game object interacts with the game environment and the players in different ways, which gives rise to interesting possibilities for action for the players, but complicates the job of the game developer.

Some games have allowed more freedom and variation through property-based objects and rules for how the objects interact. For example, in the simulation game *The Sims*, intelligence is embedded into objects in the environment, called *Smart Terrain*. The objects broadcast properties to nearby agents to guide their behavior. Similarly, the game objects in the first-person shooter game *Half-Life 2* use named links between pieces of content called *symbolic links* (Walker, 2004) that define the properties of the objects and determine how they can be affected by players and other objects.

Using this global design, the objects behave more realistically and are more interactive as they are encoded with types of behavior and rules for interacting, rather than specific interactions in specific situations. These objects afford emergent behavior and player interactions that were not necessarily foreseen by the developers.

At the basic level, objects are the same as cells in the environment, in that they both exist in the physical world and are therefore subject to the same rules of physics, such as heat transfer, fluid flow, and pressure. However, whereas all cells are uniform in structure, in that they are all sections of terrain, objects have comparatively complex physical structures. This is where the tags or labels used to create property-based game objects, such as in *The Sims* and *Half-Life 2*, come in handy.

This section describes property-based game objects that can be integrated into the Active Game World model. Objects are implemented as though they are cells, using the same low-level properties based on the object's material. Objects are also imbued with high-level properties, based on their structure, to constrain the possible physical interactions of the objects.

Objects exist within cells of the environment and can therefore be treated as additional neighboring cells for the purposes of interacting with the environment. Additionally, the high-level property tags that are attached to objects can be used to create affordances for interactions with the player and other objects.

Object Structure

In the Active Game World model, the objects are designed to have a similar structure as the cells of the environment. Each object has a set of properties that include coordinates (position in the cellular automaton), temperature, pressure, fluid, mass, wetness, and material. Objects are defined in the Active Game World as follows:

```
// Structure of an object
struct obj{
       int Material;
       float Temp;
       float NewTemp;
       float Mass;
       float NewMass;
       float Burn;
       float Damage;
       float NewDamage;
       float Wetness;
       float Height;
       float Fluid;
       float NewFluid;
};

// Objects in the Active Game World
int numObj;
numObj = 10;
obj objects[numObj];
```

The materials that objects are composed of have the same properties as materials for the terrain, including flashpoint, burning temperature, specific heat capacity, and maximum burning rate. An additional property, *strength*, was needed for objects to model the amount of pressure a particular material can withstand before exploding, which was unnecessary for terrain. Object materials are defined as follows:

```
// Structure of a material
struct material{
       float FlashPoint;
       float BurnTemp;
       float MaxBurn;
       float SHC;
       float BurnRate;
```

```
        float MaxFluid;
        float Strength;
};

// Materials in the Active Game World
int numMaterials;
numMaterials = 3;
material materials[numMaterials];
```

In strategy games, the majority of objects in the environment are buildings, such as houses, bunkers, or factories. Therefore, the materials that have been chosen for the Active Game World are common types of building materials, such as wood, metal, and brick. These materials have significantly varying properties and are therefore ideal for illustrating the emergent behavior of the system when interacting with different objects (see Table 6.2). The values for each property for each material in the Active Game World was initially estimated and subsequently tuned until satisfactory behavior was achieved.

TABLE 6.2 Object Material Properties

		Values		
Property	**Description**	**Wood**	**Metal**	**Brick**
Flashpoint	Ignition temperature of the material	2000	5000	4000
BurnTemp	Multiplier for the temperature the material burns at (amount of heat released)	5	10	8
BurnRate	Multiplier for the rate the material burns at (rate of consuming fuel)	10	5	7
MaxBurn	Maximum rate the material can burn at	300	100	200
SHC	Specific heat capacity—the amount of energy required to heat up this material	100	50	150
MaxFluid	Maximum amount of fluid the material can absorb	50	0	20
Strength	Modifier for the pressure the material can withstand before it breaks	0.6	1.0	0.8

First, metal is hard to ignite, transfers heat easily, and absorbs little fluid. Second, wood is much easier to ignite, but does not transfer heat as well and is far more absorbent. Third, brick is moderate to ignite, poor for heat transfer, and can absorb a moderate amount of water.

In addition to these low-level properties, objects have a set of high-level properties (represented as Boolean values) that encode attributes of the object's physical structure, including whether the object has volume, is solid, or is open.

Object Design

At the basic level, objects are the same as cells in the environment in that they both exist in the physical world and are therefore subject to the same rules of physics, such as heat transfer, fluid flow, and pressure. However, whereas all cells are uniform in structure, in that they are all squares of terrain, objects have comparatively complex physical structures.

In order to model the structure of objects at the same level as the rest of the system, it would be necessary to divide the objects up into cells in the same way the terrain is divided into cells. Unit mapping objects would allow the cellular automaton to dynamically determine the contours (that is, the structure) of the object in the same way as the contours of the terrain are dynamically determined. However, there are two main considerations with breaking objects down into cells.

First, given the number of objects in a game environment and the complexity of game objects, it would be prohibitively expensive to perform the necessary calculations. Second, the relative benefit to the game would be minimal, especially in comparison to the cost involved. For example, whereas it is of great benefit to a game to be able to map how water would flow from one cell of the map to the next, it is much less important to calculate exactly how water will flow into an object, such as a bucket. However, knowing that water will flow into the bucket has significant potential in terms of emergent gameplay, in that a player could fill up a bucket with water, carry it to a fire, and extinguish the fire.

The potential benefit of structural object properties to gameplay lies in knowing the important features of the game objects that will have an impact on the possible actions and interactions of the objects. A possible solution is to give the objects descriptors of the important features of their physical structure that predispose them to certain actions and behaviors. For example, only an object that has volume can be filled with water, so that a bucket can be filled with water but a sword cannot.

A logical and computationally viable solution is to have two different levels of properties for objects. As with cells, objects have low-level properties, which define how the matter of the objects interacts in the world. Additionally, objects have high-level properties, defined by the structure of the object, which determine whether it is structurally able to participate in certain interactions. As a result, objects in the Active Game World have low-level properties related to their composition and high-level properties related to their structure.

Having two levels of properties necessitates a two-part approach to modeling objects in a game environment. The first part of the approach is to treat the objects as cells, where each object has only one neighbor (its host cell). The second part of the approach is to consider the high-level properties of the objects.

High-level properties can be assembled into affordances that determine whether the object is able to participate in each interaction. For example, only objects that afford flow can participate in fluid or pressure flow. The two-part approach used for defining the interactions of objects with the environment in the Active Game World is described in the following sections.

Low-Level Properties

At the basic level of physical interactions, objects are the same as cells, because they are both entities in the physical environment that are subject to the rules of physics. Therefore, the first part of the approach to modeling objects in the Active Game World is to treat the objects as cells, where each object interacts exclusively with its host cell.

There are two key types of interactions possible between objects and cells. First, objects within a cell are affected by the cell in the same way that neighboring cells are affected by the cell (exchange of heat, pressure, and fluid). Second, objects affect their host cell as if the host were the object's only neighbor. Therefore, the object both affects and is affected by its host cell.

In some components of the Active Game World, such as heat transfer, there is no difference between object-to-cell interactions and cell-to-object interactions. However, in other components there needs to be a differentiation between these two types of interactions.

For example, a high-pressure object in a comparatively low-pressure cell will explode. However, the effect of the inverse of this state, a high-pressure cell (with a low or high-pressure object), is to damage anything within the cell from high absolute pressure (including the cell itself), rather than the cell exploding.

The interactions that take place between cells and objects are described in this section, with the differences from cell-to-cell interactions highlighted. The following sections also contain pseudo-code for each algorithm, which illustrates how the equations are integrated.

Heat

When transferring heat between two entities, whether they are two cells or an object and a cell, the heat capacities of the two entities first need to be calculated. The main difference between transferring heat between two cells and between a cell and an object is that objects are much smaller than cells, indicated by the mass of the object. As such, the heat transfer between an object and a cell will be much less than the transfer between two cells of the same size.

The heat capacity of the object, *HCObj*, is equal to the specific heat capacity of the object's material, *material(obj).SHC*, multiplied by the mass of the object, *obj.Mass*. Similarly, the heat capacity of the cell, *HCCell*, equals the specific heat capacity of the cell's material, *material(cell).SHC*, multiplied by the mass of the cell.

```
HCObj = material(obj).SHC * obj.Mass
HCCell = material(cell).SCH * cell.Mass
```

The energy flow, *EnergyFlow*, between the object and the cell is equal to the difference between the cell's temperature, *cell.Temp*, and the object's temperature, *obj.Temp*.

```
EnergyFlow = cell.Temp – obj.Temp
```

Once the energy flow is calculated, it can be used to determine whether energy will flow from the object to the cell or from the cell to the object. If *EnergyFlow* is a positive value, the cell's temperature is greater than the object's temperature and heat will flow from the cell to the object. Otherwise, heat will flow from the object to the cell. In the case where the cell's temperature is greater and heat flows from the cell to the object, the *EnergyFlow* is converted from heat to energy by multiplying by the heat capacity of the cell.

```
EnergyFlow *= HCCell
```

Otherwise, if energy is flowing from the object to the cell, the *EnergyFlow* is converted from heat to energy by multiplying by the heat capacity of the object.

```
EnergyFlow *= HCObj
```

In both cases, the *EnergyFlow* is then multiplied by a constant, *ConstantEnergy FlowFactor*, to control the speed of the cell update.

```
EnergyFlow *= ConstantEnergyFlowFactor
```

Subsequently, the new heats for the cell, *cell.NewTemp*, and the object, *obj. NewTemp*, are calculated by dividing the *EnergyFlow* by the heat capacity for the cell and object, respectively. In the case where heat flows from the cell to the object, the cell's temperature decreases and the object's temperature increases. Otherwise, if the heat flows from the object to the cell then the object's temperature decreases and the cell's temperature increases.

```
cell.NewTemp (+/-) = EnergyFlow / HCCell
obj.NewTemp (+/-) = EnergyFlow / HCObj
```

To avoid oscillations (heat moving back and forth between the object and cell), the difference in heat is distributed evenly between the object and the cell if the transfer of heat would result in the entity that previously had less heat having more heat after the exchange.

```
TotalEnergy = (HCObj * obj.NewTemp) + (HCCell * cell.NewTemp)
AverageTemp = TotalEnergy / (HCObj + HCCell)
obj.NewTemp = AverageTemp
cell.NewTemp = AverageTemp
```

The following pseudo-code implements the diffusion of heat in the Active Game World with objects:

```
PROCEDURE heat_obj (obj, cell)

    // Find current heat capacities
    HCObj = material(obj).SHC * obj.Mass;
    HCCell = material(cell).SHC * cell.Mass;
    EnergyFlow = cell.Temp - obj.Temp;

    // Energy flowing from cell to object
    IF (EnergyFlow > 0) THEN
           // Convert from heat to energy
           EnergyFlow *= HCCell;

           // A constant according to cell update speed
           EnergyFlow *= ConstantEnergyFlowFactor;
           cell.NewTemp -= (EnergyFlow / HCCell);
           obj.NewTemp += (EnergyFlow / HCObj);

           // Detect and kill oscillations
           IF (cell.NewTemp < obj.NewTemp) THEN
                  TotalEnergy = (HCObj * obj.NewTemp)
                         + (HCCell * cell.NewTemp);
                  AverageTemp = TotalEnergy / (HCObj + HCCell);
                  obj.NewTemp = AverageTemp;
                  cell.NewTemp = AverageTemp;
           END
    END
```

```
        // Energy flowing from object to cell
        ELSE IF (EnergyFlow < 0) THEN
               EnergyFlow *= -1;

              // Convert from heat to energy
                 EnergyFlow *= HCObj * ConstantEnergyFlowFactor;
                 cell.NewTemp += (EnergyFlow / HCCell);
                 obj.NewTemp -= (EnergyFlow / HCObj);

              // Detect and kill oscillations
              IF (obj.NewTemp < cell.NewTemp) THEN
                    TotalEnergy = (HCObj * obj.NewTemp)
                               + (HCCell * cell.NewTemp);
                        AverageTemp = TotalEnergy / (HCObj + HCCell);
                        obj.NewTemp = AverageTemp;
                        cell.NewTemp = AverageTemp;
                END
         END
     END PROCEDURE
```

Fluid Flow and Wetness

There are two main interactions of fluids with objects: flow and wetness. Fluid flow between an object and a cell is relatively simple compared to fluid flow between cells, because there is no height difference between a cell and the objects in the cell.

There are two cases for fluid flow between an object and a cell modeled in the Active Game World, flow from cell to object and flow from object to cell. The two fluid flow cases are discussed in this section. The wetness of an object is dependent on the wetness of the object's host cell and the absorbency of the object's material.

Flow from Cell to Object

Fluid flows from a cell to an object within the cell if the cell contains fluid and if the object contains less than the maximum amount of fluid it can hold, which depends on the size of the object. The object must also afford flowing, which is discussed in the "High-Level Properties" section.

The amount of fluid that flows into the object from the cell, *flow*, is equal to the difference between the amount of fluid in the cell, *cell.Fluid*, and the amount of fluid in the object, *obj.Fluid*. This difference is then divided by four, because the fluid from the cell must be divided amongst the cell's neighbors and the object and one quarter provides a good approximation.

The difference between *cell.Fluid* and *obj.Fluid* is multiplied by the ratio of the mass of the object, *obj.Mass*, to the mass of the cell, *cell.Mass*, to account for the difference in size between the cell and the object. Consequently, the object is filled with the same proportion of fluid as is in the cell.

```
flow = (cell.Fluid – obj.Fluid) * 0.25 * (obj.Mass / cell.Mass)
```

Flow from Object to Cell

Fluid flows from a cell when the material in that cell contains the maximum amount of fluid that it can hold. As a result, the excess fluid spills from the cell into its neighboring cells. However, fluid flow from an object does not depend on the amount of fluid that the object's material can hold. Rather, it depends on the maximum amount of fluid that the structure of the object can hold, which depends on the volume of the object.

When the fluid in an object exceeds the volume of the object, as determined by the mass of the object, the excess fluid spills over into the object's host cell. The excess fluid in an object, *excess*, is the difference between the amount of fluid in the object, *obj.Fluid*, and the mass of the object, *obj.Mass*.

```
excess = obj.Fluid – obj.Mass
```

If the object contains more fluid than it can hold, it overflows into the host cell. The amount of fluid that flows, *flow*, from the object to the cell is equal to the excess, multiplied by the ratio of the *obj.Mass* to the *cell.Mass*.

```
flow = excess * (obj.Mass / cell.Mass)
```

The following pseudo-code implements fluid flow between objects and cells in the Active Game World:

```
PROCEDURE fluid_obj (obj, cell)

    // will flow into obj if cell has any fluid
    IF ((cell.Fluid > 0)

        // and if obj is not full of water
        AND (obj.Fluid < (material(obj).MaxFluid * obj.Mass/cell.Mass))

        // and object affords flowing
        AND AffordsFlow(obj))
    THEN
        // should fill obj with same proportion of fluid as in cell
        flow = (cell.Fluid - obj.Fluid) * 0.25 * (obj.Mass/cell.Mass);
```

```
                IF (flow < 0) THEN flow = 0;

                cell.NewFluid -= flow;
                obj.NewFluid += flow;
                IF (cell.NewFluid < 0) THEN cell.NewFluid = 0;
        END

        // will flow from obj to cell if obj is over-full
        excess = obj.Fluid – material(obj).MaxFluid;
        IF ((excess > 0)

                // and object affords flowing
                AND AffordsFlow(obj))
        THEN
                // objects are smaller than cells
                flow = excess * (obj.Mass / cell.Mass);
                cell.NewFluid += flow;
                obj.NewFluid -= flow;
        END
END PROCEDURE
```

Pressure

The three main interactions of pressure with objects modeled in the Active Game World are (1) high-pressure damage, (2) flow, and (3) explosions. Objects that are located in a cell that has high enough pressure are damaged, as is the cell itself. Similar to fluid, pressure can flow from objects to cells and cells to objects. Finally, if an object contains significantly more pressure than its host cell then it will explode, under the right conditions.

The pressure ratio between an object and a cell that is necessary for an explosion to occur depends on the object's material. For example, a metal crate can hold more pressure than a wooden crate before it explodes, due to the strength of the material. Additionally, the metal crate exploding results in a bigger explosion than the wooden crate, due to the increased pressure capacity.

In the "Active Game Environment" section, neighboring cells with high pressure differences exploded. However, it was decided that cells themselves should not explode in the Active Game World, due to the scale of cells in a real-time strategy game.

It would make sense for small cells in a human-sized game to be able to explode based on pressure differences, but not for cells several kilometers across to explode in strategy games. Instead, only objects can explode within cells in the Active Game

World. The rules used for each of the pressure interactions are explained in this section.

First, if an object is located in a cell that has high absolute pressure then the object will be damaged. If the pressure of the cell, *cell.Pressure*, is greater than the constant for high absolute pressure, *high_pressure*, the object will incur damage, *obj.NewDamage*, proportional to the cell's pressure.

```
obj.NewDamage += cell.Pressure * pressure_damage_const
```

Second, if the pressure in the cell, *cell.Pressure*, is higher than the pressure in the object, *obj.Pressure*, pressure flows from the cell to the object. Pressure flows between a cell and an object only when the object affords flow. The amount of pressure that flows from the cell to the object, *PressureFlow*, is equal to the difference between the cell's pressure, *cell.Pressure*, and the object's pressure, *obj.Pressure*. The rate of the *PressureFlow* between a cell and an object is modified by the ratio of the size of the object, *obj.Mass*, to the size of the cell, *cell.Mass*.

```
PressureFlow = (cell.Pressure – obj.Pressure) * (obj.Mass / cell.Mass)
```

Third, if the pressure in an object is substantially greater than pressure in its host cell, it is possible that an explosion will occur. The ratio of the pressure, *pressure_ratio*, in the object to the cell is calculated.

```
pressure_ratio = obj.Pressure / cell.Pressure
```

If the *pressure_ratio* is greater than the pressure ratio required for an explosion, *explosion_ratio*, an explosion will occur. The *explosion_ratio* is modified by the strength of the object's material, *material(obj).Strength*, so that the *explosion_ratio* required for an object that is made of a relatively weak material, such as wood, is much lower than for an object made of a much stronger material, such as metal. Also, the object must afford exploding.

The explosion releases an amount of heat that is proportional to the size of the explosion, *pressure_ratio*, multiplied by a constant, *explosion_const*. The heat generated by the explosion of the object is released into the host cell, increasing the temperature of the cell, *cell.NewTemp*.

```
IF ((pressure_ratio > (explosion_ratio * material(obj).Strength))
       AND AffordsExploding(obj)) THEN
       cell.NewTemp += pressure_ratio * explosion_const
END
```

When an explosion occurs, as well as releasing heat, the pressure from the object is transferred to the cell. The amount of pressure that flows from the exploded object to the cell, *PressureFlow*, is equal to the difference between the pressure of the object and the pressure of the cell.

```
PressureFlow = obj.Pressure – cell.Pressure
```

If an explosion does not occur because the *pressure_ratio* is too low, pressure will flow only from the object to the cell. In order for pressure to flow from the object to the cell, the object must afford flowing, as discussed in the "High-Level Properties" section. The amount of pressure that flows from the object to the cell, *PressureFlow*, is equal to the difference in pressure between the object and the cell. The rate of the *PressureFlow* between an object and a cell is modified by the ratio of the size of the object, *obj.Mass*, to the size of the cell, *cell.Mass*.

```
PressureFlow = (obj.Pressure – cell.Pressure) * (obj.Mass / cell.Mass)
```

The following pseudo-code implements pressure diffusion between objects and cells and explosions in the Active Game World:

```
PROCEDURE pressure_obj (obj, cell)

       // high absolute pressure in cell immediately damages object
       IF (cell.Pressure > high_pressure) THEN
              obj.NewDamage += cell.Pressure * pressure_damage_const;
       END

       // cell pressure is higher than object pressure
       // and object affords flowing
       IF ((cell.Pressure > obj.Pressure) AND AffordsFlow(obj)) THEN
              // flow of pressure: cell to object
              PressureFlow = (cell.Pressure - obj.Pressure)
                    * obj.Mass/cell.Mass;
              obj.NewPressure += PressureFlow;
              cell.NewPressure -= PressureFlow;
       END

       // high pressure in object – causes explosion
       ELSE IF (obj.Pressure > cell.Pressure) THEN
              // ratio of object pressure to cell pressure
              // modified by obj material
              pressure_ratio = (obj.Pressure / cell.Pressure);
```

```
            // if pressure difference is great enough then explode
            IF ((pressure_ratio > (explosion_ratio
                   * material(obj).Strength))
                   and AffordsExploding(obj)) THEN

                   cell.NewTemp += (explosion_const * pressure_ratio);
                   PressureFlow = obj.Pressure - cell.Pressure;
                   cell.NewPressure += PressureFlow;
                   obj.NewPressure -= PressureFlow;

            // flow of pressure: object to cell
            ELSE
                   // object affords flowing out of
                   IF (AffordsFlow(o)) THEN
                          PressureFlow = (obj.Pressure - cell.Pressure)
                                * obj.Mass/cell.Mass;
                          obj.NewPressure -= PressureFlow;
                          cell.NewPressure += PressureFlow;
                   END
             END
      END
END PROCEDURE
```

Fire

The rules for the burning of an object are the same as the rules for the burning of a cell of the terrain, because burning is dependent only on the properties of the object or cell and its material. The burning temperature of the object, *Temp*, is the difference between the temperature of the object, *obj.Temp*, and the flashpoint of the object's material, *material(obj).FlashPoint*, modified by the wetness of the object, *obj.Wetness*.

```
Temp = obj.Temp – (material(obj).FlashPoint + obj.Wetness)
```

If the burning temperature is greater than zero, the object will incur damage proportional to the burning temperature multiplied by the burning rate of the object's material, *material(obj).BurnRate*, modified by the wetness of the object and multiplied by a constant.

```
obj.NewDamage += ((Temp * material(obj).BurnRate) – obj.Wetness)
        * burn_const
```

The burning temperature of the object must then be converted to an actual burning value, *Burn*. If the burning temperature is greater than double the maximum burning rate of the object's material, *material(obj).MaxBurn*, *Burn* is equal to the maximum burn rate of the object's material. Otherwise, *Burn* equals one minus the temperature divided by the maximum burn rate, multiplied by the temperature.

```
Burn = (1.0 – ((0.25*Temp) / material(obj).MaxBurn)) * Temp
```

As the object burns, the fire releases heat and the object heats up from burning. The amount the object is heated is proportional to the burning value multiplied by the burning temperature of the object's material, *material(obj).BurnTemp*, and a constant value.

```
obj.NewTemp += (Burn * material(obj).BurnTemp) * BurnHeatConst
```

The following pseudo-code implements objects burning in the Active Game World:

```
PROCEDURE fire_obj (obj)

       //Burning temperature
       tempDifference = obj.Temp - (material(obj).FlashPoint +
         obj.Wetness);

       //Damage the cell
       IF (tempDifference > 0) THEN
              obj.NewDamage += ((tempDifference * material(obj).BurnRate)
                   - obj.Wetness) * const;
       END

       //Convert to actual burning value
       IF (tempDifference > (material(obj).MaxBurn * 2)) THEN
       Burn = material(obj).MaxBurn;

       ELSE IF (tempDifference > 0) THEN
             Burn = ((1.0 - ((0.25 * tempDifference) /
             material(obj).MaxBurn))
                   * tempDifference);
       END

       IF (Burn > material(obj).MaxBurn) THEN
             Burn = material(obj).MaxBurn;
       END
       Burn -= obj.Damage;
```

```
        IF (Burn < 1) THEN Burn = 0;

        // Heat the object up from the burning
        obj.NewTemp += (Burn * material(obj).BurnTemp) * BurnHeatConst;
        obj.Burn = Burn;

    END PROCEDURE
```

Wind and Rain

Currently, wind and rain do not directly affect the objects in the Active Game World. Instead of rain directly affecting the objects, rain affects the fluid in a cell, which determines the wetness of a cell, which in turn determines the wetness of the objects in the cell. Wind does not affect objects, because the wind is on the scale of cell to cell, rather than within a cell.

Within the context of a strategy game, it is unlikely that wind would affect the objects in the environment as they are mostly very large objects, such as buildings. Wind would have a more interesting effect when modeled in human-sized games, such as first-person shooter or role-playing games, where there are small objects to blow around, such as paper and cans.

High-Level Properties

Whereas the low-level properties discussed in the previous section are pertaining to an object's composing material, the high-level properties relate to an object's structure. Low-level properties consist of continuous numeric values, such as a material's flashpoint or burning temperature, which are substituted into equations to determine when and how that material will be affected by heat, water, and pressure.

High-level properties consist of discrete descriptors of an object's structural properties, which constrain the interactions that an object will be able to take part in due to its physical structure. For example, only an object that has volume will be able to hold water, whereas an object's volume has no effect on whether it will catch on fire. High-level properties are implemented in the Active Game World as Boolean values, such as *is_open = false* or *has_volume = true*.

Objects with different combinations of high-level properties afford different types of interactions. An object's high-level properties can be combined to form different affordances. For example, for an object to afford fluid flow it must have volume, some kind of opening, and be solid (as opposed to perforated). An object can contain a large amount of pressure only when it does not have an opening.

Affordances can then be used as preconditions for certain interactions. For instance, only objects that afford flow can engage in fluid or pressure flow. Affordances are implemented in the system as functions that read in an object, check the relevant properties of the object, and return a true or false value to indicate whether the object affords the given behavior.

It would be possible to define affordances for materials in the same way as for structure. For example, rather than having a flashpoint, burning temperature, and so on, a material could simply afford burning. However, encoding these properties at a high level would reduce the emergent potential of the environment. These properties are modeled at a low level so that the high-level behavior will be emergent.

Modeling the high-level properties at a low level (reducing the object to a set of cells to define its structure) would be impractical and provide little benefit to the gameplay in comparison to the computational complexity. Therefore, it is necessary to separate the properties of objects into two levels, a low level related to the object's material to allow emergent behavior and a high level related to the object's structure to constrain the object's set of interactions to what is possible given its physical structure.

Heat and Fire

Heat is a system that is solely dependent on the composition of an object. The structure of an object does not determine whether it can transfer heat, only the material of the object does. For example, an object made of metal will transfer heat in the same way, dependent on the rules of heat transfer, whether it is a metal crate or a metal chair. Similarly, the burning of an object depends only on its material, not its structure. A wooden table will burn in the same way as a wooden house, following the rules of fire, which take the object's mass into consideration.

Fluid Flow and Wetness

There are two behaviors of fluids that need to be considered with respect to objects. First, fluids can fill objects that have structures that afford filling. Second, fluids can wet objects, dependent on the material of the object.

The second behavior depends solely on the material of the object. Objects made of wood can absorb a considerable amount of water, whereas objects made of metal can absorb substantially less water. As discussed in the previous section, the wetness of an object is determined by the wetness of its host cell and the object's material.

Objects can be filled with fluid only when their structure affords filling or flow. There are three main high-level properties that have been identified for an object to afford flow. First, an object must have volume, because an object such as a sword or a book cannot hold fluid. Second, an object must have an opening for the fluid to flow into, because a crate that is sealed has volume but no way for the fluid to enter. Third, the object must have solid sides (as opposed to perforated), because a crate that is made of wire mesh would not be able to hold fluid. Therefore, an object that *has_volume*, *is_open*, and *is_solid* affords flow.

The function *AffordsFlow* is used as a precondition for fluid flowing from an object to a cell and from a cell to an object.

```
FUNCTION AffordsFlow (obj)
       AffordsFlow = false
       IF (obj.has_volume AND obj.is_open AND obj.is_solid) THEN
              AffordsFlow = true
       END
       RETURN AffordsFlow
END FUNCTION
```

Pressure

There are two affordances relevant to pressure modeled in the Active Game World. First, pressure can only flow to and from objects that afford flow, identical to fluid flow. Second, only objects with structures that afford exploding (can contain high pressure) can explode.

For the first behavior, pressure flow, the function *AffordsFlow* is used as a precondition for pressure flowing from an object to a cell and from a cell to an object. Therefore, only an object that *has_volume*, *is_open,* and *is_solid* affords pressure flow.

The same three properties used in fluid and pressure flow are relevant to whether an object affords exploding. First, in order for an object to hold the necessary pressure, the object must have volume. Second, the object must have solid sides to keep in the pressure. Third, the object must be closed (or not open), which also keeps in the pressure.

If the object does not have these properties, it is not possible for the pressure inside the object to increase enough to exceed the external pressure so much that it explodes. Therefore, an object that *has_volume*, not (*is_open*), and *is_solid* affords exploding.

The function *AffordsExploding* is used as a precondition for an object exploding.

```
FUNCTION AffordsExploding (obj)
       AffordsExploding = false
       IF (not(obj.is_open) AND obj.has_volume and obj.is_solid) THEN
              AffordsExploding = true
       END
       RETURN AffordsExploding
END FUNCTION
```

Key Terms

- *Property-based objects* are encoded with types of behavior and rules for interacting.
- *Low-level properties* define how the matter of the objects interacts in the world.
- *High-level properties* are defined by the structure of the object and determine whether it is structurally able to participate in certain interactions.

Observable Behavior

In games, scenarios and sequences of actions are often specifically scripted and coded. The advantage of using cellular automata and a systemic approach is that the observable behavior of any object or situation in the game can be dynamic and emergent, without needing to be scripted individually for each object or situation. This allows the game engine to be flexible and respond consistently and realistically to a wide range of actions that players may take and events in any situation in the game.

This section examines four possible scenarios within a strategy game world, focusing on the interactions between the objects and the environment. For each scenario, the advantages and disadvantages of the Active Game World and a conventional scripted system are compared and evaluated in terms of the observable behavior of the objects, environment, and the interactions between the two. In order to tune the behavior of an emergent game system, a series of scenarios such as these need to be run, and the parameters and rules tuned until the game behaves as expected. The four scenarios are as follows:

- Heat and fire
- Fluid and wetness
- Pressure and explosions
- The integrated system including each of the previous components

ON THE CD

Demos for each of these scenarios in the 3D Active Game World can be found on the CD-ROM.

Scenario 1: Heat and Fire

Consider a situation where a wooden building, located in a forest, is set on fire (see Figure 6.12). Not only should the building burn and be damaged as a result, but the heat from the fire should also spread into the surrounding area. As the fire burns, the surrounding forest should heat up and catch on fire, which will cause subsequent damage to the forest and the spread of heat to surrounding areas. The heat and fire will continue to spread until the fuel is exhausted (there are no more trees) or until some other event causes the fire to stop (such as rain).

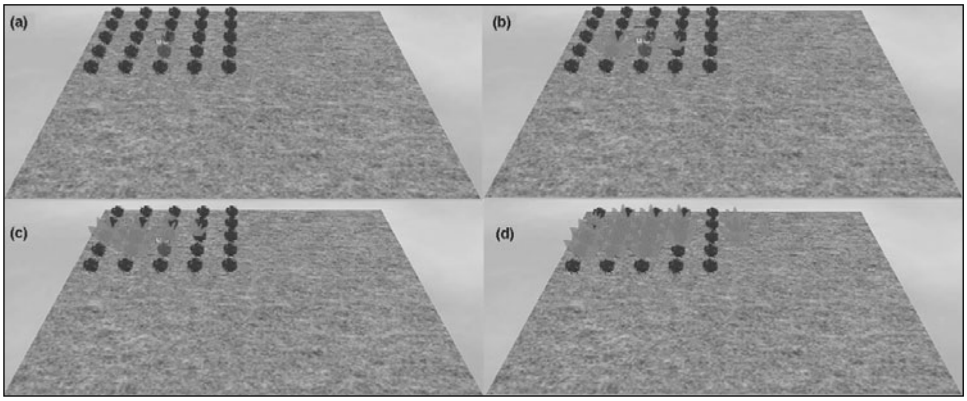


FIGURE 6.12 A wooden building in a forest is set on fire (a), the fire spreads to the forest (b–c), and the surrounding area (d).

Scripted System Evaluation

In a system where the heat and fire scenario is scripted, the observable behavior of the fire burning the building, and then burning the surrounding area, and eventually stopping would need to be specifically encoded. This would include specifying how long the fire will burn the building and each section of the terrain, how much damage will be caused by the fire in the process, and what the fire will look like while it is burning.

Specifically scripting the observable behavior of the heat and fire scenario means that the scenario will behave only in a pre-scripted and limited way. Any changes to the scenario would require the behavior of the fire to be manually rescripted.

For example, there could be more than one building, the building could be made of brick instead of wood and the surrounding terrain could be grass instead of forest. As the observable behavior of burning brick is different than burning wood, the script would need to be updated to encode the new observable behavior. The alternative would be to have a brick building burn in the same way as a wood building, which would not be as believable.

The same problem would exist for the surrounding terrain. If the building were surrounded by grass rather than forest, the way the fire spreads from the building to the terrain and then through the terrain should be significantly different for grass than for forest, which would require further alterations to the script.

The major problem that exists for specifically scripting the heat and fire scenario is that there is considerable initial effort in implementing the scenario and small changes to the material of the terrain and objects requires the script to be rewritten or the same behavior to be used for different scenarios.

Active Game World Evaluation

When the heat and fire scenario is implemented using the Active Game World, the resulting behavior depends on the underlying properties of the materials of the cells and objects, which are used by the equations for heat and fire to determine the behavior of the system dynamically. As such, the system is robust to changes in the heat and fire scenario, such as the number and position of buildings, the material of the buildings, and the material of the surrounding terrain.

Whereas a scripted system requires the observable behavior to be encoded directly, the Active Game World works at a lower level, so that the observable behavior of the system is emergent. Emergent behavior is possible because cells and objects are subject to the same rules of heat and fire and their low-level properties and materials determine how they are affected by the system's rules.

Scenario 2: Fluid and Wetness

The second scenario presents a case study for fluid and wetness, in which rain falls on a hillside and the water from the rain runs down the hill into the valley below (see Figure 6.13). Subsequently, the water pools in the valley and floods the village that is located in the valley.

In the village there are wooden houses, which absorb water from the flood. There is also a factory made of metal, which absorbs much less water from the flood. Some of the buildings are closed up, but others are open and become flooded. Eventually, the rain stops and over time the flood water and buildings dry out.

Scripted System Evaluation

Scripting the fluid and wetness scenario would require many details to be considered. The water must follow a certain path to flow down the hill into the valley, the water will accumulate in the valley to cause a flood, the flood water will soak and flow into the buildings in a certain way, and the water will dry out.

For the fluid and wetness scenario, the observable behavior of each component would need to be scripted, paying careful attention to detail. Due to the complexity of the fluid and wetness scenario, the resulting behavior would be highly specific to the situation.

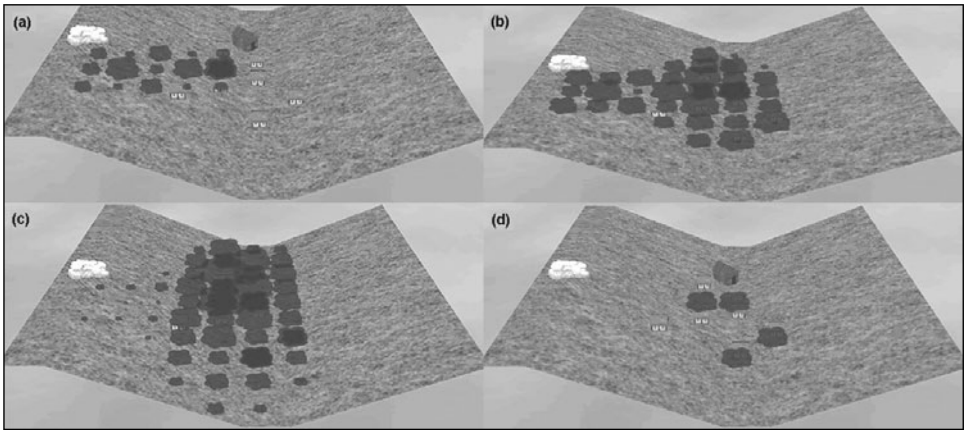


FIGURE 6.13 Rains falls on a hillside (a) and runs into the valley below (b). The houses that are open are flooded (c) and remain flooded for a while after the water has dried up (d).

For example, the contours of the hill, the location of the town, or the position of the rain would each greatly change the observable flowing of the water. If the town were located on the side of the hill, if the rain fell on the other side of the hill or if the contours of the hill forced the water to flow in a different direction then the town would not flood.

Additionally, when the water does flood the town, the way it interacts with each type of building is different and this behavior would need to be specifically scripted. If the buildings were different, in terms of material or affordances (such as whether they're open or not), the behavior of the water interacting with the buildings would need to be rescripted.

There are several variables that affect the observable behavior in the fluid and wetness scenario that require careful attention to detail and considerable initial effort in implementing. Additionally, small changes to any of the variables would give rise to significant changes in the observable behavior of the system, requiring considerable effort in maintenance and updating.

Active Game World Evaluation

In contrast to scripted systems, the complex observable behavior of the fluid and wetness scenario emerged in the Active Game World as a result of the simple, low-level rules interacting with each other. There are three main components of the Active Game World that allow the behavior in fluid and wetness scenario to be emergent.

First, as with the previous scenario for fire and heat, objects and cells have common low-level properties and materials and are subject to the same rules, which allow complex, high-level behavior to emerge. The low-level properties allow the varying materials of the objects and terrain to absorb water differently.

Second, objects in the Active Game World have affordances so that structurally different objects behave differently. The high-level properties allow water to interact differently with buildings, depending on their structure (water can flow only into buildings that are open).

Third, in the Active Game World, the way that water flows over the terrain is determined by the contours. Consequently, in the fluid and wetness scenario, the water flow over the terrain is dynamically determined by where the rain falls and the contour of the terrain, which may or may not cause the town to flood.

The Active Game World allows the behavior in the fluid and wetness scenario to be dynamic and emergent due to the properties of the cells and objects, the affordances of the objects and the contours of the terrain, which all contribute to the complex, high-level behavior that is observable to the players.

Scenario 3: Pressure and Explosions

The third scenario presents a case study with pressure and explosions, in which a high-pressure object is placed in an area with significantly lower pressure, located in a village (see Figure 6.14). The object explodes and the pressure in the surrounding area is immediately increased.

As a result of the high absolute pressure, the surrounding buildings are damaged. Some buildings are more damaged than others, depending on their material. After the immediate release of pressure, the pressure flows out from the explosion into surrounding buildings and into the surrounding area.

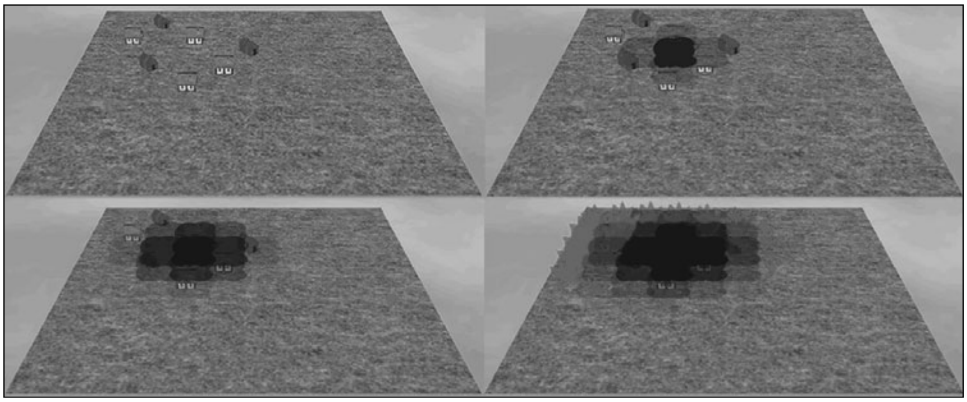


FIGURE 6.14 A high-pressure object is placed in an area with significantly lower pressure (a). The effects of three different-sized explosions are shown (b–d).

Scripted System Evaluation

The factors that need to be considered when implementing the pressure and explosions scenario include the size of the explosion, the trigger for the explosion, the spread of pressure, damage to objects and cells, and any secondary effects of the explosion. In a system that is specifically scripted, each of these factors needs to be specified and hard-coded.

The size of the explosion will determine the area that the explosion effects and how much damage is done. If the size of the explosion varies, the variables need to be manually adjusted accordingly.

In a scripted system, there would need to be a scripted trigger for an explosion to occur, such as a grenade being thrown. Additionally, any secondary effects of the explosion, such as a fire starting, would need to be hard-coded.

The effect that the explosion has on different objects and terrain would also need to be scripted. For example, more damage would be done to a tin can than to a lead barrel and this behavior would need to be specifically scripted.

Active Game World Evaluation

In the Active Game World, the rules for explosions and pressure flow allow any changes in the system to be automatically accommodated. For example, larger explosions cause more heat to be released from the explosion, automatically causing a larger area to be affected and more damage to be done to nearby objects and cells.

Events do not need to be scripted to trigger explosions (although they can be), because an explosion will naturally occur under the right conditions. Similarly, secondary effects, such as the release of heat and subsequent fires occur naturally as a result of the explosion, rather than needing to be scripted.

The flow of pressure between the objects and environment occurs dynamically, due to the affordances of the objects. Also, objects and cells made of different materials are affected by the explosion to varying degrees, depending on their low-level properties.

The Active Game World can respond dynamically in the pressure and explosions scenario due to the affordances of objects (determining how pressure will flow between objects and cells), the low-level properties of objects and cells (determining how they will be affected by explosions) and the global rules of the system.

The global rules of the Active Game World allow the explosion and secondary effects to occur naturally, as well as variations to the pressure and explosions scenario (the size of the explosion) to be automatically accommodated.

Scenario 4: Integrated System: Heat, Pressure, and Fluid

The fourth scenario is a combination of the previous three scenarios. Rain falls on a hillside and runs down the hill, causing a flood in a village that is located in the valley below. As the flood washes through the village, the buildings made of wood absorb fluid, as does the grassland on the hillside and in the valley.

Immediately afterwards, a small explosion occurs as a result of a high-pressure object in the village. The surrounding buildings in the village are damaged, but due to their wetness they are difficult to ignite. The wooden buildings catch on fire and burn briefly and the fire spreads to the forest on the hillside on the other side of the valley (see Figure 6.15).

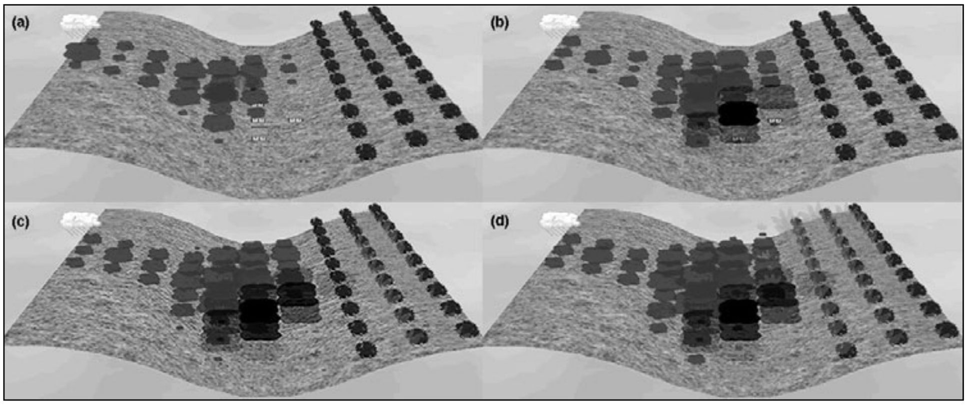


FIGURE 6.15 Rain falls on a hillside and causes a flood in the valley below (a). An explosion occurs in the village (b), damaging the buildings and causing a fire that spreads to a nearby forest (c–d).

Scripted System Evaluation

The integrated scenario combines the attributes of each of the previous scenarios. The same problems exist for the specifically scripted system as in each individual scenario, but to a far greater degree due to the increase in complexity in the integrated scenario.

There are many more variables in the integrated scenario, including the number, composition, structure, and placement of buildings; the composition, contours and layout of the terrain; as well as attributes of the flood, explosion, and fire. Changes that are made to any of these variables present complications for every aspect of the observable behavior, including water flow, fire, explosions, and damage, giving rise to a problem of combinatorial complexity.

Active Game World Evaluation

Similar to a scripted system, the increased complexity of the integrated scenario has an impact on the performance of the Active Game World. However, this impact increases the value and visibility of the emergence of the Active Game World. In the integrated scenario, the rules of each of the systems of heat, pressure, and fluid, as well as wetness, fire, and explosions, combine with the affordances of the objects, the low-level properties of the objects and cells, and the contours of the terrain, so that many simple, local interactions occur simultaneously to give rise to a living, complex environment with realistic, global, observable behavior.

Property-Based Object Summary

In the second part of this chapter, you integrated game objects into the cellular automaton of the Active Game World. You first implemented the objects as though they were cells, using the same low level properties, based on their material. Subsequently, you gave the objects high-level properties, based on their structure, to constrain the possible interactions of the objects with the environment.

The environmental systems that were modeled for objects were the same as for the cells: heat, pressure, fluid, fire, wetness, and explosions. The key differences between the object-to-cell interactions and the cell-to-cell interactions were that objects have only one neighbor (its host cell) and objects have affordances (determined by high-level properties).

For the objects in the Active Game World, high-level properties were related to an object's structure, whereas low level properties were related to its composition (material). Structure was modeled at a high level, because modeling it at a low level would result in impossible computational complexity for a computer game, with marginal improvement in behavior. Composition was modeled at a low level as modeling it at a high level would substantially decrease the emergent capacity of the objects and remove the link to the game environment.

For game objects in the Active Game World, determining whether object properties were high or low level was clear cut. However, deciding on the appropriate level to model properties comes down to a trade-off between computational complexity and the gain of modeling at a lower level. In some cases, it will be prohibitively expensive to model properties at a low level, in other cases there will be no gain in modeling properties at a low level, and other cases will not be as clear cut.

As with the environmental interactions, each of the major systems in the Active Game World (heat, fluid, and pressure) demonstrated various advantages over conventional scripted approaches for game object interactions.

For heat and fire, the major problems that exist for specifically scripted systems is that changes to the material of objects and the terrain, as well as the number and placement of objects, requires the script to be rewritten. However, the model of heat and fire implemented in the Active Game World uses the underlying properties of the materials of the objects and cells, as well as global rules for heat and fire, resulting in emergent high-level behavior.

Scripting the behavior of fluid and wetness requires considerable initial effort and careful attention to detail due to the number of variables that affect the observable behavior of the system, such as the material of objects and cells, structure of objects, and contours of the terrain. However, the Active Game World allows the behavior of fluid and wetness to be emergent due to the global rules of the cellular automaton, which dynamically process the low-level properties of cells and objects, the high-level properties of objects, and the contours of the terrain.

For pressure and explosions, the factors that are problematic in scripted systems include the size of the explosion, the trigger for the explosion, the spread of pressure, damage to objects and cells, and any secondary effects of the explosion. In a scripted system, the cause and effect of each of these factors needs to be anticipated and specifically scripted. However, the Active Game World can respond dynamically to pressure and explosions due to the affordances of objects (for pressure flow), the low-level properties of objects and cells (effect of explosions), and global rules of the system (secondary effects and variations to scenario).

The integrated scenario illustrated that a complex system with many variables poses significant problems for scripted systems, because changes to one variable in the system present complications for every aspect of the observable behavior of the system. However, in the Active Game World, the rules of each of the systems of heat, pressure, and fluid, as well as fire, wetness, and explosions, combine with the affordances of the objects, the low-level properties of the objects and cells, and the contours of the terrain, so that many simple, local interactions occur simultaneously to give rise to a living, complex environment with realistic, global, observable behavior. Also, timing is very important in games and careful attention must be paid when setting up the right timing in game scripts. For example, the game must be scripted to create a cause (for example, an explosion) before its effects (objects being damaged as a result of the explosion). In the Active Game World, timing is implicit in the rules of the system, because the effects are actual outcomes of the cause, rather than scripted reactions.

Objects with a property-based design can be incorporated into an Active Game World by dividing their properties into two levels, high level and low level. An object's low-level properties are the same as the cells in the Active Game World and as such the objects are easily incorporated into an Active Game World, because they are subject to the same rules. An object's high-level properties constrain which rules affect various types of objects, because different objects have different affordances, unlike cells of the environment, which are uniform in structure.

Property-based objects, in conjunction with the Active Game World, can facilitate emergent behavior and gameplay by means of the global rules of the system and the properties of the objects and cells. These aspects of the Active Game World enable real-time, dynamic processing of the game environment and objects to generate emergent, high-level behavior that was not specifically planned and scripted in advance.

The game objects design presented in the chapter addresses the problems associated with current scripted systems for game developers and game players. For game developers, the objects have general, global properties, simplifying design of objects, planning of interactions, and subsequent implementation and testing. For the game players, the global properties of the objects give rise to intuitive, consistent, emergent interactions, rather than pre-scripted, specific gameplay. With the use of two-part property-based objects, within an Active Game World, game design can be simplified and gameplay can be enhanced in strategy games, as well as other game genres.

Emergent Game Worlds

Players are dissatisfied with the static, unintuitive, and unrealistic worlds in current games and emergence provides the opportunity to enhance player enjoyment with game worlds that allow consistency, freedom, intuitiveness, and realistic physics.

Previous games, such as *The Sims* and *Half-Life 2*, have included property-based game objects that afford greater freedom and interaction. However, the Active Game World models an emergent game world (the environment itself, as well as its objects), rather than emergent game objects only. Emergent worlds provide far greater potential for interactions and complexity than emergent objects alone.

Obviously, the behavior of the world alone is not a game, but it defines the potential actions and interactions of the players. The more flexible and reactive the game world, the more opportunities and freedom the players have in interacting and affecting the game world. Once the game world is in place, the agents, story, and players can be added to the game. If the game world is static and linear, the remaining game elements can be only static and linear as well.

The same property-based system that is used for cell-to-cell and object-to-cell interactions can be extended for object-to-object, player-to-object, and player-to-player interactions. With a property-based approach and a simple set of rules for affordances and interactions, player interactions can easily be integrated into an Active Game World.

An active environment design, based on cellular automata, can facilitate emergent environmental effects and complex behavior. The use of a cellular automaton, as well as property-based materials, objects, and rules in the Active Game World gives rise to behavior that is not specifically scripted into the system. The behavior of heat, fire, fluid flow, pressure, and explosions responds dynamically to the changing game world and gives rise to second order effects that are not directly specified.

The Active Game World displays advantages related to its ability to dynamically determine and accommodate the specific state of the game world (such as number, type, and position of entities, terrain, and external effects), due to the underlying properties of the cells and objects. The properties of materials allow new materials to transfer heat and burn in reasonable ways that were not predetermined. The rules, height field of cells, and affordances of objects and cells allow fluid flow over contours and with object structures that are not predefined. Explosions occur spontaneously (not triggered) and second order fire effects occur spontaneously. Interactions occur in cells by the rules of heat, pressure, and fluid and these simple interactions sum to give emergent effects (for example, it is not specified that water puts out fire, but water reduces heat and raises the flashpoint and therefore prevents or stops burning).

Cellular automata can facilitate emergence in game worlds in terms of the behavior of environmental systems and the corresponding effects on terrain and objects. The cellular automaton used in the Active Game World can facilitate emergent behavior of environmental systems (water, heat, and pressure) and effects (explosions, fire) with cells of different terrain and objects of different material and structure. Cellular automata are suitable algorithms to form the basis of emergent game worlds. The grid-based structure of cellular automata allows them to be easily integrated into game systems, which are often divided into grids.

Emergent game worlds, such as the Active Game World, can provide an alternative to currently scripted game worlds. Active environments and property-based objects can facilitate the development of more enjoyable games, by giving rise to emergent behavior and the potential for emergent gameplay.

Interview with Richard Evans

Senior AI Engineer, Maxis, Electronic Arts

Richard Evans is senior AI engineer on *The Sims 3*, where he is responsible for the AI architecture. He also likes to get involved in game design.

Previously, he was head of AI at Lionhead Studios, where he designed and implemented the AI for *Black & White*. The artificial creature in *Black & White* holds the Guinness World Record for most intelligent being in a game.

Æ

What is the role of emergence and emergent gameplay in *Black & White*?
The creatures were the main source of emergence in *Black & White*. You could train your creature to behave in a wide variety of ways. For example, to only attack when provoked, to fertilize fields, to only eat old female villagers, or to eat whenever bored. With careful training, it was possible to make your creature almost entirely self-sufficient. He could look after your towns and convert new towns.

What kind of methods and techniques did you use in *Black & White* to create emergent gameplay?
I used a combination of connectionist and symbolic machine-learning algorithms: perceptron training algorithms for learning desires and decision-tree learning algorithms for learning which sorts of objects are appropriate to choose for each desire. There is an article in *AI Game Programming Wisdom* called "Varieties of Learning," which goes into more detail on the techniques used.

What are the major challenges you have faced in creating emergent gameplay in your games?
The larger the possibility space, the harder debugging becomes.

When the creature was first put in the world, he just stared at his feet. It turned out, after some debugging, that his most dominant initial desire was hunger, and he was trying to eat himself.

Another example. When the first quest was implemented in the game, involving a villager who had lost her brother, the creature came along and picked up the poor villager halfway through the cutscene, and carried her off!

Even if you can reproduce a problem, finding out why the creature performed that particular action becomes increasingly complicated as the simulation becomes richer. Visual in-game debugging tools are vital to diagnosing why decisions were made.

My colleagues Jonty Barnes and Jason Hutchens wrote an article about the problem of debugging emergent games in *AI Game Programming Wisdom*: "Scripting for Undefined Circumstances."

What key lessons have you learned about creating emergent games that you would share with other developers?
At a lofty level of generality, it's like this: create emergence by having a large set of things to draw from, and make each game configuration be a small subset of those things. Try to ensure that most things are independent from most other things.

Example: objects in *The Sims*. Each lot has a small subset of all the possible objects. The objects are nearly independent of each other. Interacting with one object rarely affects another (with rare exceptions like one object catching on fire).

Æ

What would you recommend to other game developers trying to create emergent gameplay?
The better the in-game visual debugging tools, the better the perceived quality of the AI.

You have previously advocated the importance of social activities in games. Social interactions give rise to a great deal of complexity in society—can we harness this to create new and emergent gameplay?
One way to generate emergence is to ensure that one action can satisfy many descriptions. For example: turning on a light may also be alerting a burglar.

In *The Sims 3*, we model social situations so that actions can satisfy multiple descriptions. For example, suppose a host has invited a guest over, and then the host goes off and has a bath. The action of having-a-bath can be described also as ignoring-his-guest. It is the visiting social situation itself which allows the action to be redescribed as ignoring-the-guest.

When an action has multiple independent consequences, and the set of consequences depends dynamically on the current social situations, new gameplay possibilities emerge, like flowers on a spring day.

How does *The Sims 3* improve its modeling of social activities over previous Sims games?
When playing *The Sims* for the first time, my Sim invited over another Sim. The host Sim talked to his guest for a moment, before walking off to have a bath!

This (arguably degenerative) emergent behavior arose because the Sim did not understand that, having invited his guest over, there was an expectation to look after his guest. The Sim did not understand the norms implicit in the social practice of visiting! In the words of Bob Dylan, "something is happening here, but you don't know what it is."

The Sims in *The Sims 3* have a much deeper understanding of the social situations they are in. They understand each social situation as a hierarchy of norms. They understand the consequences of violating each norm. They can be in a variety of different social situations simultaneously.

How does *The Sims 3* support and encourage emergent gameplay?
Returning to the high-level strategy of creating emergence outlined above. Create emergence by having a large set of things to draw from, and make each game configuration a small subset of those things. Ensure that each thing is independent from all the other things.

Æ

What kind of methods and techniques are used in *The Sims 3* to create emergent gameplay?
In *The Sims 3*, each agent has a small subset of desires from a large pool of possible desires. Satisfying each desire is mostly independent from other desires. This creates a vast combinatorial set of possibilities, but maintains debuggability because each desire can be tested independently (because the consequences of satisfying one desire are nearly entirely orthogonal to the consequences of satisfying other desires).

How do you go about tuning the rules for an emergent system? Do you follow a structured process?
We spend a lot of time writing tools which allow us to visualize the state of the AI from various different perspectives. The better your debugging and visualization tools, the less buggy your final behavior.

Many developers are reluctant or unable to try new techniques and approaches in their games; are they justified in these fears?
It depends on the game genre—innovation is more acceptable in some genres than others. Innovation is expected in *The Sims*.

What drives you to use non-deterministic techniques in your games?
We use non-deterministic techniques in decision-making, so that Sims do not appear robotic. The probability of performing an action should be a function of the utility of the action. If two actions have similar utility, they should have similar probability. The final stage of decision-making is transforming a utility distribution into a probability distribution.

Summary

In this chapter, you explored the framework for an emergent game world with reactive game objects. You covered the two fundamental components of game worlds: the environment (physical space) and the objects (entities within the space). You learned how to structure, implement, and tune an active environment with property-based game objects. You should now have a solid understanding of how to go about creating an emergent game world using a bottom-up emergent design approach with rules, properties, cells, and objects.

The Active Game World example demonstrates how simple, low-level interactions between the game environment and objects can give rise to high-level emergent behavior and gameplay. In the Active Game World example, you explored the creation of a system to model environmental effects. Using a similar framework with a cell-based environment and property-based objects, you can create a wide range of environmental effects, object behavior, and gameplay.

Class Exercises

1. What other fluids could be simulated in a strategy game using the fluid flow algorithm?
   a. What would be reasonable starting values for the properties of these fluids?
   b. Design scenarios to test and tune the behavior of these fluids.
2. How could wind play a larger role in the Active Game World? What other effects could it have? How would this affect gameplay?
   a. Extend the pressure algorithm to include wind effects.
   b. Write an algorithm to simulate local wind effects.
3. What other affordances could objects be given to interact with the Active Game World?
   a. What properties would be needed to define these affordances? Write algorithms to define the affordances with these properties.
   b. How could these affordances be used in the Active Game World? What interactions would they be used are preconditions for?
4. How could the Active Game World be extended to include object-to-object interactions?
   a. What types of interactions could occur between objects?
   b. How could these interactions be modeled in the Active Game World? What additional data structures, properties, and rules would be required?
   c. What effect would adding these interactions have on the observable behavior of the game world? How would you test these interactions?
5. How could the Active Game World be extended to include player interactions?
   a. What types of interactions could the players carry out?
   b. How could these interactions be modeled in the Active Game World? What additional data structures, properties, and rules would be required?
   c. How would incorporating a player affect the behavior of the game world? What would be the benefits and drawbacks of developing a game in the Active Game World?

7 Characters and Agents

In This Chapter

- Sensing
- Acting

Characters and agents are important types of objects in game worlds, because they give the game life, story, and atmosphere. Characters and agents serve many purposes and hold many positions in games, which contribute to making the game world rich, interesting, and complex.

More than anything else in the game world, players identify with and expect lifelike behavior from game characters. The more responsive, reactive, and dynamic the agents and characters in games, the more lifelike, believable, and challenging the game worlds will become.

Players expect game characters and agents to behave intelligently by being cunning, flexible, unpredictable, challenging to play against, and able to adapt and vary their strategies and responses (Sweetser, Johnson, Sweetser & Wiles, 2003). However, players often find that agents in games are unintelligent and predictable.

Players also believe that agents' actions and reactions in games should demonstrate an awareness of events in their immediate surroundings (Drennan, Viller & Wyeth, 2004). However, many games are proliferated with agents that do not demonstrate even a basic awareness of the situation around them.

Agents are a vital ingredient in creating an emergent game world. Introducing entities that have a choice of how to react to the changing environment amplifies the variation and unpredictability of a system. Reactive agents can extend emergent behavior and gameplay by adding a new level of complexity to the game world.

As agents are able to choose how to react to the environment, they are able to actively change the state of the world in ways that might not have occurred without their intervention. Also, differences between individual and types of agents, such as composition, structure, goals, personality, and so on, can add further variation and complexity. Not only can agents choose how to react to a given situation, different agents will choose to react in different ways in the same situation.

Characters and agents can be used to create emergence in games by being given an awareness of their environment and an ability to react to the changing state of the environment. The agents then become part of the living system of the game, which they sense, react to, and alter.

Agents can be given the ability to respond to the players and other agents, events, and conditions in their environment, as well as their own goals and motivations, by having a model of their environment and a set of rules for reacting. Characters and agents that follow simple rules for behavior, taking into account the complex environment around them, will become emergent entities in the game world.

This chapter covers emergence in individual characters and groups of agents in games. First, I will discuss how characters and agents can sense and interpret their environments using probing, broadcasting, and influence mapping. I will discuss how characters can use their environmental model to guide their movement. I will also present a simple, flexible, general-purpose framework that can be used for agent decision-making. The chapter then explores an example of individual agents reacting to an emergent game world, while pursuing a goal, using the Active Game World from Chapter 6.

The second part of the chapter explores group movement and decision-making. I will discuss methods for achieving emergent group movement in games using agent-based steering behaviors. I will also present a framework for creating emergent group tactics using an agent-based approach, illustrated with a 2D agent-based game called *Halloween Wars*.

Sensing

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators. An agent is considered autonomous if it relies on its own percepts, rather than the prior knowledge of its designer.

The agents in most games rely heavily on the prior knowledge of their designers and little on their current situation. Many agents in games, such as units in strategy games and villagers in role-playing games, do not react to the environment in any way. This behavior demonstrates a lack of situational awareness, which is an agent's dynamic mental model of its operating environment and its place in it.

Situational awareness gives an agent a sense of what is happening in its current environment, what could happen next, what options there are for action and the possible outcomes of those actions. Situational awareness is the foundation for making decisions in complex operational environments.

Giving an agent an awareness of its environment and a way to sense and model the situation is the most crucial step in creating reactive, dynamic, and emergent behavior. The more information and intelligence that can be embedded in the environment, the simpler the agents themselves can become.

An important decision that needs to be made when designing an agent is whether the agent is to be reactive, goal-directed, or some combination of the two. A purely reactive agent is suited to highly dynamic environments where little information about previous actions and states is necessary. At the other extreme, a purely goal-directed agent is suited to a static environment where planning and considering previous moves are highly desirable.

For example, a monster in a first-person shooter or a role-playing game would be more suited to simply reacting to what is currently happening in the game. However, an agent governing the strategy for the AI in a strategy game needs to carefully plan its moves depending on what has happened so far in the game.

The ideal framework for facilitating emergent agent behavior is to have simple agents in a complex environment. The emergence comes from the interactions between agents, between the agents and the players, and the collective interactions of the agents with the game world. In order to achieve this, the agents must be given a way to sense and model their environment.

Some common approaches to sensing game environments are probing, broadcasting, and influence mapping. In short, probing is where agents ask for particular pieces of information as they need it, broadcasting is where the agents are sent all the information whether they need it or not, and influence mapping is where the information is collected, stored, and synthesized separately from the agents.

KEY TERMS

- *Agents* are decision-making entities in games that sense and react to the game world.
- *Reactive agents* react to their environment based on the current situation and require little information about previous actions and states.
- *Goal-directed agents* plan their actions in relation to goals and must consider previous and future states and actions.

Æ

- *Situational awareness* is an agent's mental model of its environment and its place in it.
- *Sensing* allows agents to monitor the changing state of their environment.

Probing

There are some games in which the agents sense and react to other agents by actively probing the environment for information. For example, the agents in *Half-Life* have sight and hearing and periodically "look at" and "listen to" the world. Also, the game *Thief: the Dark Project* uses the same core concepts as *Half-Life*, but with a wider spectrum of states.

The agents in *Half-Life* and *Thief* are based on finite state machines that take into consideration what they can "see" and "hear" in the environment. The method used by these agents is to periodically run through a list of rules to determine whether they sense an opponent (Leonard, 2003):

```
PROCEDURE look
       Get a list of entities within a specified distance
       FOR EACH entity found
              IF I want to look for them
                     AND they are in my viewcone
                     AND I can raycast from my eyes to their eyes
              THEN
                     IF they are the player
                            AND I cannot see the player until they see me
                            AND they do not see me
                            END look
                     ELSE
                            Set various signals depending on my relationship
                                   with the seen entity
                    END
             END
       END
END PROCEDURE

PROCEDURE listen
       FOR EACH sound being played
              IF the sound is carrying to my ears
                     Add the sound to a list of heard sounds
                     IF the sound is a real sound
                            Set a signal indicating heard something
```

```
                END
                IF the sound is a “smell” pseudo-sound
                       Set a signal indicating smelled something
                END
         END
END PROCEDURE
```

The agents must actively check to determine whether they can sense something at given time intervals, unlike real vision and hearing, which arrive at the senses continuously. Depending on the agents' frequency of probing the environment, it is likely that events and actions will be missed.

Probing is quite fast and efficient if there are only a few specific things that the character is checking. However, as the number and frequency of these checks increases, the character can spend most of its time probing the environment. When the character is running a large number of checks, it is likely that most of these probes will be negative. With a lot of agents in a large environment, this can get out of hand quickly.

The agents in *Halloween Wars* use probing to assess the position of their own and enemy entities, as well as their own and enemy flags (see the "Tactics in Halloween Wars" section). The use of probing is possible and efficient, due to the simplicity of the environment, rules, and the knowledge that the agents require for decision-making. If the number or type of entities, and subsequently the required probes was increased, it could quickly become too demanding to use.

ADDITIONAL READING

For further information on probing in games:

- Leonard, T. (2003) Building an AI Sensory System: Examining the Design of Thief: The Dark Project. *Gamasutra*, March 7, 2003. Online at: http://www.gamasutra.com/gdc2003/features/20030307/leonard_01.htm.

BROADCASTING

The agents in *The Sims*, unlike *Half-Life* and *Thief*, continuously receive information from the environment. In *The Sims*, the intelligence is embedded in the objects in the environment, known as "Smart Terrain." Each agent has various motivations and needs and each object in the terrain broadcasts how it can satisfy those needs. For example, a refrigerator broadcasts that it can satisfy hunger. When the agent takes the food from the refrigerator, the food broadcasts that it needs cooking and the microwave broadcasts that it can cook food. Consequently, the agent is guided from action to action by the environment.

When information is broadcast to agents, they are sent all the events that are happening in the game world and they must sort out what they need. This results in a large amount of redundancy and unused information, but the trade-off is flexibility. Agents can be set up to listen for the information they need, and discard the rest. Broadcasters can also be configured to select what information they will send and to which agents.

Although the agents in each of these games are able to sense entities in the environment in some way, they are still unable to sense the state of the environment itself. The agents in *Thief* and *Half-Life* are limited to sensing other agents in the environment and the agents in *The Sims* are limited to sensing other agents and objects in the environment. These agents would still be unable to react to events and states of the environment.

The broadcasting approach, as used in *The Sims*, could be extended to incorporate the environment as well as the game objects. However, there are a finite number of objects in the environment in *The Sims* and a finite number of ways to interact with these objects. When considering the environment itself and the possible events and states in the environment, the problem becomes infinitely more complex and broadcasting could potentially become too computationally demanding and difficult to manage.

Translating the broadcasting approach to a game environment would require every element and event in the environment to project information to every agent in every location. Influence mapping, a technique used in many strategy games, is more applicable to the problem of agents reacting to the game environment, as opposed to other agents or objects.

Additional Reading

For further information on broadcasting in games:

- Millington, I. (2006) *Artificial Intelligence for Games*. Boston, MA: Morgan Kauffman.

Influence Mapping

Influence mapping, a technique used in many strategy games, divides the game map into a grid with multiple layers of cells, each of which contains different information about the game world. For example, the layers could store data for combat strength, vulnerable assets, area visibility, body count, resources, or traversability.

The values for each cell in each layer are first calculated based on the current state of the game and then the values are propagated to nearby cells, thereby spreading the influence of each cell (see Figure 7.1). This influence propagation gives a more accurate picture of the current strategic situation, because it not only shows where the units are and what they are doing, but also what they might do and the areas they potentially influence.

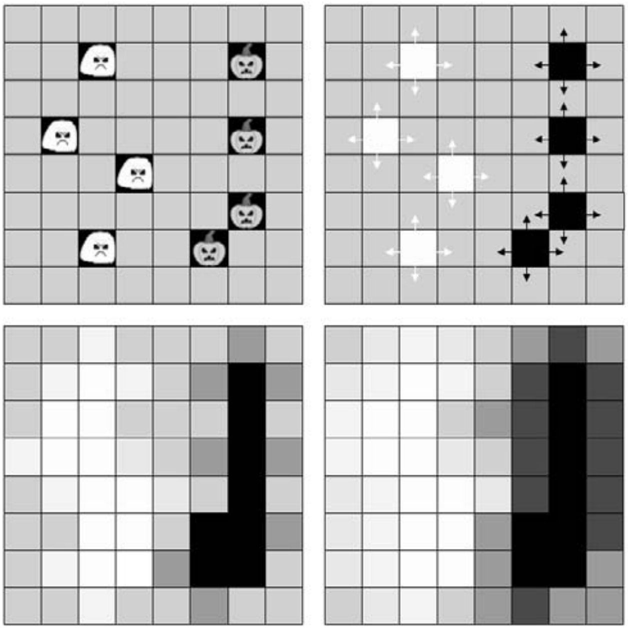


FIGURE 7.1 Influence propagation in an influence map.

Influence maps can be used for strategic assessment and decision making, because their structure makes it possible to make intelligent inferences about the characteristics of different locations in the environment. For example, areas that have high strategic control can be identified, as well as weak spots in an opponent's defenses, the enemy's front, flanks and rear, prime camping locations, strategically vulnerable areas, choke points on the terrain, and other meaningful features that human players would choose through experience or intuition.

Each layer, or set of layers, provides information about a different aspect of the game. For example, the influence map can indicate where a player's forces are deployed, the location of the enemy, the location of the frontier, areas that are unexplored, areas where significant battles have occurred, and areas where enemies are most likely to attack in the future. When these layers are combined, they can be used to make strategic decisions about the game (see Figure 7.2). For example, they can be used to make decisions about where to attack or defend, where to explore, and where to place assets for defense, resource-collection, unit-production, and research.

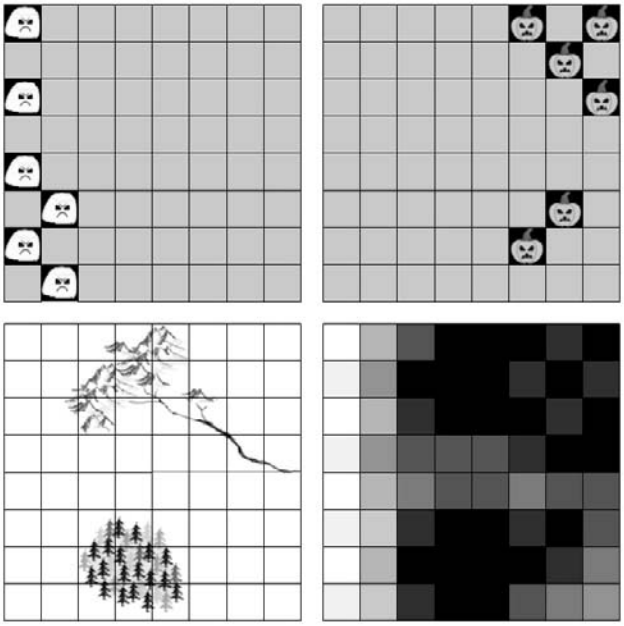


FIGURE 7.2 Layers of the influence map can be combined to make strategic decisions.

Influence maps are commonly used in games for strategic, high-level decision-making. However, it is also possible to use them for tactical, low-level decision-making, such as individual agents or units reacting to the environment. Similar to strategic influence maps, a tactical influence map requires values for the environmental factors that need to be considered and a method for combining these factors into values that can be used for decision-making.

The factors that need to be considered by the agents include any events and effects that are relevant to their decision. The method to combine these factors is the same as in strategic influence maps, a weighted sum that weights the factors in the environment, depending on how important they are for the decision that is being made.

The advantage of using influence maps over broadcasting and probing is that the agent is presented with a single value (calculated using the weighted sum to combine all the factors) instead of numerous messages being sent to the agent about the environment. Therefore, the agents can choose the best cell (based on the weighted sum) for the decision that it is making (such as where to move to avoid danger).

Influence mapping has further advantages over probing, because the agent is continuously adapting its behavior to the environment (rather than probing at given time intervals) and its behavior is a function of its environment (rather than following a prescribed set of rules).

Influence mapping provides passive sensing of a continuous environment (as opposed to discrete entities), allows the agents' situational awareness to evolve as a function of the environment, and gives rise to reactive and emergent behavior. The reactive agents in the Active Game World use influence mapping to sense the state of the environment (see the "Agents in the Active Game World" section).

Additional Reading

For further information on influence maps in games:

- Sweetser, P. (2004) Strategic Decision-Making with Neural Networks and Influence Maps. *AI Game Programming Wisdom 2*. Hingham, MA: Charles River Media, pp. 439–446.
- Tozour, P. (2001) Influence Mapping. *Game Programming Gems 2*. Hingham, MA: Charles River Media, pp. 287–297.
- Woodcock, S. (2002) Recognizing Strategic Dispositions: Engaging the Enemy, *AI Game Programming Wisdom*. Hingham, MA: Charles River Media, pp. 221–232.

Key Terms

- *Probing* involves agents periodically querying the state of their environment.
- *Broadcasting* involves the environment sending game agents information about events and the state of the environment.
- *Influence mapping* provides a persistent map of the changing state of the environment, in terms of the influence of various aspects of the game world.
- *Influence propagation* spreads the influence of a cell to nearby cells, providing a more accurate picture of the current strategic situation and what might happen next.

Acting

After the agent has sensed its environment and has an understanding of its situation, it must choose an action. Even if the agent has a sophisticated world model, if it fails to act or react appropriately, it will appear lifeless and unintelligent. There are a wide range of specific actions that agents are required to take in game worlds, which vary depending on game genre. There are two major types of actions that agents are required to take—individual actions and group actions. Individual actions require the agent to behave autonomously and make decisions based on their own situation and needs. Group actions require the agent to play a role in a group of agents, which involves cooperation and coordination.

Individual

Agents that act individually are usually game characters or enemies. In first-person shooter games, a large proportion of the agents are there to fight the players. The primary actions of these agents are to run, jump, dodge, hide, and shoot enemies. In role-playing games, agents include friendly and enemy characters, as well as monsters and animals. The actions of these agents include talking, fighting, walking, and appearing to follow normal lives and routines. In strategy games, individual agents (or units) must move, attack, guard, and hold positions. Agents in sports games must move around the field or court, score goals, pass, tackle, and so on. The cars in racing games drive around the track, dodge or ram other cars, and sometimes perform stunts. The most common actions for agents in all of these types of games are movement and decision-making.

Movement

Characters spend a large amount of their time moving around, performing actions such as running away, walking around town, driving, moving to a strategic location, or charging at the players. As agents spend such a large amount of their time moving, they must do it efficiently, smoothly, and intelligently. In terms of emergent behavior, the pathfinding of individual characters is not that interesting. There are many good references on pathfinding in games, with A* being the method of choice.

Deciding where to move to, on the other hand, is of more interest, especially when the agent takes into consideration the state of the environment, other agents, as well as its own goals and personality. The reactive agents in the Active Game World use their environmental model to guide their movement (see the "Agents in the Active Game World" section).

Decision-Making

With enough information about their environment, agents can use a simple set of rules to decide how to act and react appropriately. Agent decision-making in games involves choosing an action based on goals, personality, and the current state of the game. Agents in games must decide when to run away, attack, hide, eat, talk, sleep, heal, and so on. The next section presents a simple, flexible, general-purpose framework that can be used for agent decision-making.

Key Terms

- *Acting* involves agents responding to changes in their environment.
- *Movement* requires agents to decide where to move and how to get there.
- *Decision-making* involves agents choosing an action to take in their environment.

Agents in the Active Game World

To exemplify emergence in individual agents, I incorporated reactive agents into the Active Game World described in Chapter 6. In the Active Game World, agents have an identical structure to the objects discussed in Chapter 6. Each agent has a set of properties that include coordinates (position in the cellular automata), temperature, pressure, fluid, mass, wetness, and material.

The materials that agents are composed of have the same properties as materials for terrain and objects, including flashpoint, burning temperature, specific heat capacity, and maximum burning rate. In strategy games, there are a wide range of possible agent types, such as humans (marines), vehicles (tanks), boats (fishing boats), and aircraft (B-52s). Therefore, possible materials for agents include flesh, metal, and wood (see Table 7.1). The values for each material were initially estimated and then tuned until the materials burned at an acceptable rate and duration.

Agents also have the same high-level properties as objects, which encode attributes of the agents' physical structure. These attributes determine whether agents will engage in various interactions (for example, a human cannot be filled with water but a boat can).

Agent Design

The agents in the Active Game World are identical to the objects in terms of structure and composition. The defining difference between agents and objects is that an object is acted upon only by the environment and responds according to its physical composition and structure, whereas agents have a choice of how to respond to the environment. If an agent does not respond to the environment in a way that

TABLE 7.1 Agent Material Properties

		Values		
Property	**Description**	**Wood**	**Metal**	**Flesh**
Flashpoint	Ignition temperature of the material	2000	5000	300
BurnTemp	Multiplier for the temperature that the material burns at (amount of heat released)	5	10	3
BurnRate	Multiplier for the rate that the material burns at (rate of consuming fuel)	10	5	15
MaxBurn	Maximum rate that the material can burn at	300	100	200
SHC	Specific heat capacity: the amount of energy required to heat up this material	100	50	100
MaxFluid	Maximum amount of fluid a cell can hold	50	0	20
Strength	Modifier for the pressure the material can withstand before it breaks	0.6	1.0	0.5

seems reasonable to the players (such as preserving its own safety), they seem unrealistic and unintelligent.

For example, if an object (for example, a crate) is in a room that is on fire, the object's only course of action is to sit there and burn. However, an agent in the same situation would be expected to try and escape from the room in order to preserve its own life. If the agent were only to stand in the room and burn, it would appear neither intelligent nor realistic.

For an agent to react sensibly to the environment in the Active Game World, it is necessary for it to have two things. First, it must have a way to sense the environment and second, it must have a way to choose a suitable reaction, based on what it has sensed.

An agent's understanding of its situation in the Active Game World is represented as a weighted sum of the factors affecting each cell on the map. Based on the utility value of each cell, the agent chooses a cell to move to and reacts at a level that reflects its current situation (for example, if the agent's current cell is on fire then it panics). After the agent chooses a destination, its task is simply to move toward it.

This section discusses the "comfort" function that determines the utility of each cell, the agent's level of reaction, and the agent's choice of destination cell.

Comfort Function

The utility function for the agents in the Active Game World determines how comfortable each cell is for the agents and is therefore called a comfort function. The comfort function is a weighted sum of the factors that affect the agents' comfort in each cell and includes temperature, fire, pressure, and wetness. Each of these factors is weighted according to how distressing it is for the agent.

For human agents in the Active Game World, fire is the most distressing, followed by temperature, pressure, and wetness. However, these weights (*W1*, *W2*, *W3*, *W4*) can be tuned to reflect different priorities of different agents. For example, an alien might find water far more dangerous than heat. The comfort function returns a real value between zero and one, with a lower value representing a more comfortable cell.

```
Comfort = Min(((fire*W1) + (temp*W2) + (pressure*W3) + (wetness*W4)) ,1)
```

The comfort function provides an efficient alternative to the environment sending the agent multiple messages about its state, such as "it's hot" or "it's raining." Instead, the relevant factors are weighted and combined into a single value that gives the agent an estimate of the safety and comfort of its current location. The purpose of the comfort value is twofold. First, it provides a means for the agents to determine how comfortable they are in the current cell and to react accordingly. Second, it provides a means for the agents to assess surrounding cells and find a suitable destination. These two tasks are discussed in the following sections.

Level of Reaction

The comfort function returns a real value between zero and one, which allows the agent to react with varying degrees of distress, providing for more diverse and interesting behavior (see Table 7.2). A comfort value of less than 0.1 represents a comfortable cell and the agent does not react. A comfort value of 0.1 or more and less than 0.3 represents a cell that is uncomfortable and the agent reacts calmly and moves to a more comfortable cell. A comfort value of 0.3 or more and less than 0.6 represents a cell that is distressing and the agent runs from the cell. Finally, a comfort value of 0.6 or more represents a cell that is painful, which causes the agent to panic and run from the cell.

The agent's level of reaction is denoted by its speed of movement, as well as its animation and sound. Scaling the agents' reactions allows the agents to react in varying ways to different situations, while greatly simplifying the process of determining how the agents will react. Instead of the agents considering each element in the environment individually, the comfort function determines the agents' level of

TABLE 7.2 Agent Reaction Levels

Value	Level	Reaction
< 0.1	Comfortable	None
0.1–0.3	Uncomfortable	Calmly moves to more comfortable cell
0.3–0.6	Distressing	Runs from the cell
> 0.6	Painful	Panics and runs quickly from cell

discomfort and the agents respond accordingly by choosing the reaction level that corresponds to their comfort value.

Choosing a Destination

If the agents are not comfortable in their current cell then they must locate and move to a more comfortable cell. Each agent reassesses its situation each time step, by calculating the comfort value for the cell it is standing in or passing through and finding a destination cell based on the comfort of its neighbor cells.

As long as the agent is not comfortable, it will keep reassessing its situation and finding a new destination, which means that agents can change destination while they are moving toward their current destination, if they find a better destination. Also, as the state of the environment is continuously changing, the destination the agent found last cycle may no longer be a comfortable cell. In choosing a destination, the agents evaluate the comfort values of the cells in a neighborhood of a given size and choose the cell with the lowest comfort value.

Key Terms

- *Comfort function* is a weighted sum of the factors that affect the agents' comfort in each cell and includes temperature, fire, pressure, and wetness.
- *Level of reaction* is determined by the comfort function and allows the agent to react with varying degrees of distress, providing for more diverse and interesting behavior.
- *Choosing a destination* occurs when agents are not comfortable in their current cell, so they find a destination in neighboring cells that is more comfortable.

Agent Tuning

Tuning is an important and time-consuming step in the development of any emergent system. The more structured, methodical, and documented the tuning process, the easier and smoother the tuning will be. Three rounds of tuning were conducted to investigate and tune the behavior of the agents in the Active Game World, in terms of efficiency, effectiveness, and observable behavior. The method used to tune the agents is described in the "Process" section.

The first round of tuning aimed to determine the most appropriate neighborhood size that should be used by the agents when choosing a destination. The second round of tuning investigated combining different sized neighborhoods to gain the benefits of both reactive and goal-directed behavior. The third, and final, round of tuning involved combining desirability, in the form of a goal, with the comfort-based reactive behavior of the agents. This section discusses the aims, process, and outcomes of each round of tuning.

KEY TERMS

- *Tuning* involves testing and tweaking parameters and rules until desired behavior is achieved.

Process

Several conditions were investigated in each round and 10 trials with 10 agents were run in each condition. Each round was conducted on a 10-by-10 grid of cells in the Active Game World. The criteria that were used to evaluate the performance of the agents were:

- The number of cycles the Active Game World ran before the agents converged (that is, they located and reached comfortable cells).
- The number of local optima (comfortable cells) on which the agents converged.
- What (if any) strategies or patterns the agents exhibited.

The initial state of each trial was randomly generated, including the position of the agents, the position of rain, and the number and position of explosions. After the trials were started, notes were made in relation to the criteria of efficiency, strategy, and patterns.

The Active Game World was stopped as soon as the agents converged for the first time. At this point, the number of cycles the agents took to converge was noted, as was the number of optima that the agents converged on (an optimum was considered to be a single cell). If the agents failed to converge (which usually occurred if the agents died before finding an optimum) then it was noted that the agents did not converge. A screenshot was taken of the final state of the Active Game World in each trial, showing the state of the system, the number of cycles, and the position of the agents.

Round 1: Determining Neighborhood Size

The aim of the first round of tuning was to determine the best neighborhood size, in terms of agent performance and behavior, that agents should evaluate when choosing a destination (such as where to move to maximize comfort). Four conditions were investigated, including three conditions where the agents used neighborhood sizes of one (n = 1), two (n = 2), and three (n = 3) to choose destinations (see Figure 7.3). The final condition involved the evaluation of the entire grid to find a global optimum.

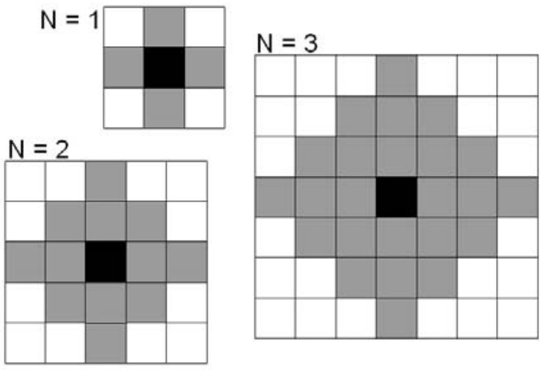


FIGURE 7.3 Neighborhood areas evaluated for n = 1, n = 2, and n = 3.

Outcomes Each of the neighborhood sizes that were evaluated in the first round of tuning had various advantages and drawbacks. The agents with a neighborhood size of one performed the best at avoiding immediate danger. However, their short sight meant that they often ran toward more dangerous situations or became stuck in larger hazards as they were unable to find a way out.

With a neighborhood size of two, the agents were better at choosing safe destinations and appeared more organized, but still expressed the problems associated with short sight. The agents with a neighborhood size of three were exceptional at picking particularly desirable cells and appeared organized as many agents moved to similar locations. However, the problems for these agents were almost the opposite of the previous agents, because they performed the best at choosing a destination but were unable to avoid immediate hazards in getting to their destination. They would often put themselves in great danger (such as run through fire) to get to a safe destination cell.

Moving through hazards to reach a safe goal was a far more severe problem when the agents were just moving toward a global optimum. The agents would move across the whole map to get to their destination, rather than looking for a local optimum or a "good enough" cell in local proximity. The global optimum agents also took significantly more cycles to reach a goal cell than the agents in each of the previous conditions, as the global optimum was continually changing.

Using a global optimum would be more reasonable if the goal cell was particularly desirable, such as fulfilling a mission or going home. The global optimum works well if the agents are in close proximity to the global optimum, but does not work at all if they are far away. Therefore, it would be much more logical for the agents to find a local optimum, as in the previous conditions where they were evaluating their local neighborhood.

There were desirable traits expressed by the agents with short sight (avoiding immediate danger) and the agents with longer sight (finding a local optimum). As a result, a logical solution was to endeavor to combine these two approaches to enable the agents to move to a nearby, safe cell while avoiding hazards along the way. Consequently, the next round of tuning investigated how these two approaches could be combined and whether they would yield an improvement in behavior.

Round 2: Optimizing Agent Navigation

The previous round of tuning demonstrated that neither purely reactive nor goal-directed behavior was desirable for the agents in the Active Game World. The reactive agents were able to avoid immediate danger but often ran into another hazard and the goal-directed agents chose a suitable goal but ran through hazards to reach it. Therefore, the aim of the second round of tuning was to determine if a combination of these two approaches is more effective than either approach individually and what combination of reactive and goal-directed behavior is the most suitable for the Active Game World.

The agents in the second round of tuning combined the reactive and goal-directed behaviors of the agents in the first round by first selecting a goal (the most comfortable cell) from the area around the agent (n=3). Second, the agents then evaluated the cells in their immediate neighborhood (n=1) and chose which way to move, based on the comfort of the immediate cells and how close the immediate cells advanced the agent toward the goal (see Figure 7.4).

There were three conditions evaluated in the second round of tuning:

- *Condition 1:* Weighted the comfort of each cell in the agent's immediate area with equal importance (50%) to the proximity of each cell to the agent's goal cell in the local area (50%).
- *Condition 2:* Weighted the proximity of the immediate cells to the goal cell more importantly (75%) and the comfort of the immediate cells less importantly (25%).

- *Condition 3:* Weighted the proximity of the immediate cells to the goal cell less importantly (25%) and the comfort of the immediate cells more importantly (75%).

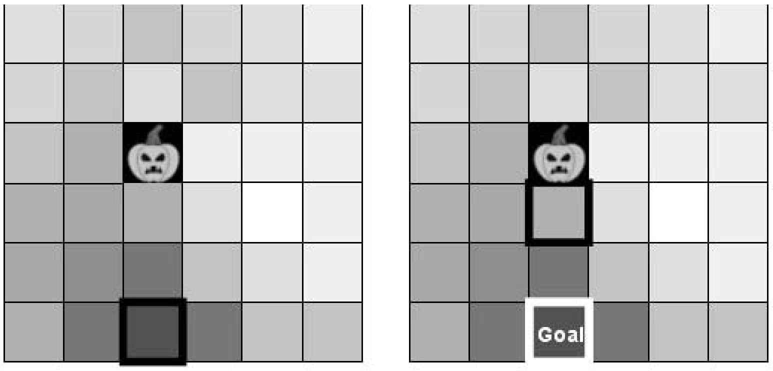


FIGURE 7.4 The agents first selected a goal from their local neighborhood (left), and then a direction to move from their immediate neighborhood (right).

Outcomes The agents in the second round of tuning displayed definite advantages over the agents in the first round. The agents in the evenly weighted and goal-directed conditions appeared far more intelligent, because they moved toward a goal rather than running back and forth randomly. These agents appeared more realistic, because they moved around hazards on the way to their goal rather than just running in a straight line, which made the agents in the previous experiment appear flat and synthetic.

The agents in the evenly weighted condition displayed more depth as they did not always react in the same way, sometimes they would appear organized and at other times they would appear more independent, with their behavior being heavily dependent on the current situation. The agents in the evenly weighted condition took the least amount of time to converge on safe cells.

The agents in the goal-directed condition behaved in a similar way to the agents in the evenly weighted condition, but became stuck more often and still ran through hazards. The agents in the reactive condition had the least desirable behavior as they often appeared to move randomly, did not appear organized, and often became stuck.

Therefore, the second round of tuning suggested that the most suitable combination of reactive and goal-directed behavior for the agents in the Active Game World is approximately equal, where it is more desirable to err on the side of goal-directed than on reactive behavior.

Round 3: Combining Comfort and Desire

The first and second rounds of tuning gave rise to agents that efficiently, intelligently, and realistically react to the environment by moving from danger to safety. However, in a computer game situation, it is likely that agents will have greater goals or desires that they need to fulfill, apart from simply surviving and reacting sensibly to the environment. For example, marines in a strategy game might be on a mission to kill the enemy in a particular cell or a villager in a role-playing game might want to stay near its house or shop.

Drawing on the notion of "desirability" values from influence maps, goal areas could be given high desirability values for the agents. Additionally, desirability values could then be propagated out to surrounding areas to indicate that these areas are more desirable as they are near the goal. Therefore, the aim of the third round of tuning was to combine the desire to reach a greater goal with the agents' current behavior of reacting to the environment and avoiding hazards.

The third round of tuning combined an influence map to propagate the desirability of the cells with the cellular automata to determine the comfort of the cells. The goal of the third round of tuning was to determine the combination of desire and comfort that would give the agents the best behavior, in terms of avoiding hazards and reaching their goal.

The scenario for the third round of tuning was that 10 agents have been given the order of getting to a single goal (for example, marines sent to attack an enemy tank). The method used in the third round was to propagate the desire out from the goal position (see Figure 7.5). I used a propagation constant of 0.7 (that is, the de-

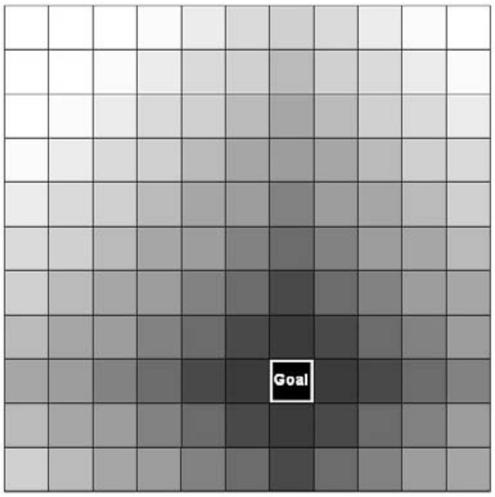


FIGURE 7.5 Propagation of desirability of the goal cell in an influence map.

sirability value is multiplied by 0.7 for each step out from the goal). This value was chosen because it allows the influence to spread over the entire map of 10-by-10 cells, with a high concentration of desirability near the goal and low levels away from the goal.

Figure 7.6 shows the desirability on a map with a single goal and desirability combined with comfort (darker is less desirable). Next, the agents calculated the comfort values for their local neighborhood (n = 3). Subsequently, the agents selected the best cell in this neighborhood, based on the desirability value combined with the comfort value of each cell, which became their goal.

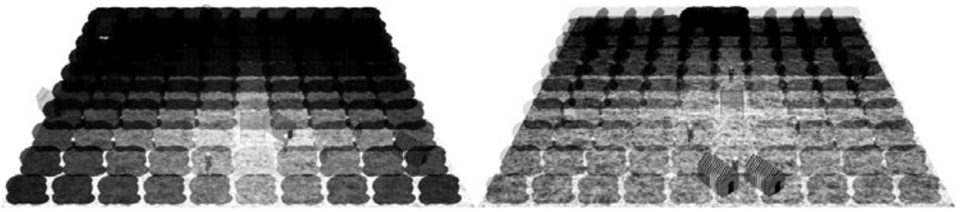


FIGURE 7.6 Desirability on a 10-by-10 map with a single goal and propagation constant of 0.7 (left) and combined with comfort values (right).

The three conditions that were investigated in the third round of tuning were designed to test different influences of comfort and desirability on the agent's choice. The three conditions were evenly weighted (50% desirability—50% comfort), goal-oriented (75% desirability—25% comfort), and self-preserving (25% desirability—75% comfort).

After the agent has chosen the best cell in its local neighborhood, based on comfort and desirability, it then chose which cell to move to in its immediate neighborhood (as in Round 2). The best cell in the immediate neighborhood was chosen based on its comfort value (50%) and its closeness to the chosen cell in the local neighborhood (50%), which is the condition that was selected in the second experiment. Figure 7.7 illustrates the process of choosing a local goal based on comfort and desirability, and then a movement direction based on comfort and proximity to the local goal.

Outcomes The first three conditions demonstrated that an equal weighting of desirability and comfort gave the agents the most acceptable observable behavior, in terms of organization, avoiding hazards, and navigating the environment realistically and intelligently. The agents converged reasonably efficiently, but only about half of the agents found the goal as they opted for comfort over the goal.

When the weighting was tipped toward comfort or desirability, the agents' behavior appeared random, less organized, and less intelligent. The goal-directed

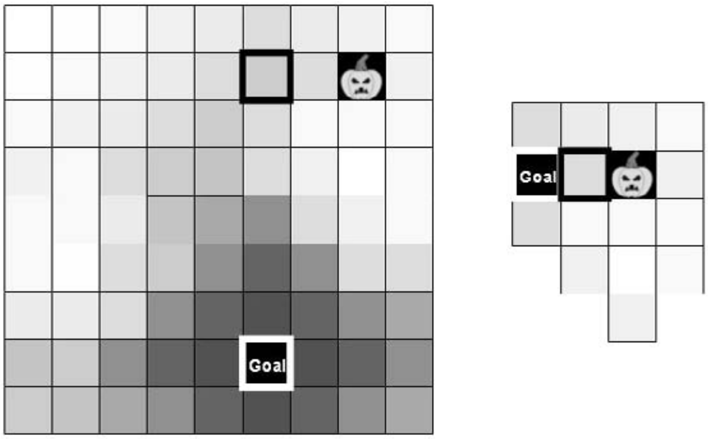


FIGURE 7.7 The agents choose a local goal based on comfort and desirability (left), and then a movement direction based on comfort and proximity to the local goal (right).

agents converged in a reasonable period of time, but only about half the agents found the goal. The self-preserving agents required significantly more cycles to converge than the agents in the previous conditions. There was no significant difference between the number of agents that found the goal in the self-preserving condition and in the previous conditions.

Propagation Constant The agents in each of the first three conditions were not particularly successful at finding the goal. Therefore, a fourth trial was run with equal weighting to optimize behavior, but with a greater propagation constant to increase desirability values around the goal.

Increasing the propagation constant (0.8) resulted in greater differentiation between cells on the influence map and further improvements in observable agent behavior (see Figure 7.8). The most noticeable change with a propagation constant of 0.8 was the improvement in the agents' behavior, in terms of organization, intelligence, and rational behavior. The agents were consistently able to move in organized and intelligent ways, exhibiting interesting and rich behavior.

In one situation, the agents were moving toward a goal that was blocked by rain and they waited for the rain to pass before moving toward the goal, rather than running through the rain or getting stuck. The increased differentiation between cells on the influence map provided the agents with a clear view of the best way to navigate through the environment.

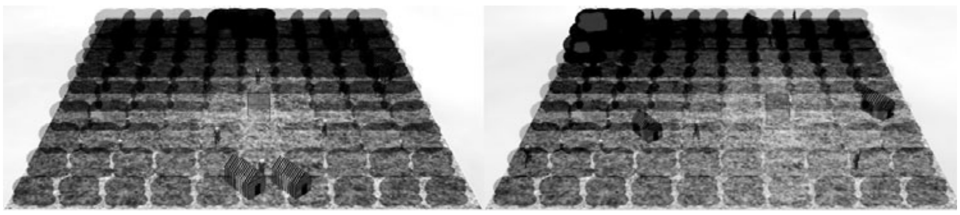


FIGURE 7.8 Increasing the propagation constant from 0.7 (left) to 0.8 (right) resulted in greater differentiation between cells on the influence map and further improvements in observable agent behavior.

KEY TERMS

- *Propagation constant* determines how much the influence in an influence map will decrease for each step out from the origin.

Multiple Goals A fifth and final condition was tested to determine how the agents would perform with multiple goals instead of one. The agents in the multiple goals condition were much better at finding the goals, but did not appear as intelligent or realistic as the agents in the previous condition. The agent were more likely to find the goals in this condition as there were more goals, but mostly because the desirability from the goals was cumulative, allowing more cells to have higher values and the influence to propagate further.

Outcomes of Agent Tuning

The outcome of the first two rounds of tuning was a model for agents that dynamically respond to the environment in an intelligent and realistic way, based on concepts from cellular automata and influence maps. The outcome of the third round was an extension of this model that also integrates goal-directed behavior to enable the agents to respond to the environment while pursuing a goal.

The tuning process produced an agent model that successfully integrates goal-directed behavior (based on agent desires) with situation awareness (based on comfort), which enabled the agents to both react to the environment in an intelligent, realistic, and organized way, while simultaneously satisfying their desire to reach a goal.

An advantage of the model developed through this process is extensibility, in that it can be extended to incorporate any aspects in the game world that are relevant to the agents' behavior (for example, other agents, terrain, and events). It would be possible to incorporate other models for agent behavior, such as flocking, so that the agents can take into consideration the movement of other agents around them.

The simplicity and flexibility of this model means that it can be used to govern the behavior of almost any agent in any circumstance. The agent design used in the Active Game World creates agents that can dynamically react to the changing situation of their environment, as well as an intelligent pathfinding algorithm that allows agents to find a safe path to a goal, based on aspects of their environment.

Reactive Agent Algorithm

This section summarizes the agent model algorithm and code that was developed and tuned through the process discussed in the previous sections. First, the desirability of the goal is propagated by iterating through each cell on the map and setting the propagated influence value in each cell, depending on its distance from the goal, as well as the goal's desirability and the propagation constant:

```
float CalcDesire (cells goal)
{
       // distance = city block distance of x, y from goal
       const float distance =  abs(x – goal.x) + abs(y – goal.y);

       // Desirability = goal's desirability *
       // Power (propagation constant, distance from goal)
       float desire = goal.desire * Math.Pow (0.7, distance);

       return desire;
}
```

Comfort

In each cycle, each agent calculates the comfort value for each cell in its local neighborhood (n=3). The comfort value is a weighted sum of the environmental factors in the cellular automata, including temperature, pressure, fire, and wetness:

```
float GetComfort ()
{
       // fire > temperature > pressure > wetness

       // Min(((fire*W1) + (temp*W2)
       // + (pressure*W3) + (wetness*W4)), 1)

       const float comfort = (temperature * TEMPCONST)
              + (fire * FIRECONST)
              + (pressure * PRESSCONST)
              + (wetness * WETCONST);
```

```
        if (comfort > 1.0)
        {
               comfort = 1.0;
        }

        return comfort;
}
```

Local Goal

Subsequently, the agent finds the optimal cell in its local neighborhood. The "goodness" of each cell in the local neighborhood is the sum of 50 percent of the comfort of that cell and 50 percent of the desirability of that cell.

```
void GoalDest (float comfort, float speed, int n)
{
       int destX, destY, minX, minY, maxX, maxY;
       float thisComfort;
       cont int empty = -1;

       minX = std::max (position.x – n, 0);
       maxX = std::min(position.x + n, numCells);
       minY = std::max(position.y – n, 0);
       maxY = std::min(position.y + n, numCells);

       // for each cell in local neighborhood (n=3)
       for (int i = minX, i < maxX, i++)
       {
              for (int j = minY, j < maxY, j++)
              {
                    // only calculate if not previously calculated
                    // and store
                    if (cells[i][j].comfort == empty)
                           cells[i][j].comfort
                           = cells[i][j].GetComfort();

                    thisComfort = cells[i][j].comfort;

                    // add in desire from influence map
                    // Goodness of local cell
                    // = 50% comfort + 50% desirability
                    thisComfort =
                           (cells[i][j].comfort * 0.5)
                           + ((1 – cells[i][j].desire) * 0.5);
                    // set destination to the most comfortable cell
                    // in neighborhood
                    if (thisComfort < comfort)
                    {
                          comfort = thisComfort;
                          destX = x;
                          destY = y;
                    }
              }
        }
        // find destination within immediate neighborhood (n=1)
        ImmDest(comfort, speed, 1, destX, destY);
   }
```

Immediate Goal

The cell that the agent identifies as the optimal cell in its local neighborhood becomes its goal. Next, the agent assesses its immediate neighborhood (n=1) to find the most optimal cell. The "goodness" of each cell in its immediate neighborhood is the sum of 50 percent of the comfort of that cell and 50 percent of the proximity of that cell to the agent's goal in the local neighborhood.

```
   void ImmDest (float comfort, float speed, int n, int goalX, int goalY)
   {
         int minX, maxX, minY, maxY;
         float minComfort;

         minX = std::max (position.x – n, 0);
         maxX = std::min(position.x + n, numCells);
         minY = std::max(position.y – n, 0);
         maxY = std::min(position.y + n, numCells);

         // Goodness of immediate cell = 50% comfort
         // + 50% proximity to local goal
         minComfort = (comfort * 0.5) + ((abs(goalX – position.x)
               + abs(goalY - position.y)) / 8.0f);

         // for each cell in local neighborhood (n=1)
         for (int i = minX, i < maxX, i++)
         {
               for (int j = minY, j < maxY, j++)
               {
                     // 50% weighting = divide by 8
                     goal_dist = (abs(goalX - i) + abs(goalY - j)) / 8.0f;
```

```
                    // 50% weighting = multiply by 0.5
                    thisComfort = (cells[i][j].comfort * 0.5) + goal_dist;

                    // set immediate destination to most comfortable cell
                    // in neighborhood
                    if (thisComfort < minComfort){
                          minComfort = thisComfort;
                          destX = x;
                          destY = y;
                    }
             }
       }
       // move to destination at speed
       Move(destX, destY, speed);
}
```

Reaction

After the agent determines its destination cell in the immediate neighborhood, it moves toward that cell at a pace dependent on its current comfort. Each cycle, the agent re-evaluates its local and immediate neighborhood and updates its goal and destination.

```
void React ()
{
       float comfort, speed;
       int n;

       comfort = GetComfort(x, y);

       // neighborhood size = 3
       n = 3;

       // if comfortable — stand still
       if (comfort < 0.1)
       {
              StopMove ();
       }

       // if uncomfortable — move
       else if (comfort < 0.3)
       {
              // set speed — 1, animation walk
              speed = 1.0f;
```

```
                // move – to dest
                GoalDest(comfort, speed, n);
        }
        // if distressed – move quickly
        else if (comfort < 0.6)
        {
                // set speed – 2, animation run
                speed = 2.0f;
                // move – to dest
                GoalDest(comfort, speed, n);
        }
        else
        {
                // set speed – 3, animation run
                speed = 3.0f;
                // move – to dest
                GoalDest(comfort, speed, n);
        }
}
```

Observable Behavior

Introducing entities that have a choice of how to react to the situation amplified the variation and unpredictability of the Active Game World. Rather than the simple physical exchange that existed in Chapter 6, a new level of depth was introduced to the Active Game World by agents that actively change their own state.

Agents in the environment are not confined to the state of their current cell. Instead, they are free to react to the changing environment in ways that optimize values that are important to them, such as comfort and desirability. In doing so, the agents actively change the state of the environment, because they carry effects between cells in new and dynamic ways. For example, if a tank catches on fire and reacts by rolling into a group of trees, those trees will, in turn, catch on fire, whereas the outcome would have been different without the active role of the agent.

Furthermore, because the agents have the same set of physical and structural properties as objects and cells in the Active Game World, different agents are affected in varying ways by the same circumstances and subsequently react in different ways in these scenarios. This section examines four scenarios and discusses the contrasting outcomes and considerations for agents, objects, and environments in scripted and emergent systems with respect to these scenarios.

The four scenarios used for evaluation and tuning are (1) heat and fire, (2) fluid and wetness, (3) pressure and explosions, and the (4) integrated system, including each of the previous components. Demos for each of these scenarios in the Active Game World can be found on the CD-ROM.

Scenario 1: Heat and Fire

The first scenario presents a case study for heat and fire, in which a fire is started in a military base that contains a variety of buildings (for example, metal bunkers and wooden barracks) and agents (for example, tanks or people). As discussed in Chapter 6, the interactions of fire with each of the different types of objects and terrain are emergent in the Active Game World (see Figure 7.9). Similarly, the interactions of fire with the different types of agents in the scenario are emergent.

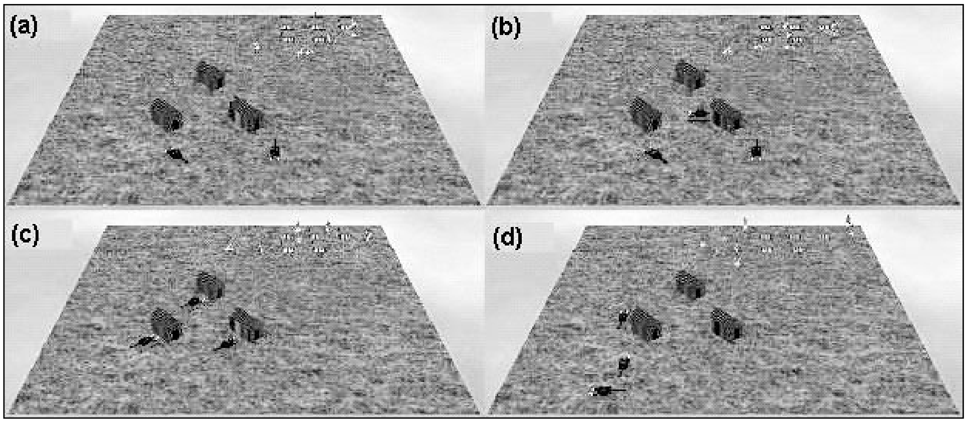


FIGURE 7.9 A fire is started in a military base (a) and spreads throughout the base (b–d), affecting different agents (tanks, people) and buildings (bunkers, barracks) in different ways.

The observable behavior of the fire is emergent, due to the high- and low-level properties of the entities in the scenario, combined with the rules for the interactions of heat and fire. Therefore, the observable behavior of fire in the heat and fire scenario depends on the position, composition, number, and variety of objects, agents, and terrain in the scenario.

The behavior and effects of the fire emerge and change as a function of the situation. The agents also react dynamically to the changing situation and in varying ways, depending on their low-level properties (a tank has a higher threshold for discomfort than a human) and high-level properties (a tank can move faster than a human and a boat cannot move on land). Finally, the ways that the agents react to the heat and fire scenario feed back to actively change the state of the scenario. For example, a tank that is on fire will propagate the fire as the tank moves into cells that the fire might not have reached otherwise.

It would be difficult and time-consuming to script the observable behavior of the fire, the effects of the fire on the agents, objects, and terrain, as well as the resulting reactions of the agents to the situation, especially with any level of realism.

As discussed in Chapter 6, it is prohibitively complex to script the interactions of fire with game objects and environments to the level of complexity implemented in the Active Game World. Each attribute of the objects and terrain that need to be considered add a new level of complexity to the problem. Furthermore, the layout of the objects, agents, and terrain in the heat and fire scenario magnifies the problem. Finally, when the reaction of the agents to the fire needs to be considered (even without the agents' actions feeding back into the scenario), the problem becomes impossible.

Consequently, it is likely that scripting fire in a game scenario would result in flat, unrealistic, uniform fire, or spending tedious hours specifically scripting the multitude of combinations that are possible, which would fall far short of what is simulated in an emergent system due to time and logistical constraints.

Scenario 2: Fluid and Wetness

The second scenario presents a case study for fluid and wetness, in which rain falls on a hillside and runs down the hill and floods the village in the valley below. The village contains different types of buildings, villagers, vehicles, and a boat. As discussed in Chapter 6, the interactions of water with different types of objects, agents, and terrain are emergent in the Active Game World (see Figure 7.10), as well as interactions of water with different terrain contours.

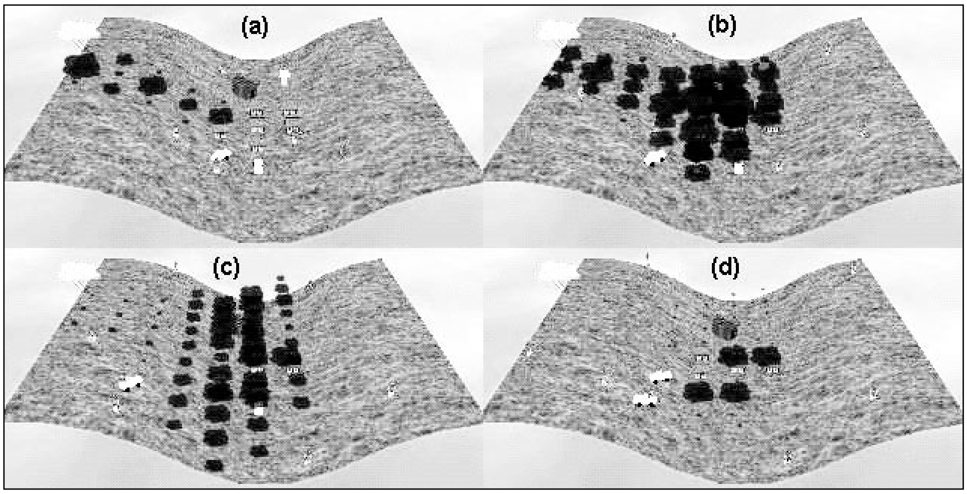


FIGURE 7.10 Rain runs down a hillside (a) and floods the village in the valley below (b–c) and dries out over time (d). The flood affects and interacts with the agents (trucks, people, and a boat) in different ways.

The water from the rain runs down the hill and floods the valley, due to the emergent interactions of the water with the contours of the hill and valley. Some objects and agents (such as houses and cars) are filled with water, due to their high-level properties and other agents can float on the water (such as boats), due to their ability to move on water.

The different objects, agents, and terrain become wet from the water to varying degrees, depending on their composition. For example, a wooden house absorbs more water than a metal shack, making it harder to ignite and more water-damaged. Again, small changes in the fluid and wetness scenario can give rise to significantly different observable behavior. For example, changing the contours of the hill could mean that the village will not flood. Also, different agents respond to the water in different ways, because a flood is far more dangerous to a villager than it is to a boat.

Scripting the behavior of the fluid and wetness scenario specifically would be time-consuming, difficult, and impractical. Changes to the contours of the terrain have a significant effect on the series of events in the scenario, by determining whether the water will flood the valley, whether it will pass through the valley, and the magnitude of the flood. Subsequently, the various entities in the valley are affected by the flood water in varying ways, which would be prohibitively expensive to script specifically. Furthermore, the agents add another layer of complexity to the problem.

Again, there are two possible approaches to scripting the fluid and wetness scenario. The first is to have a limited number of pre-scripted scenarios that can take place and the second is to attempt to script the scenario with some level of realism and flexibility, which is prohibitively time-consuming and complex.

Scenario 3: Pressure and Explosions

The third scenario presents a case study for pressure and explosions, in which a bomb explodes in a military base that contains different types of buildings (such as a metal bunker or wooden barracks) and agents (such as people and tanks). As discussed in Chapter 6, the observable effects of the interactions of the explosion (that is, pressure) with the objects, agents, and terrain in the pressure and explosions scenario are emergent in the Active Game World (see Figure 7.11), as are the secondary effects of the explosion (for example, fire started as a result of heat generated by explosions).

The objects, agents, and terrain are affected in varying ways by the explosion, depending on the strength of their composing material, their high-level properties (for example, objects with volume can be filled with pressure), size, and number of explosions, as well as the position, size, number, and type of entities in the scenario. The result is a dynamic, emergent chain of interactions that arise from the rules for interaction, properties of the entities, and the initial and evolving state of the scenario.

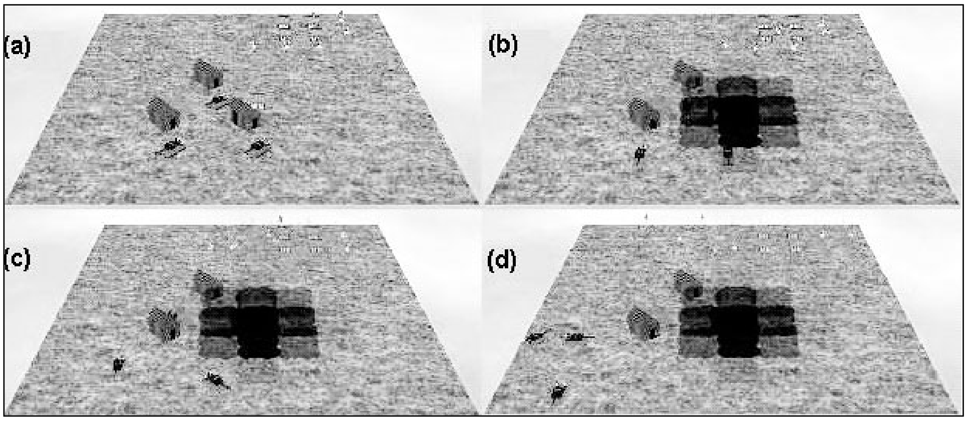


FIGURE 7.11 A bomb explodes in a military base (b), causing a fire (c) that spreads through the base (d).

Furthermore, the actions of the agents again add an extra layer of complexity to the Active Game World. The agents react in varying ways to the pressure and explosions scenario and as a function of the changing state of the scenario. In doing so, they actively change the system state and propagate effects in ways that might not have occurred naturally.

The same problems exist for specifically scripting the pressure and explosions scenario, as discussed in the previous scenarios and in Chapter 6. The Active Game World takes into consideration many different factors, such as the position and magnitude of the explosion, proximity of game objects and agents to the explosion, high- and low-level properties of objects, agents, and terrain that result in varied effects from the explosion, and so on.

The actions and reactions of the agents, which vary by type and situation of agents, add another layer of complexity to the problem. Therefore, the pressure and explosions scenario could be scripted to be executed in a prescribed way or the system could attempt to consider various attributes of the situation and run a precoded script. In either case, the script is preset, rigid, and cannot be interacted with by the players or changed as a result of the situation.

Scenario 4: Integrated System: Heat, Fluid, and Pressure

The fourth scenario is a combination of the previous three scenarios. Rain falls on a hillside and runs down the hill, flooding the village in the valley below. Subsequently, the flood washes away and an unrelated explosion occurs in the village.

In the Active Game World (see Figure 7.12), the initial flood accumulates in the valley, as a result of the interactions of the water with the contours of the hill. The flood water then interacts with the buildings, vehicles, and people in the village in varying ways, depending on their low and high-level properties. The flood washes away due to the contours of the landscape and dries out over time.

When the explosion occurs, the entities in the village are affected by the high pressure in different ways, depending on their composition. Under the right conditions, a fire starts in the village as a result of the heat released by the explosion. The fire then spreads through the village, affecting different buildings and agents in different ways, depending on their composition, their wetness from the flood, position, size, and various other factors.

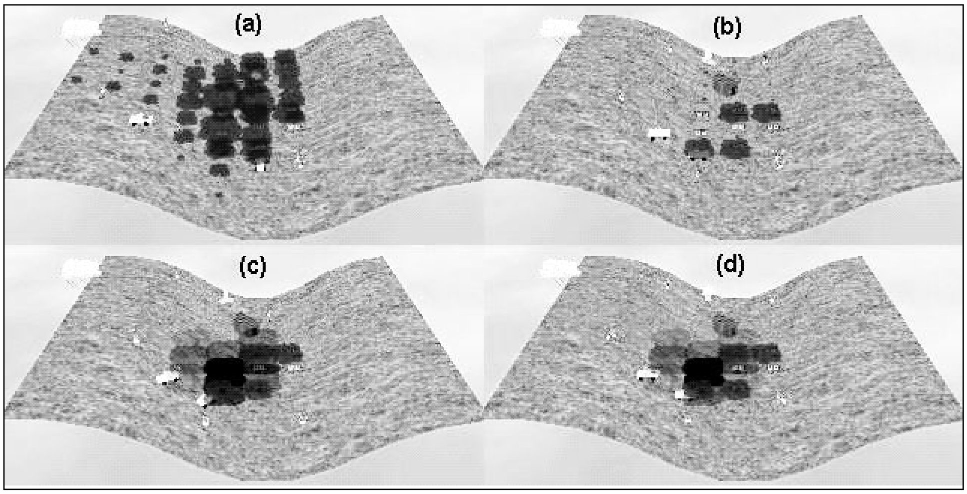


FIGURE 7.12 Rain runs down a hillside and floods a village (a) and dries out over time (b), followed by an explosion in the village (c) and a fire caused by the explosion (d).

As discussed in Chapter 6, the complexity, power, and flexibility of the Active Game World becomes most apparent in the integrated scenario. The simple, low-level rules of each system (pressure, heat, and water) interact to give rise to interesting, rich, and complex high-level, observable behavior.

The low-level properties of the agents, objects, and terrain, the high-level properties of agents and objects, and the contours of the terrain combine with the system rules to create a living, evolving, and complex system. Changes to the initial and continuing state of the integrated scenario give rise to different outcomes and observable behavior of the system.

Each event that occurs in the Active Game World (rain, flood, explosion, fire) is not the result of pre-planning and scripting. Instead, each event occurs because the conditions of the integrated scenario are right for each of these events to happen, as would be the case in the real world.

The rules of how things work in the Active Game World (such as hot things burn, and water flows downhill) combine and interact to mould the series of events that occur in an integrated scenario. With the addition of the agents (entities that have a choice of how to react in a given situation), the complexity of the system is deepened further. As the system state changes, the agents choose how to react to their environment (affected by their high- and low-level properties and goals), which in turn feeds back into the state of the system.

The interactions and actions of the agents give another level of complexity to the observable behavior of the Active Game World, bringing the environment to life with thinking, responding, interacting entities. A script to run the events described in the integrated scenario would need to be highly specific to the scenario. Any change to the initial setup or any change to effects throughout the scenario would require multiple changes to be made to the script. It would not be possible to achieve the level of detail, interactivity, and dynamic behavior as is described in the integrated scenario in a specifically coded system.

Emergent Individual Behavior

This section discussed the design, implementation, and tuning of reactive game agents in the Active Game World. These agents use influence maps, in conjunction with the cellular automata in the Active Game World, to dynamically respond to the environment. The resulting agent model works by first populating the agents' influence map with values based on the comfort of the cells and the agents' desire to move to a goal (if they have one). Subsequently, the agents find the best cell in the influence map in their local neighborhood and decide which way to move (north, south, east, or west), based on the comfort of the cell in each direction and the proximity of this cell to their local optima.

Three structured rounds of tuning were conducted to determine the design that would achieve the best behavior for the agents. The first round aimed to determine the appropriate neighborhood size that agents should evaluate when choosing a destination. The section concluded that it would be desirable to combine the ability of agents with larger neighborhoods to find local optima with the ability of agents with smaller neighborhoods to avoid immediate threats.

The second tuning session aimed to find the most appropriate combination of immediate area (reactive) and greater area (goal) evaluation. From the second round of tuning, we found that an equal weighing of reactive and goal-directed behavior gave rise to the most appropriate agent behavior in the Active Game World.

The third round of tuning aimed to determine how well the agents perform with a goal, because many agents in games are required to do more than react to the environment. From the third round of tuning, it was demonstrated that increased differentiation between cells on the influence map provided the agents with a clear view of the best way to navigate through the environment, which had an important impact on the success of the agent's behavior.

The Active Game World, including cells, objects, and agents, exhibited the same advantages over scripted systems as were discussed in Chapter 6, as well as a new level of complexity and emergence added by the autonomous agents. As discussed in Chapter 6, the agents, objects, and cells have low-level properties, related to their physical composition, which determine how they will interact with heat, pressure, and water. Also, the agents and objects have high-level properties, related to their physical structure, which further constrain the ways in which they are able to interact with heat, pressure, and water. Therefore, the interactions that occur in any given scenario are dependent on, and emerge as a function of, a variety of factors, including number, type, and position of entities in the environment, terrain, and external effects (for example, rain and wind).

Reactive agents can further the emergent behavior and gameplay discussed in Chapter 6 by adding a new level of complexity to the game world. As the agents are able to choose how to react to the environment, they are able to actively change the state of the world in ways that might not have occurred without their intervention. Furthermore, the differences between individual and types of agents, such as composition, structure, goals, personality, and so on, means that different agents will choose to react in different ways in the same situation.

The result of each component of the system (objects, agents, and cells) working together is a complex, rich, and living world that provides a suitable environment for interesting and emergent gameplay to take place.

The reactive agent model is also very extensible, in that it can be extended to incorporate any aspects in the game world that are relevant to the agents' behavior (for example, other agents, terrain, and events). The simplicity and flexibility of this model means that it can be used to govern the behavior of almost any agent in any circumstance.

The model can be used for agents that dynamically react to the changing situation of their environment, as well as an intelligent pathfinding algorithm that allows agents to find a safe path to a goal, based on aspects of their environment. With the use of the influence map structure, the model can be easily adapted to be used for different decisions with different variables and weightings. With the cellular automata feeding into a layer of the influence map, the environmental values from the cellular automata can be used for other decisions in which environmental factors are relevant, such as strategic decision-making (for example, using flaming catapults upwind of an opponent).

The reactive agent model described in this chapter provides a possible solution for incorporating agents that appear intelligent to the players by reacting sensibly to the game environment, into game worlds. As described, reactive agents can be incorporated into game worlds by giving the agents a measure of comfort in the current situation (via cellular automata or other means), as well as a map for deciding where they might move to maximize their comfort.

As this design closely resembles an influence map, it is also possible to integrate goal-directed behavior, and potentially personality, group movement, and various other behaviors into the agent model. Within the Active Game World, the agents are extensions of the existing objects and are therefore subject to the same rules of interacting in the environment.

Whereas current agents in games do not demonstrate an awareness of their situation or react appropriately to events in their immediate surroundings, the reactive agents maintain a model of the comfort of their environment and react according to the changing state of their situation. The reactive agent model allows agents to dynamically react to the changing situation of their environment and to intelligently find a path to a goal, increasing their visible level of intelligent, realistic, and responsive behavior.

Additional Reading

For further information on using cellular automata and influence maps for reactive game agents:

- Sweetser, P. (2006) Environmental Awareness in Game Agents. *AI Game Programming Wisdom 3*, Hingham, MA: Charles River Media, pp. 457–468.
- Sweetser, P., and Wiles, J. (2005) Combining Influence Maps and Cellular Automata for Reactive Game Agents. 6th International Conference on Intelligent Data Engineering and Automated Learning, *Lecture Notes in Computer Science*, 3578, pp. 524–531.

For further information on movement of individual characters:

- Reynolds, C. W. (1999) Steering Behaviors for Autonomous Characters. *Proceedings of Game Developers Conference 1999*, San Francisco, CA: Miller Freeman Game Group, pp. 763–782.

Group

Many games have groups of agents that must be able to interact, coordinate, and cooperate. This is particularly important in team-based games, such as strategy games and sports games. When there are two or more sides fighting or competing, the agents must cooperate in an organized way to have any chance of success. The two most important group actions in games are group movement and tactics.

Emergence has a lot of potential to improve group behavior, with a focus on self-organization, rather than top-down orchestration.

Group Movement

Coordinated and fluid group movement can be achieved with a bottom-up, agent-based approach. Many movies and games have used flocking as a steering behavior for groups, schools, or herds of animals, people, and monsters. Games that have successfully used flocking to simulate the group behaviors of monsters and animals include *Half-Life*, *Theme Hospital*, *Unreal*, and *Enemy Nations*.

Half-Life uses flocking to simulate the squad behavior of the marines, who run for reinforcements when wounded, lob grenades from a distance, and attack the players with dynamic group tactics. *Theme Hospital* uses flocking to simulate the hustle-and-bustle of patients, doctors, and staff in a hospital. *Unreal* uses flocking for many of the monsters, as well as other creatures, such as birds and fish. *Enemy Nations* uses a modified flocking algorithm to control unit formations and movement across a 3D environment.

Movies have used flocking to simulate crowds of extras and flocks of animals. For example, the movie *Batman Returns* made use of flocking algorithms to simulate bat swarms and penguin flocks.

Flocking in Games

Flocking can be used in games for unit motion and to create realistic environments the players can explore. In a real-time strategy or role-playing game, groups of animals can be made to wander the terrain more realistically than with simple scripting. Similarly, flocking can be used for realistic unit formations or crowd behaviors.

For example, groups of archers or swordsmen can be made to move realistically across bridges or around obstacles, such as boulders. Alternatively, in first-person shooter games, monsters can wander the dungeons in a more believable fashion, avoiding players and waiting until their flock grows large enough to launch an attack.

Flocking is currently used in games where there are groups of animals or monsters that need to simulate lifelike flock behavior. It is a relatively simple algorithm and composes only a small component of a game engine. However, flocking makes a significant contribution to games by making an attack by a group of monsters or marines realistic and coordinated. It therefore adds to the suspension of disbelief of the game and is ideal for role-playing or first-person shooter games that include flocks, swarms, or herds.

Agent-Based Steering

Agent-based steering behaviors are based on individual agents having a few simple rules to guide their movement in relation to their environment and other agents.

ON THE CD

These types of steering behaviors can also be extended to include goals, personality, threats, and other relevant factors. The game *Halloween Wars* on the demo CD-ROM uses agent-based steering behaviors to create emergent group movement and tactics.

As discussed in Chapter 5, flocking is an artificial life technique for simulating the natural behavior of groups of entities that moves in herds, flocks, or swarms. Flocking was devised as an alternative to scripting the paths of each entity individually, which was tedious, error-prone, and hard to edit, especially for a large number of objects.

In flocking, the aggregate motion of the simulated flock is created by a distributed behavioral model like that in a natural flock. The generic simulated flocking creatures are called boids. Each boid in the flock is an individual that navigates according to its local perception of its environment, the laws of physics that govern this environment, and a set of programmed behaviors. Flocking assumes that a flock is simply the result of the interaction between the behaviors of individual boids.

Steering Behaviors

The boids in the *Halloween Wars* flocking demo are based on the four steering behaviors discussed in Chapter 5: separation, alignment, cohesion, and avoidance. These rules describe how an individual boid maneuvers based on the positions and velocities of its nearby flockmates.

```
PROCEDURE flocking ( )
       sum of mass = 0
       perceived velocity = 0

       FOR EACH boid
              sum of mass += boid.position
              perceived velocity += boid.velocity
       END

       center of mass = sum of mass / number of boids
       average velocity = perceived velocity / number of boids

       FOR EACH boid
              boid.position += boid.separation() + boid.alignment()
                     + boid.cohesion() + boid.avoidance()
       END
END PROCEDURE
```

Separation In separation, each member of a flock tries to keep a minimum distance from its neighboring flockmates. It helps to prevent boids from crowding together, while ensuring a lifelike closeness. Each boid of a flock tests how close it is to its nearby flockmates and then adjusts its steering to obtain the desired distance.

```
FUNCTION separation ()
       separation = 0
       FOR EACH boid
              IF boid is not equal to this boid THEN
                     // minimum separation = 10
                     IF abs(position - boid.position) < 10 THEN
                            // double the separation from boid
                            separation -= (position - boid.position)
                     END
              END
       END
       RETURN separation
END FUNCTION
```

Alignment Alignment involves each member attempting to go in the same direction as its neighbors. Each boid looks at nearby flockmates and adjusts its steering and speed to match the average steering and speed of its neighbors.

```
FUNCTION alignment ()
       // move 10% closer to average velocity
       RETURN (average velocity – velocity) / 10
END FUNCTION
```

Cohesion In cohesion, each member tries to get as close as possible to its neighbors. Each boid examines its neighbors, averages their positions and adjusts its steering to match.

```
FUNCTION cohesion ()
       // move 1% closer to center of mass
       RETURN (centre of mass – position) / 100
END FUNCTION
```

Avoidance Avoidance allows a flock to react to predators and obstacles. Avoidance makes each member keep a certain distance from obstacles or members in other flocks, such as predators. This provides a boid with the ability to steer away from obstacles and avoid collisions. Each boid looks ahead some distance, determines whether a collision with some object is likely, and adjusts its steering accordingly.

```
FUNCTION avoidance ()
        avoidance = 0
        FOR EACH obstacle
                // minimum separation = 20
                IF abs(position - obstacle.position) < 20 THEN
                        // double the separation from boid
                        avoidance -= (position - obstacle.position)
                END
        END
        RETURN avoidance
END FUNCTION
```

Simulated Flocking

Flocking is a stateless algorithm, because no information is maintained from update to update. Each member in the flock revaluates its environment at every update cycle, which reduces the memory requirements and allows the flock to be purely reactive, responding to the changing environment in real-time.

Each boid in the flock has direct access to the whole scene's geometric description. However, flocking requires the boid to react to flockmates in its local neighborhood, which is characterized by a distance from the center of the boid and an angle from the boid's direction of flight. The flockmates that are outside this local neighborhood are ignored. This neighborhood is the region in which flockmates influence a boid's steering.

There are several constraints that restrict how boids can move and react, namely perception range, velocity, and environment. The perception range is the distance that the boid can look around to detect flockmates, obstacles, and enemies. A flock with a larger perception range is more organized and better at avoiding enemies and obstacles. Whereas a smaller range results in a more erratic flock with groups of boids splitting off more often. The velocity refers to the boids' ability to keep up with their flockmates by how fast they can move and turn. The flock's environment can also impose constraints, such as a size limit or many obstacles or predators.

In simulated flocking, the boids initially move together rapidly to form the flock. As they are flocking, the boids at the edge of the flock either increase or decrease their flying speed to maintain the integrity of the flock. Each boid in the flock makes minor adjustments to its heading as the flock winds its way around. The boids fluidly flock around obstacles in their path, which may temporarily divide the flock, but they are soon reunited. Each boid only perceives its neighbors and their actions and reacts accordingly. However, the collective movement of the boids closely resembles real flocking, even though there are no rules that dictate the behavior of the flock as a whole.

Additional Reading

For further information on flocking in games:

- Reynolds, C. (1987) Flocks, Herds, and Schools: A Distributed Behavioral Model. *Computer Graphics* 21 (4), pp. 25–34.
- Reynolds, C. (2006) Big Fast Crowds on PS3. *Proceedings of the 2006 ACM SIGGRAPGH Symposium on Videogames*, pp. 113–121.
- Woodcock, S. (2000) Flocking: A Simple Technique for Simulating Group Behavior. *Game Programming Gems*. Hingham, MA: Charles River Media, pp. 305–318.
- Woodcock, S. (2001) Flocking with Teeth: Predators and Prey. *Game Programming Gems 2*. Hingham, MA: Charles River Media, pp. 330–336.

Key Terms

- *Group movement* requires agents to move in coordination with members of a group or team.
- *Flocking* is an artificial life technique for simulating the natural behavior of groups of entities that moves in herds, flocks, or swarms.
- *Agent-based steering behaviors* are based on individual agents having a few simple rules to guide their movement in relation to their environment and other agents.
- *Separation* involves each member of a flock trying to keep a minimum distance from its neighboring flockmates.
- *Alignment* involves each member attempting to go in the same direction as its neighbors.
- *Cohesion* involves each member trying to get as close as possible to its neighbors.
- *Avoidance* makes each member keep a certain distance from obstacles or members in other flocks, such as predators.

Flocking in Halloween Wars

ON THE CD

The *Halloween Wars* demonstration on the CD-ROM has two modes—Flocking mode and the *Halloween Wars* game mode. The mode can be changed by clicking on the button on the top right of the window. When run, it defaults to *Halloween Wars* mode. By clicking the Flocking button, you can change it into Flocking mode.

Within Flocking mode, there are four options (found on the bottom-right side of the window)—Ignore, Avoid, Predator/Prey, and Chase (see Figure 7.13).

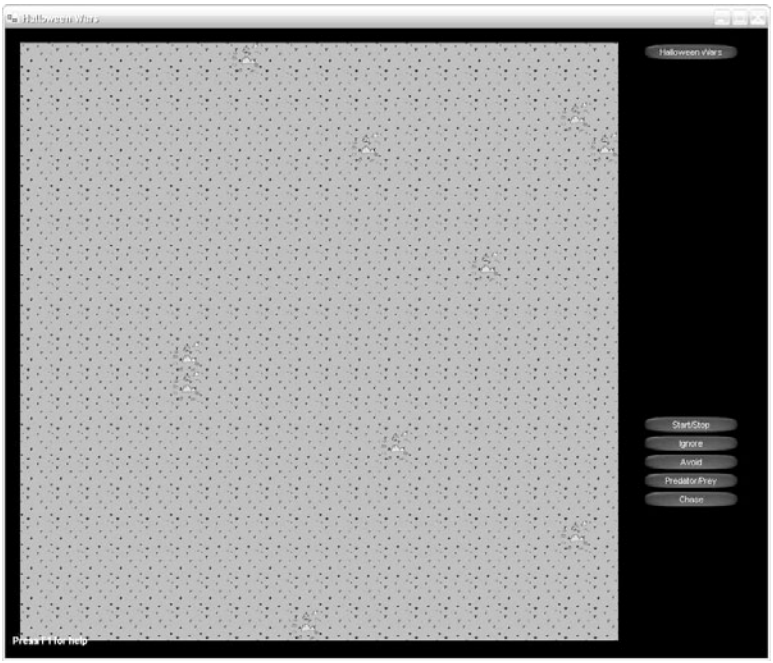


FIGURE 7.13 Flocking mode in the *Halloween Wars* demonstration includes four options—Ignore, Avoid, Predator/Prey, and Chase.

The basic steering behaviors used for the boids in the *Halloween Wars* flocking demo are separation, alignment, cohesion, and avoidance. In *Halloween Wars,* there are two, independent flocks, called *redTeam* and *blueTeam*, which correspond to the pumpkins and marshmallows, respectively. When the flocking simulation is started, each boid is randomly positioned on the map.

```
// for each boid in each team
for (int i = 0; i < army_size; i++)
{
       redSprite[i].Boid_Position();
       blueSprite[i].Boid_Position();
}

// randomly position on the map
void Boid_Position()
```

```
{
       position.X = rand.Next(map_width);
       position.Y = rand.Next(map_height);
}
```

As described in the flocking algorithm in the previous section, the center of mass and the average velocity are first calculated.

```
// create 2D vectors for the center of mass of each team
Vector2 redTeamMass = Vector2(0,0);
Vector2 blueTeamMass = Vector2(0,0);

// update each team's average velocity the their new velocity
redVelocity = redNewVelocity;
blueVelocity = blueNewVelocity;

// reset the new velocity, to be calculated in the flocking algorithm
redNewVelocity = Vector2(0,0);
blueNewVelocity = Vector2(0,0);

// add the positions of each boid together
for (int i = 0; i < army_size; i++)
{
       redTeamMass += redSprite[i].position;
       blueTeamMass += blueSprite[i].position;
}
// divide summed positions by the army size to get centre of mass
redTeamMass.X /= army_size;
redTeamMass.Y /= army_size;
blueTeamMass.X /= army_size;
blueTeamMass.Y /= army_size;
```

Subsequently, the new positions of each boid are calculated and updated using the steering behaviors. The steering behaviors that are employed vary by the mode set for the *Halloween Wars* flocking demo. Each of the four modes (Ignore, Avoid, Predator/Prey, and Chase) use steering behaviors for cohesion, separation, and alignment, as well as avoidance to avoid the edge of the map. The Predator/Prey and Avoid modes also use steering behaviors for avoidance of the other team. Finally, the Predator/Prey and Chase modes use a steering behavior similar to cohesion to move toward an object or the center of mass of the other team.

```
// Flocking algorithm for an individual boid

// initialize boid's new velocity
Vector2 velocity = Vector2(0,0);

velocity = Cohesion(teamMass);
velocity += Separation(teamSprite);

// predator/prey mode enabled
if (predatorEnabled)
{
       velocity += Avoidance(predator, enemyMass);
}

// avoid mode enabled
if (avoidEnabled)
{
       velocity += Avoidance(false, enemyMass);
}

// chase mode enabled
if (chaseEnabled)
{
       velocity += Chase(donut);
}

// avoid edge of map
velocity += AvoidEdge();

velocity += Alignment(teamVelocity, velocity);

// Move boid – but do not go outside edge of map
velocity = Move(velocity);
position += velocity;

// Update team velocity
teamNewVelocity += velocity;
```

Cohesion

Cohesion causes the boids to move toward the center of mass of the flock. Cohesion works by moving the boid a given fraction (denoted by the *COHESION_MOD*) closer to the center of mass of the flock. Both teams use cohesion to keep the flocks together in each mode. Figure 7.14 shows the cohesion steering behavior in the Ignore mode.

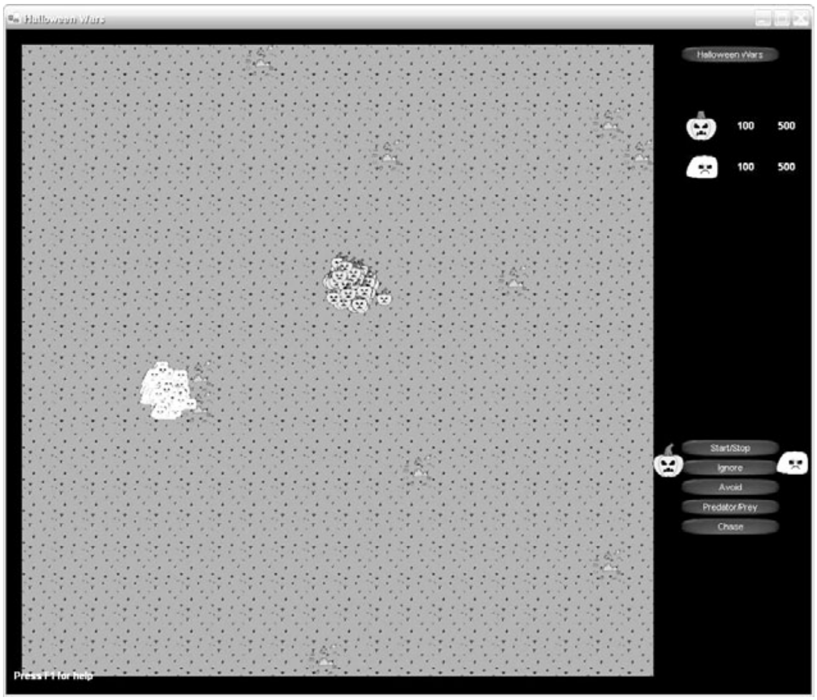


FIGURE 7.14 Cohesion draws the boids in toward their flock's center of mass.

```
const int COHESION_MOD = 300;
Vector2 Cohesion(Vector2 teamMass)
{
       Vector2 boidVelocity = Vector2(0,0);
       boidVelocity.X = (teamMass.X - position.X)/COHESION_MOD;
       boidVelocity.Y = (teamMass.Y - position.Y)/COHESION_MOD;
       return boidVelocity;
}
```

Separation

Separation is used in each of the modes and causes the boids to keep a minimum distance from other boids in the flock. Separation causes the appearance of the flock formation, because the boids attempt to maintain a minimum distance from their fellow boids, giving rise to uniformly spaced boids. Separation works by checking for fellow boids within a given distance (*SEPARATION_MIN*) and increasing the boid's distance from that boid by a given percentage (*SEPARATION_MOD*). Figure 7.15 shows the separation steering behavior in the Avoid mode.

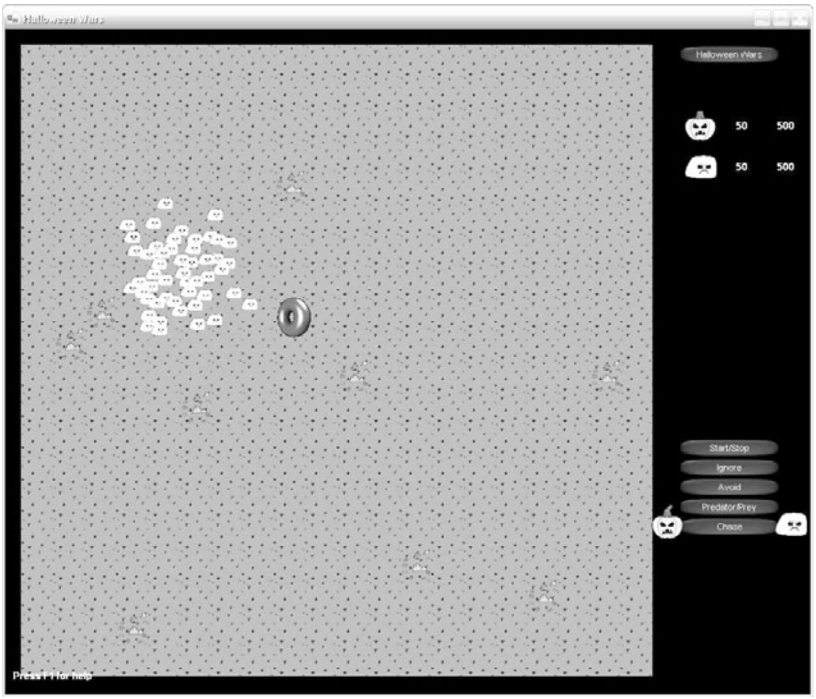


FIGURE 7.15 Separation keeps the boids a minimum distance from their fellow flockmates.

```
SEPARATION_MIN = 3
SEPARATION_MOD = 1.5
Vector2 Separation(MovingSprite[] teamSprite)
{
       Vector2 boidVelocity = Vector2(0,0);
       double distanceXY = 0;

       for (int i = 0; i < Simple2D.army_size; i++)
      {
             Vector2 separation;
             separation.X = teamSprite[i].position.X - position.X;
             separation.Y = teamSprite[i].position.Y - position.Y;

             if ((separation.X == 0) && (separationY == 0))
             {
                    distanceXY = 0;
             }
             else
```

```
                {
                       distanceXY = Math.Sqrt((separationX * separationX)
                              + (separation.Y * separationY));
                }

                if (distanceXY < SEPARATION_MIN)
                {
                       boidVelocity.X += (separation.X * SEPARATION_MOD);
                       boidVelocity.Y += (separation.Y * SEPARATION_MOD);
                }
         }
         return boidVelocity;
  }
```

Avoidance

Avoidance is used in each of the modes to deter the flock from coming into contact with the edge of the map. If a boid is within a certain distance of the edge of the map (*AVOID_EDGE_MIN*), it increases this separation distance by a given factor (*AVOID_EDGE_MOD*). Figure 7.16 shows the avoidance of the edge of the map steering behavior in the Avoid mode.

```
const int AVOID_EDGE_MIN = 10;
const int AVOID_EDGE_MOD = 2;
Vector2 AvoidEdge()
{
       Vector2 boidVelocity = Vector2(0,0);

       // Avoidance edge of screen
       if ((maxWidth - position.X) < AVOID_EDGE_MIN)
              boidVelocity.X -= ((maxWidth - position.X) *
                AVOID_EDGE_MOD);
       if ((maxHeight - position.Y) < AVOID_EDGE_MIN)
              boidVelocity.Y -= ((maxHeight - position.Y) *
                AVOID_EDGE_MOD);
       if ((position.X – minWidth) < AVOID_EDGE_MIN)
              boidVelocity.X += (position.X * AVOID_EDGE_MOD);
       if ((position.Y – minHeight) < AVOID_EDGE_MIN)
              boidVelocity.Y += (position.Y * AVOID_EDGE_MOD);

       return boidvelocity;
}
```

Avoidance is also used in the Predator/Prey and Avoid modes, but in a different way. Rather than using a separation-type rule, the boids use a rule similar (but

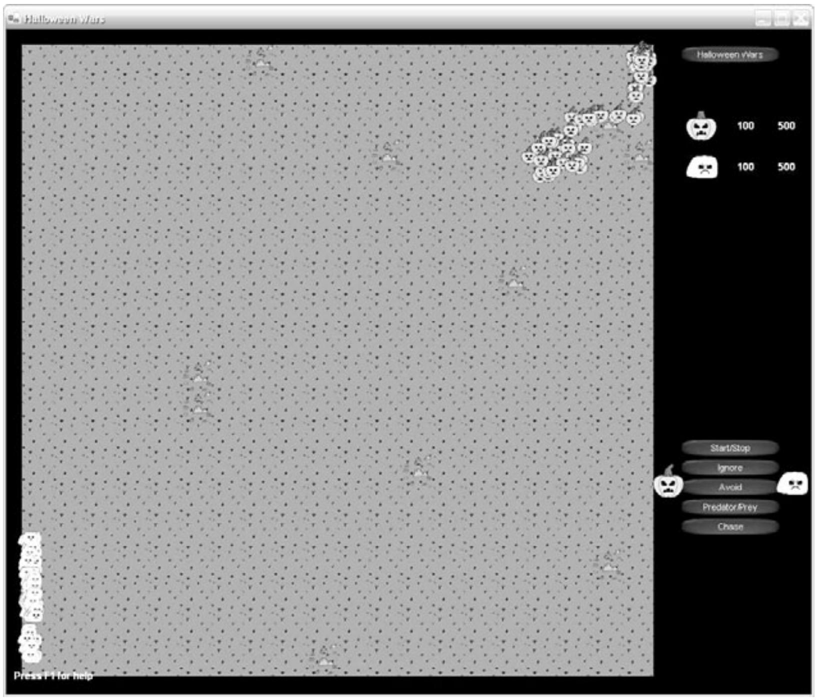


FIGURE 7.16 Avoidance keeps the boids a minimum distance from the edge of the map.

exactly opposite) to cohesion to avoid the other team. In the Avoid mode, boids in both teams check their distance from the center of mass of the other team and increase the separation by a given factor (*PREY_MOD*).

In the Predator/Prey mode, the prey (marshmallows) works in the same way as in the Avoid mode, but the predators (pumpkins) decrease their separation by a given factor (*PREDATOR_MOD*). To give them a sporting chance, the prey can move faster than the predators, which is denoted by a smaller modification factor, meaning that they can increase their separation from the predators further than the predators can decrease the separation each timestep. Figure 7.17 shows the avoidance steering behavior in the Predator/Prey mode.

```
const int PREDATOR_MOD = 1500;
const int PREY_MOD = 500;
Vector2 Avoidance(bool predator, Vector2 enemyMass)
{
       Vector2 boidVelocity = Vector2(0,0);
```

```
        if (predator)
        {
               // Chase the other team
               boidVelocity.X = (enemyMass.X - position.X)/PREDATOR_MOD;
               boidVelocity.Y = (enemyMass.Y - position.Y)/PREDATOR_MOD;
        }
        else
        {
               // Avoid the other team
               boidVelocity.X -= (enemyMass.X - position.X)/PREY_MOD;
               boidVelocity.Y -= (enemyMass.Y - position.Y)/PREY_MOD;
        }
        return boidVelocity;
}
```

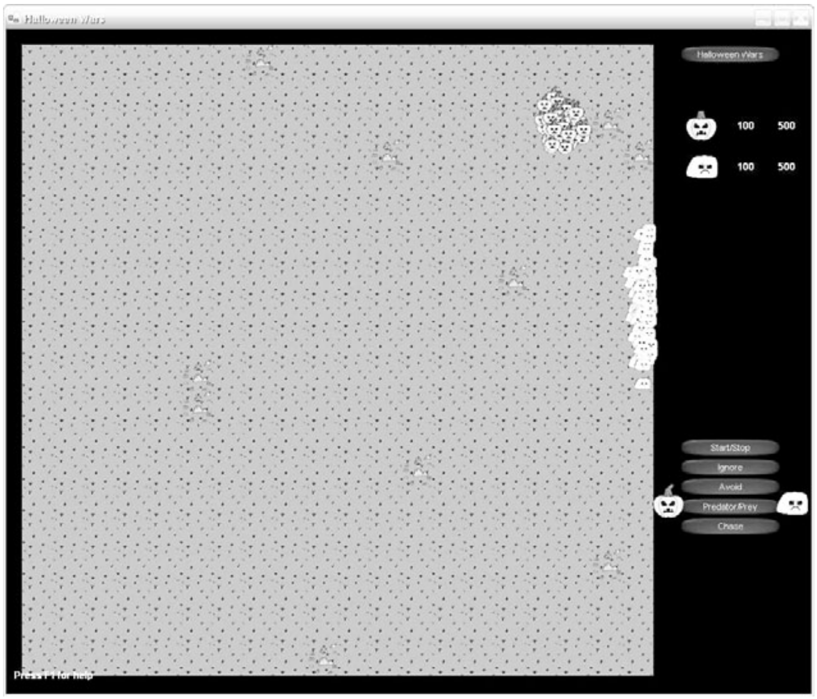


FIGURE 7.17 Avoidance enables the prey to avoid their predators.

Chase

The Chase mode uses a steering behavior similar to the predators in the Predator/Prey mode, which allows the boids to move closer to the object they are chasing. The object itself (a donut) has a random movement pattern. Each timestep, the chasers (marshmallows) move a given fraction (*CHASE_MOD*) closer to the moving object, while maintaining their flock formation. Figure 7.18 shows the marshmallows using the chase steering behavior to chase the donut.

```
const int CHASE_MOD = 500;
Vector2 Chase(const Vector2 donut)
{
      // Move toward the donut
      boidVelocity.X = (donut.X - position.X)/CHASE_MOD;
      boidVelocity.Y = (donut.Y - position.Y)/CHASE_MOD;

      return boidVelocity;
}
```

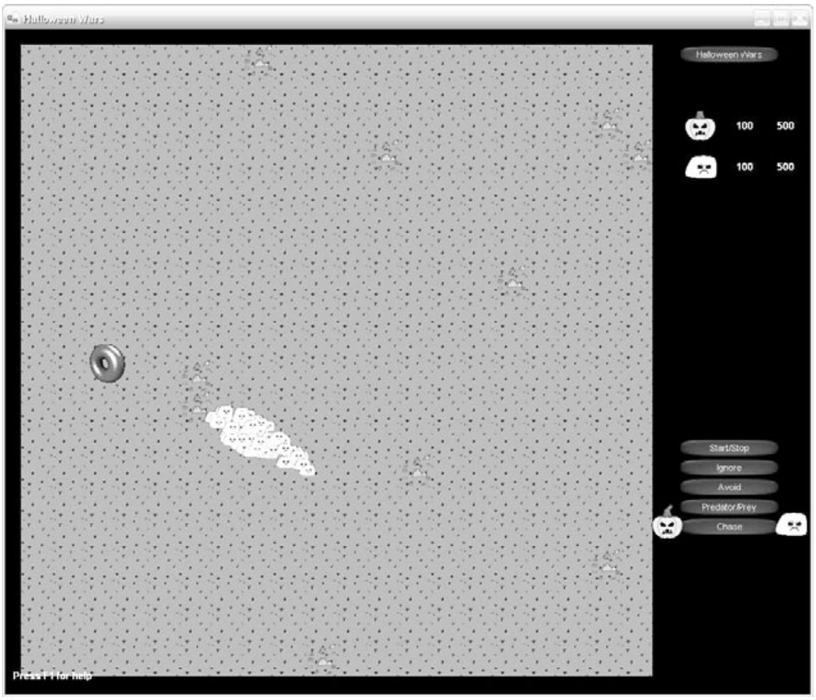


FIGURE 7.18 The marshmallows chase the donut, while maintaining their flock formation.

Alignment

The alignment steering behavior is also used in each flocking mode in the *Halloween Wars* demo, and is a staple of the flocking algorithm. Alignment allows a boid to move toward the perceived heading of the flock. Each boid takes the average velocity of the flock and changes its own heading by a given factor (*ALIGNMENT_MOD*) to move toward the average heading.

```
const int ALIGNMENT_MOD = 10;
Vector2 Alignment(const Vector2 teamVelocity, const Vector2 velocity)
{
       // Alignment
       boidvelocity.X = (teamVelocity.X - velocity.X)/ALIGNMENT_MOD;
       boidvelocity.Y = (teamVelocity.Y - velocity.Y)/ALIGNMENT_MOD;

       return boidvelocity;
}
```

Update

Finally, the boids are moved to their new position and the flocking algorithm starts again, without any memory of where they moved last cycle. In this example, the boids are also locked to the boundaries of the map. Their avoidance steering behavior does deter them from moving too close to the edge of the map, but it is possible that the repulsion from avoiding another flock or predator might outweigh their desire to stay within bounds.

```
Vector2 Move(const Vector2 velocity)
{
       Vector2 boidVelocity;

       // don't go outside edge of map
       if ((position.X + velocity.X) > maxWidth)
              boidVelocity.X = (maxWidth - position.X);
       else if ((position.X + velocity.X) < minWidth)
              boidVelocity.X = (position.X – minWidth);
       else boidVelocity.X = velocity.X;

       if ((position.Y + velocity.Y) > maxHeight)
              boidVelocity.Y = (maxHeight - position.Y);
       else if ((position.Y + velocity.Y) < minHeight)
              boidVelocity.Y = (position.Y – minHeight);
       else boidVelocity.Y = velocity.Y;

       return boidVelocity;
}
```

Constants

There are several global constants used throughout the flocking algorithm that can be tuned to substantially modify the behavior of the flock. Try altering each of these constants and see how it changes the behavior of the flock. A summary of the constants is shown in Table 7.3.

TABLE 7.3 Global Constants in Flocking Algorithm

Constant	Value	Purpose
COHESION_MOD	300	Determines how much closer the boid will move to the center of mass of the flock
SEPARATION_MIN	3	The distance within which the boid checks for flockmates
SEPARATION_MOD	1.5	Factor by which boid will increase current distance from flockmate
AVOID_EDGE_MIN	10	The distance within the boid checks for the edge of the map
AVOID_EDGE_MOD	2	Factor by which boids will increase current distance from the edge of the map
PREDATOR_MOD	1500	Factor by which the boids decrease their distance from their prey
PREY_MOD	500	Factor by which the boids increase their distance from their predators
CHASE_MOD	500	Factor by which the boids decrease their distance from the object they are chasing
ALIGNMENT_MOD	10	Factor by which the boid aligns its heading with the flock's average velocity

Emergent Group Movement

The four modes in the *Halloween Wars* flocking demo demonstrate four different behaviors, based on the same set of simple rules with minor variations. Changing the rules or modifiers, evenly slightly, gives rise to different behavior. Adding or removing individual rules has a substantial effect on the behavior. The combination of the simple steering behaviors in flocking, when applied to many individual entities, illustrates the power, potential, and variability of emergence in games.

The rules themselves are simple to program and easy to understand, but they can give rise to interesting, lifelike, and complex behavior. The key is to find the right set of simple rules and tune the variables to give the desired behavior for your game. The next section takes the same set of simple rules and alters them to suit a game scenario. The result is a simple movement, tactics, and AI system for a basic strategy game.

Tactics

Tactics involve a group or team cooperating and behaving in a coordinated way in order to achieve a group goal, such as securing an area, defeating the enemy, winning a match, or making a successful play. As well as coordinated group movement, agent-based systems can be used to create emergent group tactics. If each agent considers its goals, personality, current situation, and the behavior of its team and opponents, it can make low-level decisions that will allow the high-level tactics and strategies to be emergent.

Tactics in Halloween Wars

In the *Halloween Wars* game mode, two teams (pumpkins and marshmallows) fight against each other for domination. Each side starts with 100 soldiers, 500 reinforcements, and control of three of the six domination points (that is, donuts) (see Figure 7.19). The objective of the game is to hold all of the domination points simultaneously, or kill all of the opposing force on the field.

Reinforcements

Each side can call for reinforcements when their fielded army drops below 100 soldiers. The players can call reinforcements by clicking the Reinforcements button on the bottom-right of the screen. When reinforcements are called, the fielded army is boosted back up to 100, or as many reinforcements remain. Reinforcements can be called at any time, until the 500 soldiers in reserve are used up. Reinforcements are deployed at each side's deployment point (top-right for marshmallows and bottom-left for pumpkins).

Stances

Each side can switch its stance between Attack, Defend, Capture, and Hold, each of which corresponds to a button on the bottom-right of the window. Each of the stances has the following effect:

- Attack—Move toward the center of mass of enemy soldiers.
- Defend—Move toward the center of mass of friendly soldiers.
- Capture—Move toward the nearest enemy domination point.
- Hold—Move toward the nearest friendly domination point.

However, these stances are not direct orders that are given to the team. They change the weights that are applied to each steering behavior. For example, a team

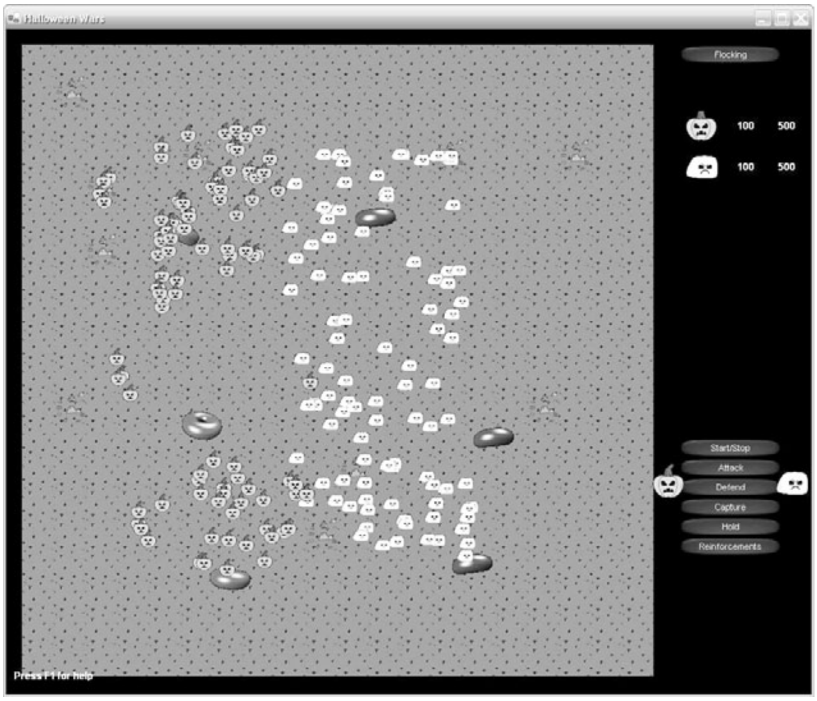


FIGURE 7.19 Each side starts with 100 soldiers, 500 reinforcements, and control of three domination points.

given the Attack stance will move more toward the other team than toward its own team members. The actual effect of each of these stances will be explained in the following flocking algorithm.

Key Terms

- *Tactics* involves agents cooperating in order to achieve a common goal.
- *Stances* change weights that are applied to steering behaviors.

Game Cycle

When a new game is started, each team is given a full complement of soldiers and reinforcements. The soldiers are randomly placed on each half of the map, as are the domination points for each team. The pumpkin's (*redTeam*) starting stance is randomly set and the marshmallow's (*blueTeam*) stance is initially set to Defend.

During each cycle, the game checks to determine if one team has won. A team can win by killing all the fielded soldiers on the other team or by capturing all the domination points. Next, the *redTeam* (controlled by the AI) evaluates the current strategic position and sets the stance of the team.

Strategic AI The strategic AI is a simple list of prioritized switches, as follows:

- If the *redTeam* do not control the majority of domination points, their stance is set to Capture.
- Otherwise, if the *redTeam* outnumber the *blueTeam*, their stance is set to Attack.
- Otherwise, if the *blueTeam* have more than twice the strength of the *redTeam*, their stance is set to Defend.
- Otherwise, their stance is set to Hold (they are outnumbered by the *blueTeam*, but by less than double).

The code for the strategic AI is as follows:

```
if (numRed <= (flags/2))
       redPriorities.current_stance = capture;
else if (redAlive > blueAlive)
       redPriorities.current_stance = attack;
else if (blueAlive > (2*redAlive))
       redPriorities.current_stance = defend;
else
       redPriorities.current_stance = hold;
```

Combat Next, the combat is resolved. The combat simply runs through the *redTeam* and checks for *blueTeam* soldiers that are in range. For each soldier that is found in range, the *redTeam* soldier has a 25% chance to kill the *blueTeam* soldier, and vice versa, with a 50% chance that both will survive.

```
for (int i = 0; i < army_size; i++)
{
       if (redArmy[i].alive)
       {
              for (int j = 0; j < army_size; j++)
              {
                    // for each enemy alive within range
                    if ((blueArmy[j].alive)
                          && (redSprite[i].InRange(blueSprite[j])))
                    {
                          int r = rand.Next(100);
                    // 25% chance to kill or be killed
                    if (r < 25)
                    {
                         blueArmy[j].alive = false;
                         blueAlive--;
                    }
                    else if (r < 50)
                    {
                         redArmy[i].alive = false;
                         redAlive--;
                         j = army_size;
                    }
               }
          }
     }
}
```

Domination Subsequently, the game checks to determine which team is in control of each flag (donut). The number of soldiers of each team that are in range of the flag is counted and the team with the majority in range controls the flag.

```
// capture the flag
for (int i = 0; i < flags; i++)
{
       // if enemy in range outnumbers friendly change ownership
       int redInRange = 0;
       int blueInRange = 0;

       for (int j = 0; j < army_size; j++)
       {
              if ((redSprite[j].InRangeFlag(flagPos[i].x, flagPos[i].y))
                     && redArmy[j].alive)
                     redInRange++;
              if ((blueSprite[j].InRangeFlag(flagPos[i].x, flagPos[i].y))
                     && blueArmy[j].alive)
                     blueInRange++;
       }
       if (redInRange > blueInRange)
              flagID[i] = RED;
       else if (blueInRange > redInRange)
              flagID[i] = BLUE;
}
```

Movement The flocking algorithm is then run for each team, to determine where each soldier will move next. Each stance has an associated set of weights for moving toward friendly flags (*friendly_flag*), moving toward enemy flags (*enemy_flag*), moving toward friendly soldiers (*friendly_army*), and moving toward enemy soldiers (*enemy_army*). Each modifier further modifies the weight of each steering behavior (see Table 7.4).

TABLE 7.4 Weights for Each Strategic Stance

Weight	Attack	Defend	Capture	Hold
friendly_flag	0.1	0.2	0.1	1.0
enemy_flag	0.2	0.1	1.0	0.1
friendly_army	0.1	1.0	0.1	0.2
enemy_army	1.0	0.1	0.2	0.1

Steering Behaviors

The flocking algorithm that is run for the game mode is very similar to the algorithm for the flocking demo mode, except for the application of the additional weights for stances and some other small variations.

```
// initialize boid's new velocity
Vector2 velocity;

velocity = Cohesion(teamMass);
velocity += Separation(teamSprite);

// chase the enemy team
velocity += Chase(enemyMass);

// avoid edge of map
velocity += AvoidEdge();

// chase flag
velocity += ChaseFlag();

velocity += Alignment(teamVelocity, velocity);

// Move boid – but do not go outside edge of map
velocity = Move(velocity);
position += velocity;
```

```
// Update team velocity
teamNewVelocity += velocity;
```

Cohesion The cohesion rule is identical to the cohesion rule in the flocking demo, except that the base cohesion modifier (*COHESION_MOD*) is increased to 500, to create a looser formation. The change in position due to cohesion is also modified by the *friendly_army* weight. Consequently, the soldiers flock closer together in the Defend stance, where the *friendly_army* weight is higher (see Figure 7.20).

```
const int COHESION_MOD = 500;
Vector2 Cohesion(Vector2 teamMass)
{
       Vector2 boidVelocity;
       boidVelocity.X = (teamMass.X - position.X)
              / (COHESION_MOD / stance.friendly_army);
       boidVelocity.Y = (teamMass.Y - position.Y)
              / (COHESION_MOD / stance.friendly_army);
       return boidVelocity;
}
```

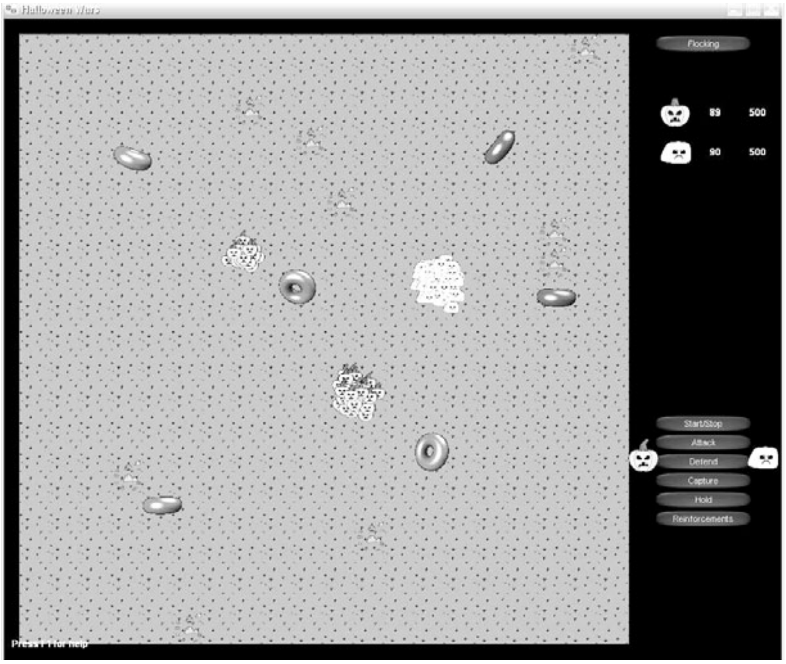


FIGURE 7.20 Soldiers in the marshmallow team flock together when set to Defend.

Separation The separation rule is identical to the separation rule in the flocking demo, with the same modifiers applied. This ensures that the soldiers keep a minimum distance from each other at all times. However, it would be possible to modify the separation distance by the *friendly_army* modifier, or another modifier, so that soldier formations would become tighter or looser depending on strategic stances. Try modifying the separation rule with the *friendly_army* modifier to see what effect it has.

Chase The chase rule works the same as the chase rule used to chase the donut in the Chase mode of the flocking demo. However, instead of chasing the donut, the soldiers are chasing the center of mass of the enemy team. The base chase modifier (*CHASE_MOD*) has the same value as in the flocking demo, but the value is also modified by the *enemy_army* weight. As a result, the soldiers move closer to the enemy army in the Attack stance, where the *enemy_army* weight is higher (see Figure 7.21).

```
const int CHASE_MOD = 500;
Vector2 Chase(const Vector2 target)
{
       Vector2 boidVelocity;

       // Chase the other team
       boidVelocity.X = (target.X - position.X)
              / (CHASE_MOD / stance.enemy_army);
       boidVelocity.Y = (target.Y - position.Y)
              / (CHASE_MOD / stance.enemy_army);

       return boidVelocity;
}
```

Chase Flag The major variation from the flocking demo comes with the need to capture and hold flags. A team's stance can be set to either capture enemy flags or hold their own flags. Each soldier chooses only one flag to move toward each cycle. The best flag for each soldier to move toward is determined by the distance between the flag and the soldier, modified by the soldier's *enemy_flag* and *friendly_flag* weights, depending on the team to which the flag currently belongs.

The result is that the soldier moves toward the closest flag of the type it is trying to pursue. When the Capture stance is set, the soldiers will move toward the nearest enemy flag, and when the Hold stance is set, the soldiers will move toward the nearest friendly flag (see Figure 7.22). If the team is set to Attack or Defend, the soldier will move slightly closer to the nearest enemy or friendly flag, respectively.

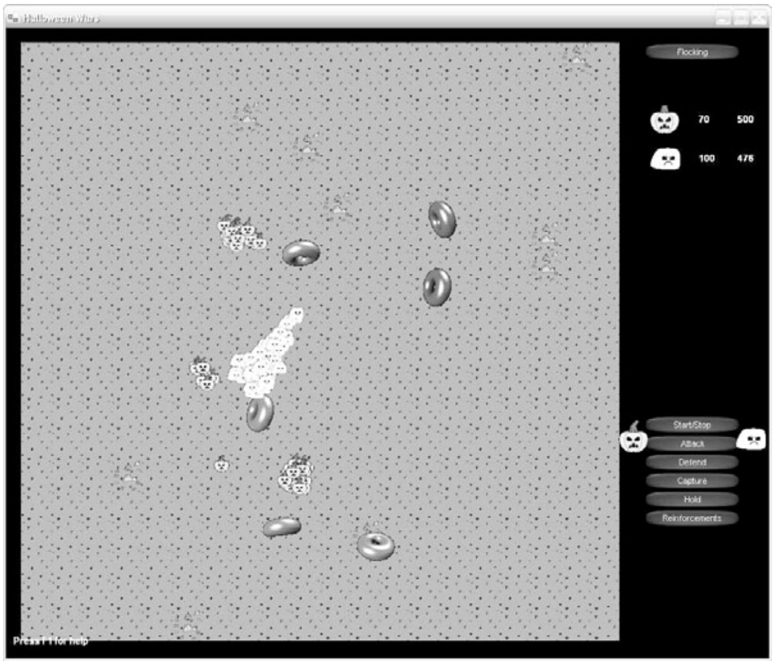


FIGURE 7.21 Soldiers in the marshmallow team move closer to the enemy pumpkin army when set to Attack.

However, if the team is set to Attack or Capture and the nearest enemy flag is too far from the soldier, it will head toward the nearest friendly flag. The point at which it deems the flag to be too far away can be tuned by the *enemy_flag* and *friendly_flag* weights. With the current weight settings, the nearest enemy flag must be twice as far away as the nearest friendly flag when the team is set to Attack (that is, *enemy_flag* = *0.2, friendly_flag* = *0.1*), and 10 times as far when they are set to Capture (that is,. *enemy_flag* = *1.0, friendly_flag* = *0.1*).

```
const int FLAG_CHASE_MOD = 300;
const int FLAG_DISTANCE_MOD = 10;
Vector2 ChaseFlag()
{
       int goal = 0;
       double goal_dist;
       float flag_mod, goal_mod;
       Vector2 boidVelocity;

       // Go toward the most desirable flag
       for (int r = 1; r < flags; r++)
```

```
        {
               // is this our flag or the enemy's flag, weight accordingly
               if (flagOwner[r] == ourFlag)
                      flag_mod = stance.friendly_flag;
               else
                      flag_mod = stance.enemy_flag;

               // is current goal our flag or the enemy's, weight
               // accordingly
              if (flagOwner[goal] == ourFlag)
                    goal_mod = stance.friendly_flag;
              else
                    goal_mod = stance.enemy_flag;

              // calculate weighted distance to current goal flag
              goal_dist = Math.Sqrt (((flagPos[goal].x - position.X)
                     * (flagPos[goal].x - position.X))
                     + ((flagPos[goal].y - position.Y)
                     * (flagPos[goal].y - position.Y)))
                     * (FLAG_DISTANCE_MOD / goal_mod);

              // calculate weighted distance to this flag
              double this_dist = Math.Sqrt (((flagPos[r].x - position.X)
                     * (flagPos[r].x - position.X))
                     + ((flagPos[r].y - position.Y)
                     * (flagPos[r].y - position.Y)))
                     * (FLAG_DISTANCE_MOD / flag_mod);

              // if this flag is better, it's the new goal
              if (this_dist < goal_dist)
                   goal = r;
       }

       if (flagOwner[goal] == ourFlag)
             flag_mod = friendly_flag;
       else
             flag_mod = enemy_flag;

       boidVelocity.X = (flagPos[goal].x - position.X)
             / (FLAG_CHASE_MOD / flag_mod);
       boidVelocity.Y = (flagPos[goal].y - position.Y)
             / (FLAG_CHASE_MOD / flag_mod);

       return boidVelocity;
 }
```

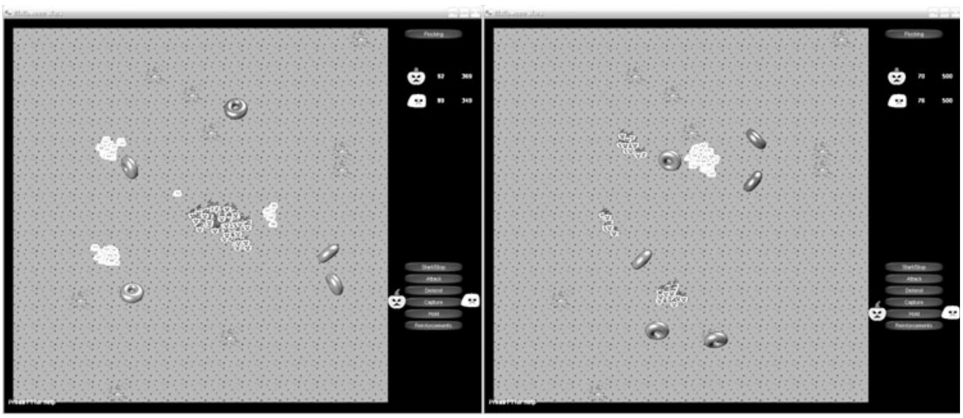


FIGURE 7.22 Soldiers in the marshmallow team move closer to the nearest enemy flag when set to Capture (left) and the nearest friendly flag when set to Hold (right).

Alignment Alignment works the same as in the flocking demo, with the addition of the *friendly_army* weight being applied to the soldier's velocity. Consequently, the soldiers align more closely to the heading of their teammates in the Defend stance than in the other stances (see Figure 7.23).

```
const int ALIGNMENT_MOD = 10;
Vector2 Alignment(const Vector2 teamVelocity, const Vector2 velocity)
{
       Vector2 boidVelocity;

       // Alignment
        boidVelocity .X = (teamVelocity.X - velocity.X)
             / (ALIGNMENT_MOD / friendly_army);
        boidVelocity.Y = (teamVelocity.Y - velocity.Y)
             / (ALIGNMENT_MOD / friendly_army);

       return boidVelocity;
}
```

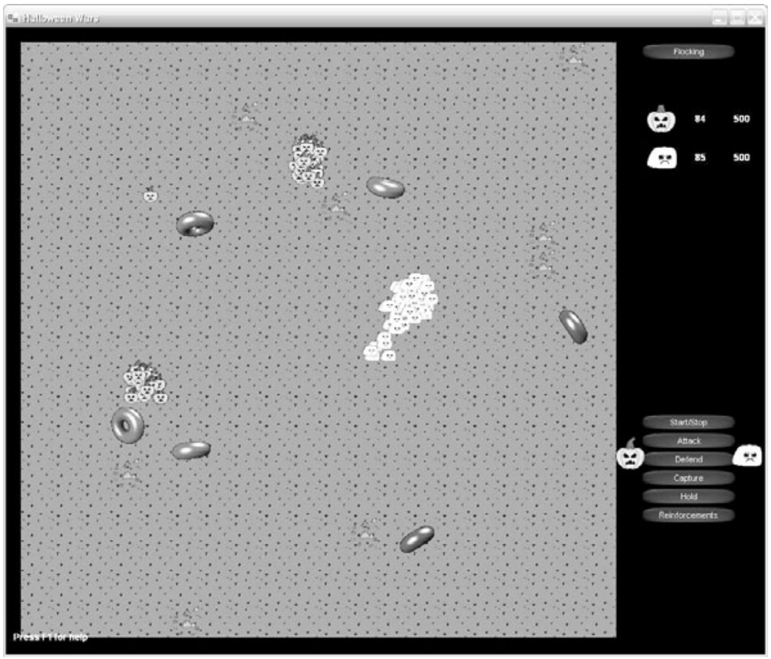


FIGURE 7.23 Soldiers in the marshmallow team align more closely to the heading of their teammates when set to Defend.

KEY TERMS

- *Chase* involves the soldiers chasing the center of mass of the enemy team.
- *Chase flag* involves a soldier moving toward the closest flag of the type it is trying to pursue.

Emergent Tactics

With the simple steering behaviors of cohesion, separation, alignment, chase, and chase flag, the two teams can attack, defend, hold, and capture flags, giving rise to interesting gameplay. However, more significantly, there are several interesting tactical behaviors that emerge from these simple rules and stances.

Each team can divide into multiple groups, defending multiple flags simultaneously or attacking on multiple fronts (see Figure 7.24).

Teams can divide and conquer, splitting into groups then attacking the enemy from two or three sides at the same time (see Figure 7.25).

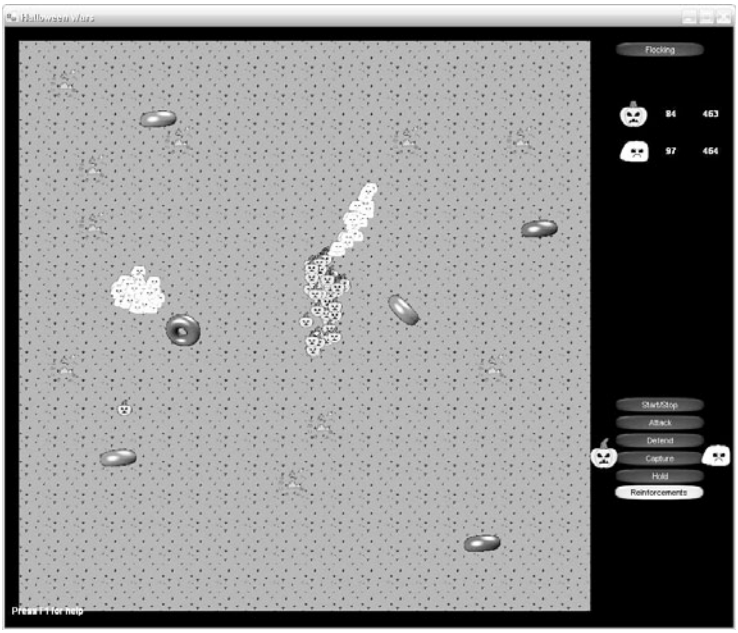


FIGURE 7.24 Teams can divide into multiple groups.

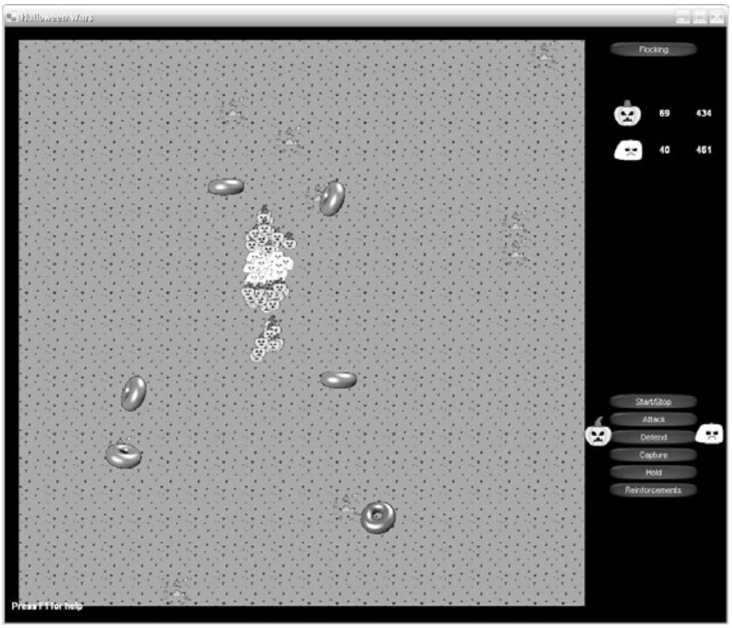


FIGURE 7.25 Teams can attack the enemy from multiple sides.

An attacking team forms an offensive formation to drive into the enemy and a defending team forms a tight, defensive structure. Two attacking teams meet head on to engage the enemy (see Figure 7.26).

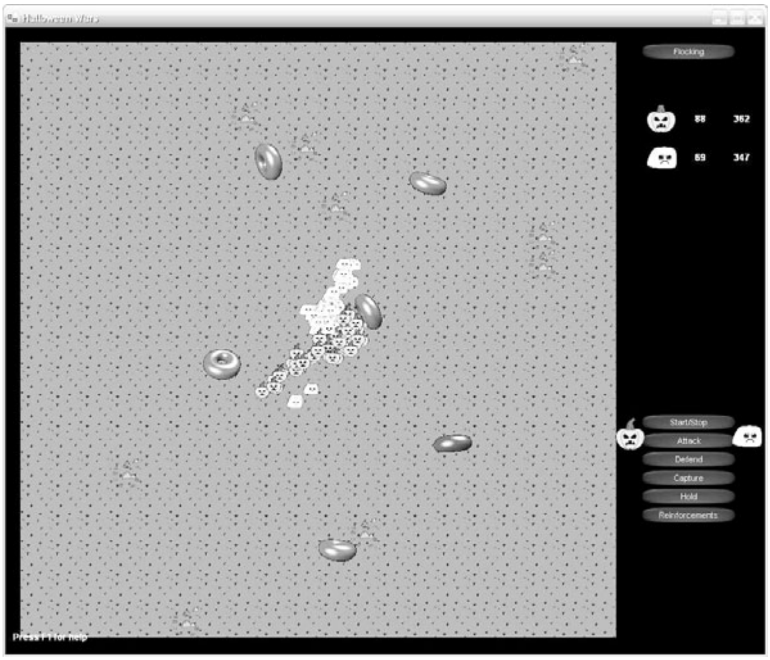


FIGURE 7.26 Attacking teams form offensive formations.

Teams that are holding their flags position themselves between their flags and the enemy (see Figure 7.27).

The teams move around fluidly, rapidly adapting to their teammates, enemies, and the state of the game. The order that stances are set and the timing of reinforcements become important, because they determine the center of mass and velocity of the teams at any one time, which affect the behavior of the teams when their stance is changed. The state of the game and the current stance and position of the other team also greatly affect the behavior of a team and the resulting gameplay.

From a simple set of steering behaviors that determine the movements of individual soldiers in relation to their own team, enemy team, and flags, the group movement, tactics, and gameplay is fluid, dynamic, and emergent. Altering the rules, adding new rules or removing rules, and changing weights and stances can give rise to great deviations in behavior and gameplay. This agent-based approach is flexible, powerful, and can give rise to emergent, lifelike behavior in a variety of games, game situations, and agents.

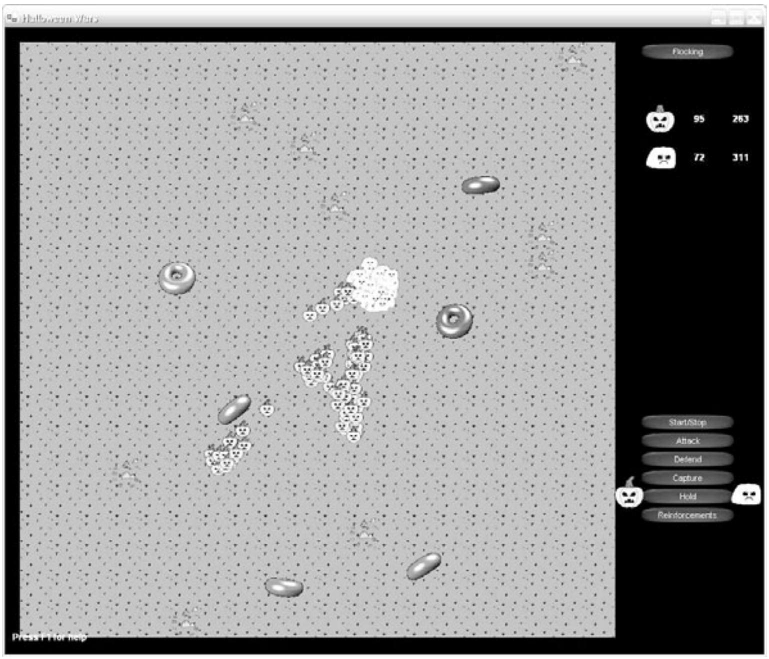


FIGURE 7.27 Teams that are holding their flags position themselves between their flags and the enemy.

ADDITIONAL READING

For further information on agent-based systems for combat simulation:

- Ilachinski, A. (2004). *Multiagent-Based Simulation of Combat*. River Edge, NJ: World Scientific Publishing Company.

Emergent Group Behavior

The basic model of agent-based movement and tactics presented in this chapter can be easily extended by adding different types of agents, new gameplay mechanics, and new steering behaviors and stances, to create a wide range of different games, behaviors, tactics, and gameplay. The players could be given more control, setting the stances of groups or individual agents, or giving more direct orders to be followed. Different types of agents could have different steering behaviors, combat mechanics, and abilities, giving a wider range of possibilities for action and combinations with other agents. New objectives, gameplay modes, terrain, and maps could be added to create a wide range of playing experiences.

Agent-based steering behaviors have many potential possibilities to be used in games. They are simple, flexible, easy to understand and program, and can give rise to interesting, varied, and emergent behavior and gameplay. Agent-based steering behaviors, such as flocking and related algorithms, can be used to create fundamental gameplay or emergent group movement and tactics, or simply to create more lifelike movement behaviors for background characters or animals.

As with any emergent system, the most time-consuming part is the tuning process. The best approach is to start with the simplest and minimal set of rules and build up from there. Add in your first one or two rules and ensure they are tuned and working properly before building on the system. The fewer rules and variables you have, the easier the system will be to tune and understand.

The most effective and robust emergent systems are built on very few, simple, fundamental rules. Remember that the emergence in an agent-based system comes from the complexity created by the interaction of a large number of very simple entities. The more complex these entities are themselves, the more chaotic and confusing the system can become. Design the system in such a way that the entities follow a set of simple, local rules for behavior, and that the emergence is a result of their interactions with each other and their reactions to their complex and dynamic environment.

INTERVIEW WITH CRAIG REYNOLDS

Senior Researcher, Sony Computer Entertainment,
US R&D

Craig Reynolds researches technology for autonomous characters at Sony Computer Entertainment's US R&D group in Foster City, California (www.research.scea.com). Recent projects include PSCrowd, a high-performance crowd simulator for PS3, and OpenSteer, an open source library of steering behaviors. He has previously worked on animation and game production, plus developed tools for both fields, at: DreamWorks, Silicon Studio, Electronic Arts, Symbolics, and Information International Inc. He won a Scientific and Engineering Academy Award in 1998 for "pioneering contributions to the development of three-dimensional computer animation for motion picture production."

Æ

In what areas do you think emergence has the most potential to improve current and future games?
Generally, I would expect the use of emergence in games to provide a less scripted, more improvisational nature to gameplay. This would tend to create more unique and less predicable experiences for the players. That in itself is probably a worthy goal, but in addition it means that the game would have more "replayability" because each time players go through the game it will be somewhat different. It also suggests that each player's experience would be unique, so this creates a topic of conversation between players, say on online forums, where they can compare notes about what happened to each of them at a given point in the game.

What are the benefits of emergence for the players and the impact on their enjoyment?
Replayability. Unique experience for each player. Freedom to explore unique and idiosyncratic strategies that the designer never thought of.

How does emergence change the gameplay and the player's role in the game?
An emergent game world is unpredictable and so it is hard for the designer to impose the same sort of fine-grained control over progress of the story, dramatic tension, and other aspects that are important to many of today's games and form the very basis of other art-forms like novels, plays, and movies. It has been noted elsewhere that there is a tension between finely crafted storytelling and "sandbox" game worlds where the players are always free to choose their own actions. It seems there may be a similar disconnect from emergent gameplay and carefully designed and satisfying drama.

What kind of methods and techniques can be used to create emergent gameplay?
Populating a game world with interacting autonomous characters produces a complex environment for any sort of gameplay. For example, a game could involve a car chase through streets full of autonomous vehicles. Sometimes traffic would help the hunter, sometimes the hunted. I've heard *Assassin's Creed* will use city crowds like this and the people in the crowds react to the player characters, so the crowd becomes part of the gameplay.

Other emergent elements can lend variety to games. Story fragments may interact with each other in an emergent drama manager, vying for the right to control the game flow for a while. Analogous techniques have been developed where character animation is synthesized from a library of motion-capture data with individual motion clips vying to move the character smoothly into a desired pose.

Æ

Autonomous characters can be augmented with procedural models of emotion and the equivalent of a long memory. Then interactions with the player's character will be modulated by the autonomous character's emotional state, which will be driven in part by memories of previous encounters between the two.

How can flocking and steering behaviors be used in games for emergence and emergent gameplay?
Crowds are related to flocks, at least in the abstract simplified sense of 'boid flocks.' In the natural world, crowds are more closely related to herds, being constrained to a surface, able to move slowly or stop, and able to handle very small separation distances. Unlike a herd of cattle on open terrain, urban crowds are constrained by many other factors: sidewalks, buildings, traffic lights, and vehicles. Dealing with these other factors requires the use of other steering behaviors such as path following, obstacle avoidance, and evasion. So an agent-based 'urban pedestrian' model *could* be seen as a specialized sort of flock model, but it is probably more useful to consider both of them as examples of interacting steering behaviors. 'Vehicles in traffic' is another similar one.

Seen from another perspective, crowds in games are less like crowds in real life and more like crowds of 'extras' in a movie. Crowds in games or movies look superficially like real crowds but can have very unrealistic motivations, such as an innate desire to avoid getting between the main character and the camera. Or, in order to advance a plot point, a villain can be allowed to slip through the crowd while the player character is subtly blocked by the action of the crowd.

The rules for flocking that you developed are very simple and elegant, but how complex was the process to arrive at these simple rules?
I chalk it all up to luck. The process was simple, but certainly not elegant. In the absence of any hard data or experimentation, I guessed at three rules. I think my feeling was that these three rules were probably necessary, but might not be sufficient. By a great stroke of luck, they worked almost unchanged. I devised the three rules both by observing natural flocks and by imagining what it would be like to be a bird in a flock, based on my own experience being in crowds and vehicle traffic.

The hardest part was the amount of time that the idea was just knocking around in my head before I actually took the time to try it. I think I mentioned it in my undergraduate thesis in 1975, saying that making a flock model would be a simple matter of programming behavior for one bird, then making many instances of it and allowing them to interact. Eleven years later I was preparing

Æ

slides for a tutorial talk at SIGGRAPH 1986 and figured that I better either implement it or stop saying how simple it would be. In a couple weeks, I was able to make a prototype layered on top of S-Geometry and S-Dynamics, the 3D tools developed by the group where I worked at Symbolics.

I was lucky that a simple combination of the three rules produced interesting behavior since in retrospect it has a lot of shortcomings. For example, Bajec (http://lrss.fri.uni-lj.si/people/ilbajec/publications.htm) pointed out that an isolated boid always flies in a straight line—very unlike an isolated bird. There has been a lot of follow-up work by others looking at various improvements to the model.

How far have agent-based steering behaviors come since your original model for flocking in 1987?
There has been a lot of subsequent work in many fields, both in science and engineering, which seems to have been inspired by the original boids model. A Google Scholar search indicates my 1987 paper has been cited 1400 times in the academic literature and its title is mentioned on 26,000 Web pages. Because it is visual and intuitive, it is often used as an example when explaining ideas like emergence, complex systems, and artificial life.

Research work that has built on these ideas range from the physics of flocks, to biologically accurate models of real animal species, to autonomous and swarm robotics, to data clustering algorithms, to data visualization, to swarm-based search techniques like Particle Swarm Optimization and Ant Colony Optimization and of course, autonomous character for applications in games and films.

You won an academy award for the flocking behavior you developed for the swarms of bats and armies of penguins in Batman Returns. What were the benefits and challenges of using flocking in the movie?
The citation on my award is worded to include a body of work on several films. Batman Returns was actually the first example of someone other than me using boids in animation production. Within about a week, I got calls from two former Symbolics coworkers asking for the boids code. Only later did I realize this marked the beginning of production on effects for Batman Returns. Andy Kopra made beautifully motion blurred bat swarms at VIFX and Andrea Losch at Boss Films with Paul Ashdown made armies of penguins. I recall Andy talking about how initially the bats were just flying freely around the set. Over time the director imposed more and more constraints on where the bats could be. Originally, they were "free as birds," but by the end he felt like he was holding a gun to their heads and barking directions at them.

Æ

How can game developers use the OpenSteer library in their games? What are the benefits and applications for developers?
OpenSteer is still in beta and development has been slow. In fact, it still has more the structure of sample code than a production-ready library. It does, at least, provide tested implementations of many steering behaviors usable for building autonomous characters. This allows quick construction of NPC controllers that combine behaviors like goal seeking, pursuit, evasion, path following, and obstacle avoidance.

OpenSteer gets downloaded about 500 times a month, whereas the forum gets a handful of new threads per month, which is probably a better gauge of how many people look at it in any detail. Of the people who extend it, many seem to be students doing class projects or thesis research, a few seem to be from the game industry using it for prototyping. Many of these use the existing demo framework as a testbed for their experiments. I know a couple commercial game development teams looked at it in more detail. Generally, they grab sections of code that provide functionality they need, then integrate it into their own game engine.

What benefits are there for game developers in using emergence in their games?
It allows gameplay to be improvised and so more varied, unexpected, and more adaptive to unexpected player actions. Games can be designed with many ways for players to choose to accomplish a goal, a complex world and adaptive characters allow for a variety of gameplay styles. As mentioned above, this makes replaying the game more fun and leads to grist for discussions in a social network of players.

What are the major challenges that game developers face in using emergence in their games?
Whenever the topics of emergent gameplay, autonomous characters, strong AI, large multipath worlds, or frankly anything beside tightly scripted games with characters "on rails"—there is always a concern about *testability* . The same properties that designers of emergent games seek cause nightmares for product testing by removing the repeatability that is so important for testing and debugging. By its nature, an emergent game is not repeatable. If a bug is found during testing, how can it be reproduced? How can the programmer recreate it to observe the symptoms? How can a fix be verified?

In addition to the problems caused by lack of repeatability there is the issue of *coverage* in testing. Before a commercial game is released to the public, testers go to great lengths to make sure that each possible game state has been reached

Æ

and verified. Typically this means the tester has gone through each level, solved every puzzle, taken every alternative. Games based on emergence can not be enumerated this way. How can a game designer have any certainty that players will not get into situations that have never been tested, which could cause the game to crash, or otherwise misbehave in some very bad way (ruin playability, lose stored game state data, or do something that would embarrass or even expose the publisher to legal liability).

Finally, there is the issue of *emergent bugs*. Just as the interaction between many simple game agents can lead to complex global behavior, it can lead to undesirable global *mis*behavior. The magic of emergence produces surprising global behavior like flocking from simple local rules. The 'inverse behavior problem' is very hard, so when a population of agents is not behaving as expected, it can be quite hard to figure out which part of which rule is responsible for the problem. Observed in isolation, each agent may seem to behave correctly. The problem only becomes apparent with hundreds or thousands of interacting agents. How can a programmer set a breakpoint on a symptom that can only be seen while observing a large multiagent simulation?

How do you go about tuning the rules for an emergent system? Do you follow a structured process?

No, and not merely because I have an undisciplined work style. I've never been one to invest a lot of time in software "design" prior to implementation. I tend to build a crude prototype followed by lots of incremental tweaking, with occasional reimplementation of key modules. Moltke the Elder said "no battle plan survives contact with the enemy." Similarly, software design prior to implementation and test is often futile. Certainly for emergent systems, I know of no practical approach other than to repeatedly test and tweak.

Even if one wanted to take a principled approach to emergent system design, the science just does not yet exist to support it. There is no underlying theory to describe emergence. Given a set of local rules, there is no reliable formal way to predict what global behavior will emerge from them. Given an emergent property of a complex system, there is no formal procedure to decompose it into local rules. It is not even clear if there will ever be formal analytical tools for these problems. For example, chaotic systems like turbulent fluid flow are well understood, but are known to be unpredictable. The property of 'sensitive dependence on initial conditions' makes long-range predictions meaningless.

Æ

Many developers are reluctant or unable to try new techniques and approaches in their games. How can we best move forwards to integrate cutting-edge research into new games?
This reluctance to try unproven technology is not surprising given the size and cost of major commercial games these days. When a game can cost ten million dollars to produce, decision making becomes very conservative. The same thing happens in feature film production. So sequels and mashups of previous winners seem the safest choices. Innovation is scary because it is less predicable. I think the most practical approach for introducing new technology is to do it in smaller game productions: indie productions, casual games, cell phone games. . . . With smaller budgets and smaller teams, there can be more tolerance for risk.

There are several kinds of risk beyond the obvious difficulty making a schedule for developing novel untested technology. If the technology is very new, its effect on gameplay can be unpredictable. A novel feature may make the game too easy or too hard, or it may just turn out to be less fun to play than it sounded before it was implemented.

How does your research at Sony Computer Entertainment feed into game development?
Libraries and demos developed in our R&D group are later made available to PlayStation developers. Some of our work is made more widely available as public presentations, academic publications, and as open source software.

What would you recommend to game developers trying to create emergent gameplay?
Start designing the emergent aspects *very* early. Getting such systems to behave as expected can be very difficult and time consuming. You don't want to be tuning the emergence on a deadline with a hundred coworkers waiting for you and delivery deadlines looming.

Summary

Emergence in games is about adding life, realism, and diversity, or at least the illusion thereof, to computer games. Game agents and characters are one of the most important components in games to have these attributes, because they are the part of the game that the players identify with the most. Players understand that the physics in games are restricted, that the world is finite, and that the narrative is often predetermined. However, they look to the characters and agents with the expectation of thought and intelligence, and even of cunning, creativity, and unpredictable behavior. If you can create characters and agents in your games that have the illusion of life, many other limitations will be forgiven.

This chapter looked at a few key components of game characters and agents. In essence, characters and agents must be able to sense, act, and react to the players and the game world. In creating emergence, you look at putting the onus on the environment, making the information readily available for the agents to sense, through probing, broadcasting, influence mapping, or other methods. The more complex the environment and the better the information is synthesized for the agent, the simpler the agent can afford to be. The emergence comes from many simple interactions with other agents and a complex world.

In terms of acting and reacting, there are two types of agents—those that act on their own and those that act in groups. There are many functions of individual agents and many that weren't covered here. Some of the more human-like functions, such as communicating and storytelling, are covered in Chapter 8. What I hoped to provide in this chapter was a simple, flexible model for creating individual agents that can sense and react to their environment, choosing how to react and where to move. With this basic framework, additional information and decisions can be incrementally added to create character models that suit other types of games and characters.

The chapter also covered some of the basics of group behavior, including group movement and tactics. The chapter approached group behavior from a bottom-up, agent-based perspective, where simple, low-level entities interact locally to produce emergent, and seemingly organized, high-level behavior. The agent-based steering behaviors presented can be used in games for a wide range of purposes, from entire gameplay systems to more natural-looking movement for background agents.

Game AI is a very large field and the concepts presented in this chapter are not meant to be all-encompassing, nor are they meant to create the most intelligent and cunning AI ever seen. Rather, I have presented a few basic concepts and examples of how emergence can be incorporated into game characters and agents. More than anything, these ideas should provide food for thought and give you a concrete place to start in creating emergence in your games, and hopefully more natural, lifelike, and diverse behaviors in your characters and agents. Once you have the basic understanding and tools, how you incorporate emergence into your games is up to you, and limited only by your imagination.

Class Exercises

1. In what types of games and for what types of agents would it be best to use each of these sensing methods and why?
   a. Probing
   b. Broadcasting
   c. Influence mapping
2. Choose one of the sensing methods above and devise a simple game that would use this method, including:
   a. What type of game is it and where is it set?
   b. What are the basic agent/character types in this game?
   c. What are the core mechanics? How does the game work?
   d. What are the fundamental player interactions? What do the players do?
3. Think of the agents or characters in your game and the sensing method you have chosen.
   a. What decisions and actions would these agents need to perform?
   b. What information would they need to perform these actions/decisions?
   c. How would they receive and process this information using your chosen sensing method?
4. How static/linear or dynamic/emergent will the behavior of these agents be?
   a. How could their behavior be made more responsive, reactive, dynamic, or emergent?
   b. What additional information would they require to become more emergent?
   c. How suitable is the sensing and processing methods you have devised to creating emergent behavior?
5. Consider the agents in the *Halloween Wars* example.
   a. What additional mechanics, game rules, stances, or steering behaviors could you add to change the gameplay?
   b. How would this change the behavior of the agents and how the game is played? How would you test and tune the game after making these changes? Would their behavior be more or less emergent?
   c. How would you add different types of agents to the game? What different properties and actions would the different agent types have?
   d. Revise the equations to incorporate different agent types.

8 Emergent Narrative

In This Chapter

- Narrative Structure
- Narrative Elements

A game's narrative is the story that is being told, uncovered, or created as the player makes their way through the game. This story might take the form of a single, linear plot that is divulged to the player at selected points in time. Alternatively, it could be the deep, underlying truth of the game world that requires the player to solve puzzles and investigate the world to discover. It could also be the product of the player's interactions in the game world—the internal story that the players create about their characters or challenges as they play the game. No matter the format of the narrative, it is central to the enjoyment and understanding of all games, even games that do not have a story.

People in all cultures teach and learn through storytelling. From a very early age, we are told stories to not only ignite our imagination, but to teach us how to live and behave in the real world. Narrative in games frames the game in a way that the players can understand and reflect upon. It is these stories that have the greatest impact on the players and that they will take with them long after they have played the game. In creating emergent narrative, game developers are tailoring the narrative to the players' experiences and putting the players center stage.

This chapter identifies three narrative paradigms that can be used in games—player as receiver, player as discoverer, and player as creator of the narrative. It is the third paradigm, player as creator, in which emergent narrative lies and so the chapter focuses on expanding this paradigm. The chapter explores two ways to put the players in the role of the creator of an emergent narrative, via storyline and conversations. The storyline is the overarching plot, as well as subplots, that play out in the game. Conversations are a more informal, continuous form of narrative. I discuss how both these narrative elements can be used to create emergent narrative and gameplay and what the implications are for the game players and game developers.

Narrative Structure

If you examine forms of narrative in games from the player's perspective, there are three main categories that can be identified. The first is the traditional "player as receiver" model that is drawn from other forms of storytelling, such as movies and books. In this form, the story is entirely prewritten and is simply transmitted to the player. The player receives the story and has no potential to affect the outcome or progression. A similar type of narrative is "player as discoverer," in which the story is embedded in the game world and the player must uncover the pre-existing plot. The third, and considerably different form, is "player as creator," which involves the player actively creating and affecting the story as a product of his or her actions and interactions. Each of these forms of narrative has been used in previous games with varying degrees of success.

Receiver

Games that put the player in the role of the receiver of narrative simply deliver the plot to the player, usually in installments throughout the game. This can be done in various ways, but usually involves the player being given a piece of the story, followed by a sequence of actions or gameplay, followed by another piece of the story, and so on. In the extreme form, the players are given a pre-rendered cinematic in which they watch a piece of the story unfold, followed by a discrete piece of gameplay, such as a game level. In this form, the story, at best, provides a backstory or motivation for completing the level and what the player actually does in the game has no bearing on how the story plays out. The player as receiver model is the simplest and most linear form of game narrative.

The player as receiver structure is common in first-person shooter games that use cinematics to tie together a series of game levels. For example, in the game *Painkiller*, an introductory cinematic provides players with a backstory that explains their motivations and situation. The players then play through a series of discrete game levels, which are interspersed with cinematics that extend the story and deepen the plot. The cinematics are all pre-scripted and pre-rendered, so they play out the same way no matter how many times the game is played. There are no alternate endings, branches, or player choices. The same format is also used in many action games, adventure games, and other level-based games. Although very simple and entirely linear, this model is used to great effect in *Painkiller* and many games like it, which is why it is so prevalent in current games.

The player as receiver model is also used in many role-playing games, especially for the central storyline. Despite subplots and side-quests that might be happening at the same time, most role-playing games have a central, linear storyline. This central story is usually tied to a particular series of quests. Once the conditions for advancing the main story have been met, a cinematic or a scripted in-game dialogue sequence will play out to give the player the next installment of the story. For example, in *Diablo II*, the players are given a pre-rendered cinematic at the completion of each chapter of the game (see Figure 8.1).

FIGURE 8.1 *Diablo II* plays a pre-rendered cinematic at the completion of each chapter. Diablo II® images provided courtesy of Blizzard Entertainment, Inc.

A similar approach is used in many real-time strategy games, such as *Warcraft III*. In general, games that use the player as receiver structure of narrative require the player to complete a level, chapter, mission, or quest to be rewarded with the next piece of the story and advancement to the next level of the game. The story and gameplay are often not tightly intertwined, with the cinematics acting more as a reward or motivation for the action than a critical part of the gameplay.

Discoverer

Games in which the player is the discoverer of the narrative are still usually very linear and scripted in nature. The pieces of the story might not occur in the same order each time, but the overarching story is linear and the outcome is predetermined. The narrative in player as discoverer games is usually more interactive than in player as receiver games. The players cannot simply wait to be told the story; they must actively try to uncover the plot. This is usually accomplished by talking to game characters, exploring, completing quests, and interacting with the game world. For this reason, the gameplay and narrative is usually more intertwined and interdependent than in player as receiver games.

Due to the interactive nature of the narrative, player as discoverer games often have branching storylines or multiple endings. The players don't just observe a piece of the story playing out; they play a role in it. This might be in the form of choosing dialogue options during a cinematic or having interactive conversations with game characters.

The player as discoverer model is used in interactive fiction (as described in Chapter 3), as well as many role-playing games. In interactive fiction, the player interacts with the game world and characters to sequentially move through the world and story. Until the players discover what to do next, the rest of the game and story is inaccessible to them.

In role-playing games, such as the *Might and Magic* series, the players often need to find the key characters to acquire information from, in order to advance the story. Depending on their interactions with these characters or other player choices, the plot might branch off into different directions.

An example of a branching storyline is in the game *Wing Commander IV*. Depending on the player's choices in the cutscenes, the story plays out differently and the game has different outcomes. The player as discoverer model has not been as successful or as prevalent as the simple and effective player as receiver model.

Creator

In the player as creator model of narrative, the player is creator or co-creator of the game's story. The story is a function of the player's actions and interactions in the game world. Narrative in player as creator games can be generated by the interactions between characters in the game world, the player's interactions in the game world, as well as any knock-on effects of these interactions. In player as creator narrative, the final destination is not important; it is the journey that counts. The player has a definite sense of agency and impact onto the game world, because he or she is actively creating and changing the world and its story.

Games that are generally not considered to have a defined storyline fall into the category of player as creator narrative. This includes simulation games, such as *The Sims* and *SimCity*, strategy games, such as *Total War* and *Civilization*, and other open, sandbox games. In these games, the players use the basic elements of the game, such as buildings, people, and armies, to create their own stories. In *The Sims*, the players are creating a life story for their characters and in *Total War* (see Figures 8.2 and 8.3) they are forging the history of a nation. The players are not discovering or receiving an existing plot, they are creating a new one through the act of playing the game.

FIGURE 8.2 Strategy game *Medieval II: Total War* does not have a defined storyline. © The Creative Assembly. Used with permission.

FIGURE 8.3 An epic, historical battle in *Medieval II: Total War.* © The Creative Assembly. Used with permission.

Some games that do have well-defined, linear storylines can also have elements of player as creator narrative. Games that have large, open game worlds with lots of possibilities for action, such as some role-playing games, can allow the players to create their own subplots. For example, in *The Elder Scrolls IV: Oblivion*, there are many optional quests that can be gained from characters throughout the world. These quests are not connected to the main storyline, do not have to be completed in a specific order, and are entirely optional. Additionally, the players can go adventuring into caves or ruins and fight monsters whenever they feel like it. Players can completely ignore the main storyline, although they won't be able to complete the game by doing so. The players have a fair amount of freedom in creating a unique path for their character through the game. Although the central storyline is linear and they will reach the same final destination, the player has a significant ability to co-create his or her journey through the game.

Player as creator narrative is emergent. Some, or all, of the story is a product of the player's interactions in the game world, interactions between objects or characters in the game world, and knock-on effects. The narrative is not predetermined and scripted; it emerges from interactions between entities in the game world. Emergent narrative does not need to be as complicated or as chaotic as it sounds. A few simple ways to achieve emergent narrative in games, using the narrative elements of storyline and conversation, are described in the next section.

Key Terms

- *Narrative* is the story that is being told, uncovered, or created as the players makes their way through the game.
- *Player as receiver* narrative is delivered to the players in installments as they play the game, usually in the form of cutscenes.
- *Player as discoverer* narrative involves the players actively trying to uncover the plot, by talking to game characters, exploring, completing quests, and interacting with the game world.
- *Player as creator* narrative allows the player to create or co-create the story, which is a function of the player's actions and interactions in the game world.

Narrative Elements

There are two key elements that can be used to create narrative in games—storyline and conversation. Narrative is formed by telling stories about events, people, and places. A player's actions in a game can form a kind of internal narrative, but it is not until the retelling that it becomes a story.

The storyline is the overarching plot, as well as subplots, that play out in the game. As discussed in the previous section, the storyline can be received, discovered, or created by the player. The storyline is often presented in installments, such as pre-rendered or in-game cutscenes, throughout the game. These installments can recap what has happened in the previous section, reveal more depth to thicken the plot, or foreshadow what is yet to come.

Conversations are a more informal, continuous form of narrative. The player can engage in conversations with various characters throughout the game, or observe conversations between other characters, to gain small pieces of information about events, people, and places in the game. By allowing emergence in storylines and conversations in games, you can create emergent narrative.

Storyline

If the player's actions in the game world and interactions with objects and characters are the low-level elements of the game world, the storyline is the high-level behavior. If the story is received (as opposed to discovered or created) by the players, their actions are irrelevant to the overarching storyline, because it is imposed over their actions. There is no connection between the low-level interactions in the game world and the high-level storyline.

For a story that is discovered, the player's interactions are forced to fit the mould of the high-level behavior. This can be considered more of a top-down approach, where the interactions are determined by the high-level design, or storyline. Interactions that are incorrect or not part of the scripted path have no consequence.

In stories that are created, the low-level actions of the player, game world, objects, and characters interact to form the overarching storyline. This is where emergence can occur. The difficulty lies in designing a story system that not only enables emergence, but that makes for compelling, believable, and coherent narrative.

There are various ways in which stories can be constructed or emerge in games. The approach that is right for a given game depends on the constraints and requirements of the game, as well as the desired effect and purpose of the narrative. There are several components of storylines in games that can be used to create a compelling narrative. These components are backstory, storytelling, story creation, and post-game narrative.

A backstory presents events that occurred prior to the start of the game and can be used to establish setting, character, and motivation. Storytelling is used throughout a game to impart additional information about the plot or game world to the player, usually via cutscenes. Story creation is the more interactive form of storytelling in which the player performs certain actions, such as completing missions or quests, to create subplots or advance the overall plot. Finally, post-game narrative is storytelling that occurs after the game is completed, which can be used to create a story out of the player's journey through the game. Each of these components can be used to create narrative in an emergent game, as discussed in the following sections.

Backstory

A backstory presents events that occurred prior to the start of the game and can be used to establish setting, character, and motivation. A backstory is useful in an emergent game as it can provide a story of events leading up to the point at which the game starts. It can provide motivation and setting for the player, without dictating how the game will proceed from the point at which the player takes control.

The backstory defines the game world and characters and identifies the major players in the world, as well their goals and motivations. Even if no further narrative is provided throughout the game, the backstory can go a long way to placing the game within context and adding meaning to the player's experiences in that world.

Telling It Backwards

Another possibility is that the backstory could be revealed as the players progress through the world. Rather than telling the story forwards as the players progress through the game, by artificially imposing a narrative over their actions or running a story alongside the gameplay, the game can tell the story backwards. The players' role in the game could be to piece together something that has already happened, so their actions will not affect how it "plays out," but each discovery will give them insight into the world they are exploring. Therefore, rather than going forwards, the story progresses deeper (into the truth) or backwards (finding out what has happened before).

If the narrative becomes about finding out events that took place before the player entered the game or about uncovering the truth about the game world, there is less need to railroad the player to follow a particular narrative path. The story can also be more directly linked to the players' actions and they can have more freedom in creating their own path through the game. The players' role is to construct the story in the present and future, but what they are discovering has happened in the past.

Hebbian Stories

One model that could be used is to let the players discover pieces of the story in any order. Depending on the pieces that they uncover, other pieces could become invalid, unlocked, or have a lower or higher chance to be uncovered. A neural network model could be used to connect pieces of the story together via weights, rather than having a simple or branching path to follow.

Consider a Hebbian network for example (see Chapter 5); each unit would be a piece of the story (or state of the game world that is associated with a piece of the story) and each piece would be connected via a weight to other pieces or states. Upon the activation of one piece of the story, other pieces of the story could become activated or inhibited (see Figure 8.4).

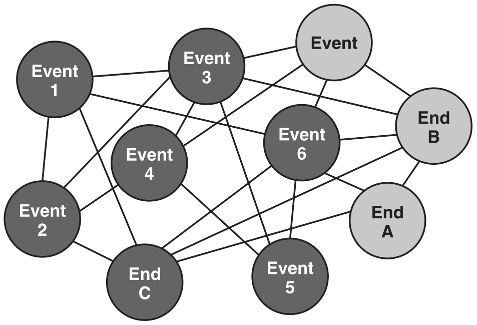


FIGURE 8.4 A Hebbian network for backstory.

In order to provide closure on the story, the activation pattern of the network could result in the players discovering different *end-states*, or truths about the world they are exploring. The end-states themselves would map to nodes in the network, with an end-state being reached when sufficient other nodes have been activated to active an end-state. This could bring about an end to the game, or simply an end to the story that is being told.

Depending on the players' actions in the game, the story would be pieced together in different ways, with the whole picture varying from player to player. As the story being told is the backstory, there is no need to railroad the players along a particular path. Even if they do end up in one of a set of end-states, their journey has been their own. Varying paths can lead to the same end-state in non-linear ways, due to the activation pattern of the neural network.

Key Terms

- *Storyline* is the overarching plot, as well as any subplots, that play out in the game.
- *Backstory* presents events that occurred prior to the start of the game and can be used to establish setting, character, and motivation.
- *Hebbian stories* are based on Hebbian networks, with each unit corresponding to a piece of the story and connected via weights to other pieces. Upon the activation of one piece of the story, other pieces of the story become activated or inhibited.

Storytelling

Storytelling is used throughout a game to impart additional information about the plot or game world to the players, usually via cutscenes. You can view sandbox games as having an entirely emergent storyline, which is created out of the actions of the players, as defined in the player as creator approach. However, this storyline is never actually *told* to the players, it is more of an internal narrative. It also has little creative input from the game designers. Instead, the players are given the tools to create their own stories, without any predefined structure. As such, the narrative is unlikely to follow an ideal story arc, with dramatic tension rising and reaching a climax, followed by a resolution (see Figure 8.5).

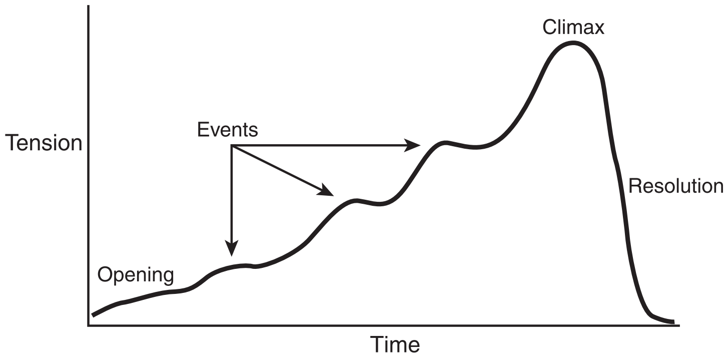


FIGURE 8.5 A traditional story arc.

At the other end of the scale, the story is entirely delivered to the player, who has no role in its creation, as per the player as receiver model of narrative. The game designer has complete control in telling the story, which is likely to be entirely separate from the gameplay. The narrative is kept safe from intervention and derailing by the player's actions, by being encapsulated in pre-rendered or in-game cinematics. The player has no active role in these cinematics, which are delivered to the player between game levels or sections of player freedom and interaction.

Storytelling Continuum

Between these two extremes, there is a graduation between player as the creator and player as the receiver of the narrative. This is where the potential for emergent storytelling in games becomes interesting. Imagine if you were to break a game's story into a number of components or elements. The smaller the components, the more freedom the player has in combining and interacting with the components to create the narrative. With small enough components, the game becomes a sandbox and the narrative is the product of the player's interactions.

The larger the components, the more you have preplanned chunks of story that can be joined together in more predictable ways. With large enough components, the story is one linear storyline that is delivered to the player, or perhaps a branching storyline that the player interacts with in a choose-your-own-adventure style. The size of the components also relates to the player's role in constructing the story. The smaller the components, the more control the player has in constructing the story. The larger the components, the more the designer has pre-constructed the story for the player.

You can draw a continuum from narrative in sandbox games to storylines that are a single component that is broken up into pieces and given to the players at certain points in time. At the sandbox end of the continuum, the pieces are very small and the players assemble them with a great deal of freedom to create stories that are likely to be unique and highly emergent. However, these stories often exist only internally to the players. At the single-storyline end of the continuum, the narrative runs parallel to the gameplay and the game is switched into story-mode from gameplay mode, to give the players the next chapter in the story. The story provides motivation for their actions, but is not tightly intertwined with them. The narrative is entirely told to, and not created by, the players.

Separating Actions from Story

An important consideration is how much the player's actions are separated from the components of the story and how much impact the player has on the story. In sandbox games, the player's interactions are directly pieced together to make the story. However, there is also potential for other stories to be superimposed over the top of the player's actions. These higher-level stories are created from key moments and components of the player's low-level actions (see the "Story Creation" and "Post-Game Narrative" sections).

In the choose-your-own-adventure style story, the player's actions or choices also map directly onto the components of the story. However, only particular actions have any consequence. At key moments, the player makes a choice that will determine the next piece of the story. All their other actions and choices have no impact on the overall story. The player's choice of pieces is also limited to what can come next at the point in time when the choice is given. The story branches from this point in time forwards and the player must choose which path the story will follow. Again, this model can be entirely separate from the player's moment-to-moment actions and the main gameplay.

If you look at game narrative across multiple vertical or hierarchical levels, you can see the player's low-level actions at the bottom and the game's high-level story at the top. How you map these things together is of key importance. From what you have explored so far, you might conclude that:

- The narrative in sandbox games maps directly to the player's actions. However, it is not very effective as a storytelling mechanism and it is implicit, or internal to, the player.
- In single-storyline games, the narrative does not map at all to the player's actions.
- In branching storyline games, the narrative maps directly to the player's actions, but only at key moments. For the most part, the story is separate from the player's actions.

An alternative to the current direct forms of mapping player actions to storyline is to create levels of abstraction between the player's actions and the story that is being told. These additional levels are applicable to both the cases where you are adding a narrative to the player's created story (see the "Story Creation and Post-Game Narrative" section) and the cases where you are giving the players impact on the central narrative of a game.

Key Terms

- *Storytelling* is used throughout a game to impart additional information about the plot or game world to the players.
- *Story arcs* are generated by increasing dramatic tension, culminating in a climax, followed by a resolution.
- *Storytelling continuum* is a graduation between player as the creator and player as the receiver of the narrative.

Emergent Storytelling

Between the extremes of story creation and linear storytelling, the player can be empowered to co-create the central narrative of the game, by impacting how the chunks of story are pieced together. Aristotle defined two core concepts of narrative—Muthos (plot) and Mimesis (mimetic activity). Mimesis includes the actions and behaviors and Muthus is the organization of events to form the overall plot structure.

Aristotle defined Mimesis according to Muthos, so that the structure of the plot determines the actions and behaviors of the characters and events. However, to empower the player and tell a story, you need Muthos and Mimesis, or actions and plot, to behave more like equals. There must be an exchange between low-level actions and high-level plot structure to give the player a role in creating the plot and impacting the world.

If the overarching storyline is determined by the player's cumulative choices and actions in the game, rather than simple switches throughout the game, each decision becomes more important. The world and story become more fluid and the game becomes less rigid and predictable. If the players know that it doesn't matter who they kill or how they talk to most characters in the game, their actions and behavior have no consequence, making the game synthetic and shallow.

You could use a similar plot structure to the one described in Hebbian stories, so that each time a bit of the story is revealed, it feeds into activating or inhibiting other pieces of the story. At the same time, each action and decision the player makes affects the activation of the story network. However, in telling a story forwards, there are likely to be more constraints on the order that pieces of the story can be told and the valid combination of story elements.

Another possibility would be to use a more linear approach of weighted sums and activation thresholds, with a more clearly defined structure of the element dependencies and incompatibilities (see Figure 8.6). The approach would be closer to current methods of telling stories in predetermined chapters, but would have the benefit of being more dependent on the player's actions in the game. Each time the player does something or says something in the game, it would move the game closer or further away to certain plot elements. Plot elements could include a cutscene, the introduction of a new character, a new mission, or a game event. Once the culmination of the player's actions, or the weighted sum, surpasses a plot threshold, the story would be propelled forwards in a given direction.

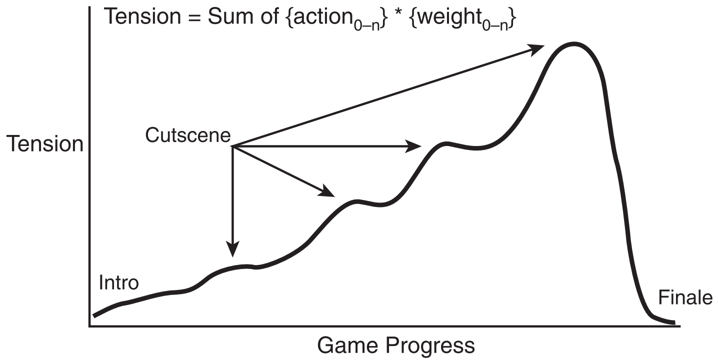


FIGURE 8.6 Emergent storytelling structure.

To attempt to create a more compelling plot structure, or match the emergent storyline more closely with the traditional story arc (see Figure 8.5), you could use a guiding plot algorithm to help the progression follow a pattern of dramatic tension. Based on the length of play, player actions, action-levels of the game, and recent interactions, you can determine when it is time for something to change, something exciting to happen, or some new element to appear in the game. The easiest way to incorporate this *plot-propulsion* factor would be to weight it into the thresholds for moving the plot forwards. Consequently, as the need to propel the plot forwards increases, the thresholds to trigger new plot elements would be reduced.

The plot elements could be a variety of things, such as characters, objects, events, conversation elements, cutscenes, and so on. The size and number of the plot elements will determine the degree of variation and emergence that is possible in the storyline. At one end of the scale, the elements could simply be cutscenes that join together to make a linear story arc. The dependencies would enforce that they occur one after the other and there would no variation in the story. However, the times that they occur in the game could vary to adapt to the player's actions and accommodate the player's pace of playing and the need to advance the story.

In order to increase the potential for variation and emergent storytelling, the components would need to be broken down into smaller parts, with more variability on the order and combination in which they are able to occur. The more the components are broken down, the more potential they have to be rearranged and recombined and the more capacity the game has for emergent storytelling. However, it also becomes possible and more likely that the created story might not make sense or adhere to the requirements for interesting or exciting plot development.

The best approach and the level of variability and emergence that is permitted depends on the type of game that you are making, the gameplay and story that you want to achieve, and how much control you require over the storyline. The open end of the scale could be used to create a loose narrative for sandbox games, while the more restricted end of the scale could add more variation or adaptability to games with linear or branching storylines.

Key Terms

- *Emergent storytelling* empowers the player to co-create the central narrative of the game, by impacting how the chunks of story are pieced together.
- *Plot algorithms* help the plot follow a pattern of dramatic tension, by determining when it is time for something to change, something exciting to happen, or some new element to appear in the game.
- *Plot propulsion factor* weights into the thresholds for moving the plot forwards, reducing the thresholds as the need to propel the plot forwards increases.
- *Plot elements* are basic components that make up the plot, including characters, objects, events, conversation elements, and cutscenes.

Story Creation

Story creation is the more interactive form of storytelling in which players perform certain actions, such as completing missions or quests, to create subplots or advance the overall plot. This story creation does not occur completely randomly or by accident in many games, which are designed to create exciting story moments through conflict or action. The players can also be supported and nurtured in the self-creation of their narrative. Furthermore, stories are not just told through words or movies, they are reflected in everything in the game world. The physical space of the world itself is an important tool in creating a narrative. Storytelling via the physical game world is an ability that is unique to games and harnessing it will provide an effective storytelling medium to amplify the player's sense of agency and centrality in the game world. Finally, you can also make explicit the story a player has created in the game by retelling it. The following sections explain these concepts in detail.

Designing for Story Creation

Many games, such as role-playing games, are designed to allow the players to create and follow branching side-plots and mini-stories during the game. If you look at games like *The Elder Scrolls*, *Neverwinter Nights*, and *Might and Magic*, the world is proliferated with characters, objects, and events that are stories waiting to happen. Every quest the players embark on, every encounter they have, and every character they talk to, adds to the growing story of their game. In these examples, the basic narrative elements that the players are given are large enough to contain small chunks of pre-planned story, but small and independent enough to be combined to make an emergent narrative. The narrative is emergent as the players can create their own combinations of elements and paths through the game. An emergent story in no way needs to be random or unpredictable. The more the game can support the players in authoring an exciting, deep, and dramatic narrative, by providing the tools and elements to do so, the better the player experience will be.

Even if you look to games that are seemingly entirely open, with much smaller elements and less scripted components, you can see that the created story is engaging, as the elements have been designed to facilitate story creation. For example, the elements in *The Sims* game world lend themselves to story creation. There is only a limited set of interactions that are possible with every object and character, but these interactions have been carefully chosen to create dramatic tension and exciting moments. *The Sims* is a streamlined version of life, and what has been left out is as important in creating the overall experience as what has been included in the game.

Supporting Story Creation

In a way, the narrative in games is simply created out of the actions of the players. The story is the story of how they have played the game, what actions they have taken, and how they have shaped and changed the game world. However, for the most part, this story is implicit and internal to the players. The game itself has no model or history of this story, except the current state of the game world that has resulted from a player's series of actions to date.

You can take this story creation to a new level by supporting the players in building and tracking the story of their games. If the game keeps track of the players' progress and path through the game, their story is made explicit, concrete, and real. As the players progress through the game, you can keep track of key interactions, important moments, and interesting statistics, to generate the story of their play. The story can be told concurrently as the players progress through the game or the players can be given a retelling of their journey at the end of the game (see the "Post-Game Narrative" section). The story can also be told to the players in different ways and with varying levels of abstraction.

Journals

Good examples of supporting the players in creating their own stories, as well as providing a medium to retell the story to the players, are the journals, logs, and diaries that are used in many role-playing games. These journals have multiple functions, in that they keep track of the player's current goals (for example, quests or missions), track the player's progress through the game world (for example, world map), and provide an interface to accessing a variety of information and game functions. Simultaneously, the various aspects of game journals form a narrative of where the players have been, what they have done, who they have encountered, what their current understanding of the game world is, and what they're currently trying to achieve.

The game *Elder Scrolls IV: Oblivion* has a very extensive journal that tracks the player's current, active, and completed quests and shows a map of the world and local area, which the players can also use to travel around by clicking on locations. The completed quests log forms a story of what a player has done so far in the game and in what order he or she did it. The active quest log shows all the tasks that the player has done and is yet to do in a single quest line (for example, all the quests for the Mage's Guild). Finally, the current quest log shows the potential paths that the player's story can take next. Depending on which task the player chooses to take on next, he or she will explore different parts of the game world, meet different characters, gain different skills, and unlock new parts of the story.

Oblivion also keeps track of a host of different statistics and counts of things that the players have done in the game. These range from how long the players have been playing, how many times they've upgraded their skills, and how many creatures they've killed, to how many jokes they've told, how many horses they've stolen, and how many diseases they've contracted. These counts provide an interesting summary of the player's actions and effects on the game world. They also provide interesting bites of information that players can share and compare with other players. Trying to increase one of these counts or trying to beat someone else's statistics can provide a goal, or even a game, in itself.

Commentary

Another form of storytelling that is not often acknowledged as narrative is the commentaries in sports games. Commentaries provide a verbal summary and explanation of the action that is taking place in a sports game, as it happens. Commentaries are a very good example of adding narrative to emergent gameplay. The low-level actions involve things like kicking a ball, passing between players, scoring goals, but the story that is created in the commentary is much more than these basic interactions. The commentary tells a story of the game that is in progress, but it also provides content that extends beyond what is currently happening in the game.

If you think about real-world sports commentaries, they include extra information about the background of the players, what's been happening recently in their careers and lives, and the social dynamics between players or teams. All these little bits of information add spice and interest to the commentary (that is, the story that is being told) and a greater context to the game that is being played.

For example, in the *NHL* series of games, there are two main types of commentary: color commentary and play-by-play. Play-by-play describes what is happening at a given time (for example, "Smith passes to Johnson"), based on transcriptions of actual hockey games. The commentator has a set of events they can talk about (for example, fights, shots, passes, and so on) and different emotional states (for example, normal or excited). The closer the puck is to the net, the more excited the commentator sounds. The music and ambient sounds (for example, crowd cheering) are also driven by this emotional state (that is, proximity of puck to net), to increase the excitement of the game at key moments.

In color commentary, the commentator makes a qualitative comment on what is happening (for example, "that was a wicked slap shot!"), based on game statistics. The color commentary gives the commentator an opinion, making them sound more intelligent. The game keeps track of what has been said and the same comment is not triggered again for a long time to avoid repetition.

In *NHL 08*, the commentary focuses on team and player rivalries. Storylines are introduced pre-game and change in-game as a result of major events, such as big hits, injuries, and goals.

Self-Documentation

Some games provide the players with the tools to document and retell their own stories. For example, *The Sims 2* gives the players a photo album tool in which they can capture important moments in their game. The players can look over these moments later and the story they tell of their experiences in the game or the lives of their Sims. *The Sims 2* also includes a movie-making feature, in which the players can film the actions of their Sims. In order to encourage the community to use this feature, Maxis held movie-making competitions on their Web site following the release of *The Sims 2*.

Players of *The Sims 2* can also share their photo albums and movies with friends and other members of the gaming community. People don't only like to review their own stories and gameplay; they also like to share them with others. Players often recount fantastic game experiences to each other, create stories and fiction based on their playing experiences, and share screenshots, movies, and save games with each other via game communities and forums.

The more you can support players in documenting and sharing their game experiences and stories, the easier it is for players to become connected to their characters, games, fellow gamers, and communities. Building a strong game community and social relationships between players keeps them active and interested in your game (see Chapter 9).

Machinima The popularity of end users using game engines to create movies has been growing in recent years, with the spawning of a new form of entertainment, called machinima. In machinima, end users use game engines to create movies and stories that may or may not be related to the actual content of the game. Many games include tools that allow players to easily capture in-game footage, which they can then edit to create movies. Game tools that are used to make machinima include level editors, script editors, cinematic editors, and replay functions. One such tool is the cinematic editor in *Medieval II: Total War*, CinEd (see Figure 8.7), which allows end users to create massive-scale battle movies. The *Total War* engine was used to recreate historical battles in the TV series' *Time Commanders* and *Decisive Battles*.

One of the most famous examples of end-user machinima is *Red vs. Blue: The Blood Gulch Chronicles*, created in the Halo engine. *Red vs. Blue* became so popular that the creators, Rooster Teeth Productions, made a total of 100 episodes over five seasons. The *World of Warcraft* engine is also frequently used for machinima, even featuring in the *South Park* episode "Make Love, Not Warcraft." One of the more famous of the end-user created *World of Warcraft* machimina is the movie *Leeroy Jenkins,* in which a character, Leeroy, manages to single-handedly botch the well-laid plans of his guild members, having been AFK (away from keyboard) during the raid preparations.

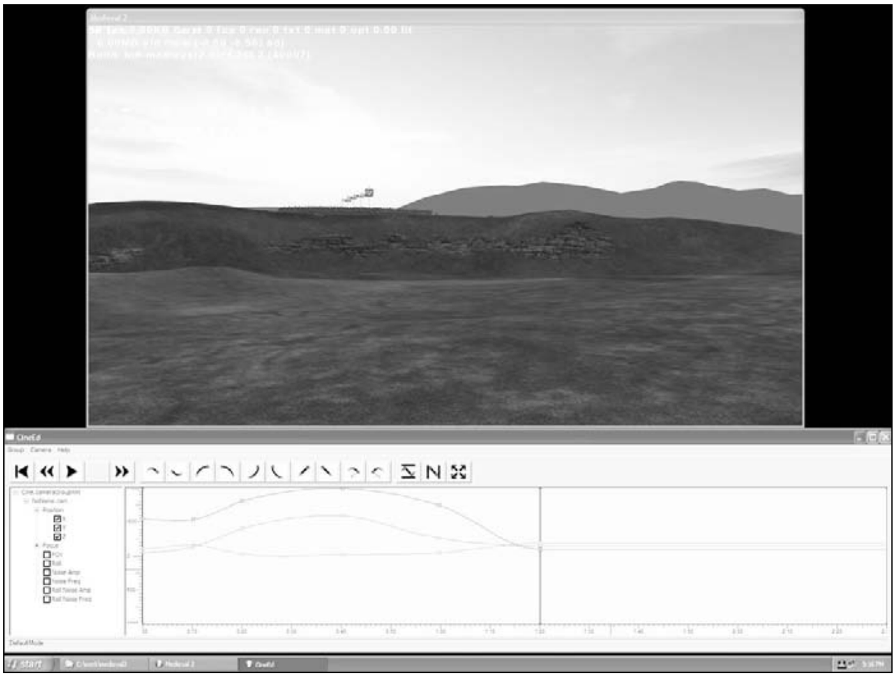


FIGURE 8.7 CinEd, a tool for creating in-game movies in *Medieval II: Total War*.

© The Creative Assembly. Used with permission.

Key Terms

- *Story creation* is a more interactive form of storytelling in which the player performs certain actions, such as completing missions or quests, to create subplots or advance the overall plot.
- *Journals*, logs, and diaries are used in many role-playing games to keep track of the player's current goals, track the player's progress through the game world, and provide an interface to accessing a variety of information and game functions.
- *Commentaries* provide a verbal summary and explanation of the action that is taking place in a sports game, as it happens.
- *Self-documentation tools*, such as photo albums and movie makers, allow players to document and retell their own stories.
- *Machinima* is a media in which end users use game engines to create movies and stories out of in-game footage.

Physical Storytelling

Games have the luxury of not being limited to words on a page or predefined camera angles and sets. Games are dynamic—they can move and change to adapt and suit the player in a variety of ways, ranging from minor, subtle modifications to very evident alterations. One way to tell the story of how the player is impacting the game world is to physically manifest the player's impact in the game world itself. Costume and set changes can be made very simply, cheaply, and quickly in games, on-the-fly.

Game designers already convey most of the information about a game world and its story to the player via the game space. The game space includes the artifacts, characters, music, lighting, scenery, and so on, in the game world. The game space can reflect the changing state of the world and plot, as well as the player's effects on the world, by changing the game space visually, aurally, and physically.

The narrative can be physically manifested in the game world by changing the world as a result of the development of the players, characters, and narrative. As the player develops, the player character's appearance can be changed (for example, to be appear more sinister). The game *Black & White* changes the appearance of the player's creatures to reflect how it is developing and changing to great effect.

The world can also be changed, by making changes to the ambient lighting to reflect the mood (for example, darker times) or swapping in scenery, objects, and characters. Game characters can be changed to look different and alter their reactions to the player, via expressions, body language, voice, and behavioral responses.

The physical space of the game is the "set" where the game's narrative plays out. Changing the mood, look, ambience, and music of the space can have a dramatic impact on the story and feeling of the game. Using the physical space to tell the story is an accessible and effective way to support an emergent narrative. One such example is the use of generative music in games.

Generative Music

Generative music is music that is procedurally or algorithmically generated or altered using computer software. Similar to the other types of emergent systems discussed in this book, generative music can have varying degrees of structure and constraints imposed by audio designers or composers. The fewer rules and restrictions set for the music system, the more variation and randomness that will be displayed in the music. Conversely, the more structure and behavior that is predetermined, the less variation the generated music will exhibit.

In a game scenario, it is likely that the music will be required to convey a particular mood or atmosphere. Consequently, to create generative music for a game, the composer will most likely need to define most of the structure and parameters for the music, allowing only minor variations to be made by the music engine in-game. The type of music that will be generated and the degree of control that the composer requires depends on the gameplay and the desired effect of the music. Games that make use of generative music include *Creatures 2* and *Spore*.

In *Creatures 2*, the composer, Peter Chilvers, designed the music engine to score the emotion of the game as it is being played. Each of the major areas in the game (for example, swamp, volcano, terrariums, and laboratories) has its own atmospheric music, but within each setting, the music can vary according to mood of the creatures and level of threat. Each composition has a set of players and each player has a set of instructions for responding to the mood and threats in the game. For example, the music becomes softer when the creature is sad and harsher when a threat is present. A script controls the music engine, setting the volume, panning, and interval between notes as the mood and threat changes.

Brain Eno is working on a generative music score for *Spore*, to create sound for the game that is as procedural as the game itself. The music in *Spore* is generative, so that players won't have the same musical experience in a particular part of the game at any moment. As the players progress through the game and explore different parts, their choices will impact the music in various ways. The landscape of the game space also affects the music that is playing. A piece of software used in *Spore*, called The Shuffler, procedurally generates fragments for the soundtrack from a number of samples. In *Spore*, the players won't hear the same music repeatedly, whereas most game music is based on a prewritten set of tracks looping.

Additional Reading

For further information on generative music:

- Gameware Development. The Music Behind Creatures. Online at: http://www.gamewaredevelopment.co.uk.
- Wooller, R., Brown, A., Miranda, E., Berry, R., and Diederich, J. (2005) A Framework for Comparison of Processes in Algorithmic Music Systems. *Generative Arts Practice*. Sydney, Australia: Cognition Studios Press, pp. 109–124.
- Wright, W., and Eno, B. (2006) Playing with Time. The Long Now Foundation. Online at: http://www.longnow.org/projects/seminars/.

Key Terms

- *Physical storytelling* involves physically manifesting the narrative in the game world, by changing the world as a result of the development of the player, characters, and narrative.
- *Generative music* is music that is procedurally or algorithmically generated or altered using computer software.

Plot Generation

So far, you've learned about tracking, narrating, counting, or recording the player's low-level actions to create a story that retells exactly what the player has done. Although this keeps track of the players' progress through the game and gives them a detailed account of what they have done and accomplished, it might not make for an interesting or well-crafted story. You've moved from taking the player's internal or implicit story and making it explicit, but it's still not the same as a well-written narrative.

In addition to tracking and recounting all the player's low-level actions and each detail of their journey, you can look at abstracting and refining these details to give a more streamlined account of their story and their impact on the game world. You can remove a lot of the unimportant actions and inconsequential choices to extract and generate a main plot for the game. This is similar to the emergent storytelling approach discussed previously, except that you're not starting with any predefined plot elements. All you have are the low-level actions and interactions of the players and a structure for generating a plot (Muthos) from their actions (Mimesis).

Perspective

In creating a story out of the player's actions, you need to be able to look across their actions and interactions and find key moments of interest and trends that can be added to a story. A story is just a series of events told from a particular perspective. Although the perspective or filter you use to tell the story might not align exactly with how the player saw the events occur and unfold, it can be entertaining and interesting all the same.

The story could be told from the perspective of a third party, onlooker, or disembodied narrator that has no connection to the player. The perspective that is chosen will determine a lot about the type of story that is told and the events and interactions that are important. For example, in a city building game, the story could be told from the perspective of the mayor, a citizen, or a separate narrator. Each story would be different, with varying elements being important and an alternate focus.

In creating a generated storyline, you should first choose the perspective from which the story will be told and who or what type of character will tell the story. The more you know about this character and their perspective, the more you can identify what will be important to them about the events that are unfolding and what events and actions you should track. Different characters, with varying motivations and backgrounds, will also interpret the same events in different ways. Their preferences, experiences, and biases, will also affect how they will retell the story. Potential storytelling characters include:

- *The player*—The story is told from the perspective of the player, such as a journal (for example, "I talked to," "I discovered," or "I was attacked by").
- *A game character*—The story is told from the perspective of a non-player character in the game. The story could be told directly about the actions of the player (for example, "I watched as he entered the guild") or indirectly about how the player's actions affect their life (for example, "Due to the increases in taxes, I was unable to pay my rent this month").
- *A disembodied narrator*—The story is told from the perspective of an all-seeing eye that was not physically present in the game. The narrator could be an impartial observer who simply recounts the player's journey without bias or a character that will interpret and spin the player's actions in a particular way, based on their own beliefs and characteristics.

Event Tracking

After you have identified the perspective from which the story will be told, you can identify the elements that you will need to track while the players are playing the game. These elements will also need to be important and interesting to the players and will most likely relate to the core concepts of the gameplay.

For example, if the game is a first-person shooter, the elements to track will most likely relate to who or what the players have killed, how frequently they use each type of weapon, what kind of tactics they use, how often they pick up armor or health packs, and enemies they had difficulty dispatching.

In a strategy game, you could keep track of how many units of each type the players constructed, how much resources they allocated to researching, upgrading, defending, and attacking, how many battles they won, how large an attacking force they used to raid an opponent's base, and what kind of buildings they destroyed.

Once you have all these basic statistics, you can use them to generate timelines of the gameplay or to identify trends. You could trigger story elements based on the number of times they did a particular thing, the ratio between different behaviors, or identifying a series of events as a strategy. How the basic numbers are combined or filtered to identify patterns, behaviors, and interesting story elements depends entirely on the specific gameplay and the type of story that is being told. A good place to start is to watch people playing your game and the types of stories that seem to naturally emerge from their play. How would they describe how the game played out or how would you tell a story about it?

Medium

Now that you know who will be telling the story and what the content will be based on, it is also necessary to decide how the story will be told. The story could be conveyed to the player in a written form (for example, a journal, log, diary, book, notes), in a movie form (for example, in-game cutscenes that are generated at various points in the game), or verbally (for example, a narrator that tells the story). The story could also be told as the player progresses through the game or as a summary at the end of the game (see the "Post-Game Narrative" section).

The stories generated from a first-person shooter game could range from blow-by-blow descriptions of each individual fight the player was involved in to colorful narratives generated about the player's path through a level or the whole game. Stories could be generated by:

- Using prewritten sentences that can be selected and pieced together—Describing a fight by selecting the text that describes each of their moves and the enemy's responses or substituting words (for example, "shot," "leg," "five points of damage") into prewritten sentences.
- Having movies or narratives that relate to particular patterns of play—Analyzing the player's statistics and choosing a prewritten story or movie that fits their behavior.
- Generating a story to fit a prewritten structure based on key statistics—A statement about how many monsters they killed, followed by a reference to their use of a particular weapon, and then a description of an important battle, and so on.

A more adventurous option would be to have an in-game cinematic that can call on particular models, behaviors, and sequences to be generated on-the-fly based on the player's game. It could show the player's character performing key actions or reliving important moments in the game. The player's actions and choices could be encapsulated in a succinct cutscene that recaps the whole game or a section of the game.

For example, the cutscene could show that when the player killed a key character it set particular events in motion. Or when players were faced with a particular decision, their choice had the following consequences. Recapping the player's actions and choices in this way would provide a sense of consequence and importance to their actions and decisions. Additionally, generating moving sequences in-game is quite easy and cheap to do in many game engines. With the ability to swap in assets and the right scripting and logic behind the generation, dynamic cutscenes generated in-game could be fairly straight-forward and highly effective.