Let Run your Neurons
team
LRN

AI GAME ENGINE PROGRAMMING

- Provides a detailed guide for programmers interested in creating an AI Engine for any game genre
- Breaks down AI elements and solutions by genre, and provides concrete examples from popular games
- Explains distributed AI as a paradigm that can help with the organization of almost any AI engine
- Includes a companion CD-ROM with all the code implementations for both basic and complex AI techniques and useful AI resources code

Game Programming Series

BRIAN SCHWAB

AI Game Engine Programming

Brian Schwab

CHARLES RIVER MEDIA, INC.
Hingham, Massachusetts

TeamLRN

Copyright 2004 by CHARLES RIVER MEDIA, INC.
All rights reserved.

No part of this publication may be reproduced in any way, stored in a retrieval system of any type, or transmitted by any means or media, electronic or mechanical, including, but not limited to, photocopy, recording, or scanning, without prior permission in writing from the publisher.

Publisher: Jenifer Niles
Cover Design: The Printed Image

CHARLES RIVER MEDIA, INC.
10 Downer Avenue
Hingham, Massachusetts 02043
781-740-0400
781-740-8816 (FAX)
info@charlesriver.com
www.charlesriver.com

This book is printed on acid-free paper.

Brian Schwab. *AI Game Engine Programming.*
ISBN: 1-58450-344-0

All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks should not be regarded as intent to infringe on the property of others. The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products.

Library of Congress Cataloging-in-Publication Data
Schwab, Brian.
AI game engine programming / Brian Schwab.
p. cm.
ISBN 1-58450-344-0 (pbk. with cd-rom : alk. paper)
1. Computer games—Programming. 2. Artificial intelligence. I. Title.
QA76.76.C672S39 2004
794.8'1526—dc22

2004016330

Printed in the United States of America
04 7 6 5 4 3 2 First Edition

CHARLES RIVER MEDIA titles are available for site license or bulk purchase by institutions, user groups, corporations, etc. For additional information, please contact the Special Sales Department at 781-740-0400.

Requests for replacement of a defective CD-ROM must be accompanied by the original disc, your mailing address, telephone number, date of purchase, and purchase price. Please state the nature of the problem, and send the information to CHARLES RIVER MEDIA, INC., 10 Downer Avenue, Hingham, Massachusetts 02043. CRM's sole obligation to the purchaser is to replace the disc, based on defective materials or faulty workmanship, but not on the operation or functionality of the product.

To Harley:
Run free, you Jerky.

TeamLRN

Contents

Preface xxvii

Part I Introduction 1

1 Basic Definitions and Concepts 3

What is Intelligence? 3

What is "Game AI"? 4

What Game AI is Not 7

How This Definition Differs from That of Academic AI 9

Applicable Mind Science and Psychology Theory 11

Brain Organization 11

Knowledge Base and Learning 12

Cognition 15

Theory of Mind 17

Bounded Optimality 24

Lessons from Robotics 25

Summary 28

2 An AI Engine: The Basic Components and Design 29

Decision Making and Inference 30

Types of Solutions 30

Agent Reactivity 30

System Realism 31

Genre 31

Content 32

Platform 32

TeamLRN

Development Limitations 34
Entertainment Limitations 36
Input Handlers and Perception 37
Perception Type 37
Update Regularity 37
Reaction Time 38
Thresholds 38
Load Balancing 38
Computation Cost and Preconditions 38
Navigation 40
Grid Based 40
Simple Avoidance and Potential Fields 41
Map Node Networks 42
Navigation Mesh 43
Combination Systems 45
Bringing It All Together 46
Summary 47

3 Alsteroids: Our AI Test Bed 49
The `GameObj` Class 49
The `GameObj` Update Function 51
The `SHIP` Object 52
The Other Game Objects 54
The `GameSession` Class 55
Primary Logic and Collision Checking 57
Object Cleanup 58
Spawning Main Ship and Powerups 59
Bonus Lives 60
End of Level and Game 60
The `Control` Class 61
The AI System Hooks 61

Game Main Loop 63
Summary 63

Part II Game Genres 65

4 Role Playing Games (RPGs) 67
Common AI Elements 72
Enemies 72
Bosses 73
Nonplayer Characters (NPCs) 74
Shopkeepers 74
Party Members 75
Useful AI Techniques 77
Scripting 77
Finite-State Machines (FSMs) 78
Messaging 78
Examples 79
Exceptions 80
Specific Game Elements That Need Improvement 81
Role-playing Does Not Equal Combat 81
Grammar Machines (GMs) 82
Quest Generators 82
Better Party Member AI 83
Better Enemies 84
Fully Realized Towns 85
Summary 86

5 Adventure Games 87
Common AI Elements 88
Enemy AI 88
Nonplayer Characters (NPCs) 89

TeamLRN

Cooperative Elements 89
Perception Systems 90
Camera 90
Useful AI Techniques 90
Finite-State Machines (FSMs) 90
Scripting Systems 91
Messaging Systems 91
Fuzzy Logic 91
Examples 92
Areas That Need Improvement 93
Additional Types of Stealth Goals 93
A Return to Traditional Adventure Roots 93
Better NPC Communication 94
User Interface 94
Summary 94

6 Real-Time Strategy (RTS) Games 97
Common AI Elements 97
Individual Units 97
Economic Individual Units 98
Commanders and Medium-level Strategic Elements 99
High-level Strategic AI 99
Town Building 100
Indigenous Life 100
Pathfinding 100
Tactical and Strategic Support Systems 101
Useful AI Techniques 103
Messaging 103
Finite-State Machines (FSMs) 103
Fuzzy-State Machines (FuSM) 103
Hierarchical AI 104

Planning 104
Scripting 105
Data-Driven AI 105
Examples 107
Areas That Need Improvement 108
Learning 108
Determining When an AI Element Is Stuck 108
Helper AI 109
Opponent Personality 109
More Strategy, Less Tactics 110
Summary 111

7 First-Person Shooters/Third-Person Shooters (FTPS) 113
Common AI Elements 116
Enemies 116
Boss Enemies 116
Deathmatch Opponents 116
Weapons 117
Cooperative Agents 117
Squad Members 118
Pathfinding 118
Spatial Reasoning 119
Useful AI Techniques 119
Finite-State Machines (FSMs) 119
Fuzzy-State Machines (FuSMs) 123
Messaging Systems 124
Scripting Systems 124
Examples 124
Areas That Need Improvement 125
Learning and Opponent Modeling 125
Personality 126

TeamLRN

Creativity 127
Anticipation 127
Better Conversation Engines 127
Motivations 128
Better Squad AI 128
Summary 128

8 Platform Games 131
Common AI Elements 137
Enemies 137
Boss Enemies 137
Cooperative Elements 138
Camera 138
Useful AI Techniques 139
Finite-State Machines (FSMs) 139
Messaging Systems 139
Scripted Systems 140
Data-Driven Systems 140
Examples 140
Areas That Need Improvement 141
Camera Work 141
Help Systems 141
Summary 142

9 Shooter Games 145
Common AI Elements 150
Enemies 150
Boss Enemies 151
Cooperative Elements 152
Useful AI Techniques 152
Finite-State Machines (FSMs) 152

Scripted Systems 152
Data-Driven Systems 153
Exceptions 153
Examples 153
Areas That Need Improvement 155
Infusion of Actual AI 155
Summary 155

10 Sports Games 157
Common AI Elements 158
Coach- or Team-Level AI 158
Player-Level AI 159
Pathfinding 160
Camera 161
Miscellaneous Elements 161
Useful AI Techniques 163
Finite-State Machines (FSMs) and Fuzzy-State Machines (FuSMs) 163
Data-Driven Systems 170
Messaging Systems 171
Examples 171
Areas That Need Improvement 172
Learning 172
Game Balance 173
Gameplay Innovation 173
Summary 174

11 Racing Games 177
Common AI Elements 179
Track AI 179
Traffic 180
Pedestrians 181

TeamLRN

Enemy and Combat 181
Nonplayer Characters 181
Other Competitive Behavior 182
Useful AI Techniques 182
Finite-State Machines (FSMs) 182
Scripted Systems 182
Messaging Systems 182
Genetic Algorithms 183
Examples 183
Areas That Need Improvement 183
Other Areas of Interest Than Crime 184
More Intelligent AI Enemies 184
Persistent Worlds 184
Summary 185

12 Classic Strategy Games 187
Common AI Elements 198
Opponent AI 198
Helper AI 198
Useful AI Techniques 199
Finite-State Machine 199
Alpha-Beta Search 199
Neural Nets (NNs) 200
Genetic Algorithms (GAs) 200
Exceptions 200
Examples 201
Areas That Need Improvement 201
Creativity 201
Speed 201
Summary 201

13 Fighting Games 203
Common AI Elements 205
Enemies 205
Collision Systems 206
Boss Enemies 206
Camera 206
Action and Adventure Elements 207
Useful AI Techniques 207
Finite-State Machines (FSMs) 207
Data-Driven Systems 207
Scripting Systems 208
Examples 208
Areas That Need Improvement 209
Learning 209
Summary 209

14 Miscellaneous Genres of Note 211
Civilization Games 211
God Games 220
War Games 223
Flight Simulators (Sims) 229
Rhythm Games 233
Puzzle Games 234
Artificial Life (Alife) Games 235

Part III Basic AI Engine Techniques 239

15 Finite-State Machines 241
FSM Overview 241
FSM Skeletal Code 245
The FSMState Class 246

TeamLRN

The FSMMachine Class 247
The FSMAIControl Class 249
Implementing an FSM-Controlled Ship into Our Test Bed 250
Example Implementation 251
Coding the Control class 252
Coding the States 254
Performance of the AI with This System 264
Pros of FSM-Based Systems 266
Cons of FSM-Based Systems 267
Extensions to the Paradigm 268
Hierarchical FSMs 268
Message- and Event-Based FSMs 268
FSMs with Fuzzy Transitions 269
Stack-Based FSMs 270
Multiple-Concurrent FSMs 270
Data-Driven FSMs 271
Inertial FSMs 272
Optimizations 273
Load Balancing Both FSMs and Perceptions 273
Level of Detail (LOD) AI Systems 274
Shared Data Structures 275
Design Considerations 276
Types of Solutions 276
Agent Reactivity 276
System Realism 276
Genre 277
Content 277
Platform 277
Development Limitations 278
Entertainment Limitations 278
Summary 278

16 Fuzzy-State Machines (FuSMs) 281
FuSM Overview 281
FuSM Skeletal Code 285
The `FuSMState` Class 286
The `FuSMMachine` Class 288
The `FuSMAIControl` Class 289
Implementing an FuSM-controlled Ship into Our Test Bed 290
Example Implementation 291
A New Addition, The `Saucer` 291
Other Game Modifications 291
The FuSM System 292
Coding the Control Class 293
Coding the Fuzzy States 296
Performance of the AI with This System 300
Pros of FuSM-Based Systems 302
Cons of FuSM-Based Systems 303
Extensions to the Paradigm 304
FuSMs with a Limited Numbers of Current States 304
An FuSM Used as a Support System for a Character 302
An FuSM Used as a Single State in a Larger FSM 305
Hierarchical FuSMs 305
Data-Driven FuSMs 305
Optimizations 306
Design Considerations 306
Types of Solutions 306
Agent Reactivity 307
System Realism 307
Genre 307
Platform 308
Development Limitations 308
Entertainment Limitations 308
Summary 308

TeamLRN

17 Message-Based Systems 311
Messaging Overview 311
Messaging Skeletal Code 313
The Message Object 313
The `MessagePump` 314
Client Handlers 318
Example Implemation in Our AIsteroids Test Bed 319
The `MessState` Class 320
The `MessMachine` class 320
The `MessAIControl` Class 321
Coding the States 327
Performance of the AI with This System 329
Pros of Messaging Systems 330
Cons of Messaging Systems 331
Extensions to the Paradigm 332
Message Priority 332
Message Arbitration 332
Automatic and Extended Message Types 333
Optimizations 333
Design Considerations 334
Types of Solutions 334
Agent Reactivity 334
System Realism 334
Genre and Platform 335
Development Limitations 335
Entertainment Limitations 335
Summary 335

18 Scripting Systems 337
Scripting Overview 337
Example Implementation in Our AIsteroids Test Bed 339
A Configuration Script System 339

Performance of the AI with This System 345
Extensions to the Configuration Script Paradigm 346
Embedding Lua 346
Lua Overview 346
Lua Language Fundamentals 347
Integration 350
Example Implementation in the AIsteroids Test Bed 355
Performance of the AI with This System 360
Pros of Scripting Systems 360
Cons of Scripted Systems 362
Extensions to the Scripting Paradigm 364
Completely Custom Languages 365
Built-In Debugging Tools 365
A Smart IDE for Writing Scripts 366
Automatic Integration with the Game 366
Self-Modifying Scripts 367
Optimizations 367
Design Considerations 368
Types of Solutions 368
Agent Reactivity 368
System Realism 369
Development Limitations 369
Entertainment Limitations 370
Summary 370

19 Location-Based Information Systems 373
Location-Based Information Systems Overview 373
Influence Maps (IMs) 373
Smart Terrain 375
Terrain Analysis (TA) 375
How These Techniques Are Used 376
Occupance Data 376

TeamLRN

Ground Control 377
Pathfinding System Helper Data 377
Danger Signification 378
Rough Battlefield Planning 378
Simple Terrain Analysis 378
Advanced Terrain Analysis 379
Influence Mapping Skeletal Code and Test Bed Implementation 381
The `OccupanceInfluenceMap` 387
Uses within the Test Bed for an Occupance IM 392
The `ControlInfluenceMap` 393
Uses within the Test Bed for a Control-Based IM 396
The `BitwiseInfluenceMap` 397
Uses within the Test Bed for a Bitwise IM 403
Other Implementations 404
Pros of Location-Based Information Systems 405
Cons of Location-Based Information Systems 406
Extensions to the Paradigm 406
Optimizations 406
Design Considerations 407
Types of Solutions 407
Agent Reactivity 407
System Realism 408
Genre and Platform 408
Development Limitations 408
Entertainment Limitations 408
Summary 408

Part IV Advanced AI Engine Techniques 411

20 Genetic Algorithms 413
Genetic Algorithms Overview 413
Evolution in Nature 413
Evolution in Games 415

Basic Genetic Method 416
Initialize a Starting Population of Individuals 416
Evaluate Each Individual's Success within the Problem Space 416
Generate New Individuals Using Reproduction 417
Representing the Problem 417
The Gene and Genome 417
The Fitness Function 420
Reproduction 421
Implementing a Genetic Algorithm System into the AIsteroids Test Bed 428
Performance within the Test Bed 442
Pros of Genetic Algorithm-Based Systems 445
Cons of Genetic Algorithm-Based Systems 446
Extensions to the Paradigm 448
Ant Colony Algorithms 448
Coevolution 449
Self-Adapting GAs 449
Genetic Programming 449
Design Considerations 450
Types of Solutions 450
Agent Reactivity 450
System Realism 450
Genre 451
Platform 451
Development Limitations 451
Entertainment Limitations 451
Summary 452

21 Neural Networks 453
Neural Nets in Nature 453
Artificial Neural Nets Overview 455
Using a Neural Net 458
Structure 458

TeamLRN

Learning Mechanism 459
Creating Training Data 460
An Aside on Neural Network Activity 461
Implementing a Neural Net within the AIsteroids Test Bed 464
The `NeuralNet` Class 465
The `NLayer` Class 470
The `NNAIControl` Class 474
Performance within the Test Bed 480
Optimization 481
Pros of Neural Net-Based Systems 482
Cons of Neural Net-Based Systems 483
Extensions to the Paradigm 484
Other Types of NNs 485
Other Types of NN Learning 486
Design Considerations 487
Types of Solutions 487
Agent Reactivity 487
System Realism 488
Genre and Platform 488
Development Limitations 488
Entertainment Limitations 488
Summary 489

22 Other Techniques of Note 491
Artificial Life 491
Artificial Life Usage in Games 492
Artificial Life Disciplines 492
Pros 495
Cons 495
Areas for Exploitation within Games 495
Planning Algorithms 496
Current Usage in Games 497

Pros 499
Cons 499
Areas for Exploitation within Games 500
Production Systems 500
Pros 502
Cons 503
Areas for Exploitation within Games 503
Decision Trees 503
Pros 505
Cons 506
Areas for Exploitation within Games 506
Fuzzy Logic 507
Pros 509
Cons 509
Areas for Exploitation within Games 509
Summary 510

Part V Real Game AI Development 513

23 Distributed AI Design 515
Basic Overview 515
A Real Life Example 515
The Distributed Layers 517
The Real Life Example Revisited 517
The Perceptions and Events Layer 519
The Behavior Layer 519
The Animation Layer 521
The Motion Layer 524
Short-Term Decision Making (ST) 525
Long-Term Decision Making (LT) 525
Location-Based Information Layer 526

TeamLRN

Brooks Subsumption 527
Game Breakdown 527
Goals 527
Distributed *Super Mario Bros.* 528
AI Enemies Implementation 529
AI Player Implementation 534
Summary 538

24 Common AI Development Concerns 541
Design Considerations 541
Concerns with Data-Driven AI Systems 542
The One-Track Mind Syndrome 544
Level of Detail (LOD) AI 545
Support AI 548
General AI Design Thinking 550
Entertainment Considerations 551
The All Important Fun Factor 551
Perceived Randomness 553
Some Things That Make an AI System Look Stupid 554
Production Concerns 555
Coherent AI Behavior 556
Thinking about Tuning Ahead of Time 556
Idiot Proof your AI 557
Consider Designer Used Tools Differently 558
Summary 558

25 Debugging 559
General Debugging of AI Systems 559
Visual Debugging 559
Provides a Variety of Information 560

Helps with Both Debugging and Tuning 560
Timing Information 560
Watching for State Oscillation 561
Useful for Console Debugging 561
Debugging Scripting Languages 561
Double Duty Influence Mapping 561
Widgets 562
Implementation 562
`BasicButton` 566
`Watcher` 567
`RadioButton` 568
`OnOffButton` 568
`ScrubberWidget` 569
Integration within a Program 570
Summary 575

26 Conclusions, and the Future 577
What Game AI Will Be in the Future 578

Appendix: About the CD-ROM 581

Index 583

TeamLRN

Preface

There are not many books on general game programming, and even fewer on game artificial intelligence (AI) programming. This text will provide the reader with four principal elements that will extend the current library.

1. *A clear definition of "game AI."* Many books use a general or far too wide sweeping meaning for the term *AI*, and as such, the reader never feels completely satisfied with the solutions provided and may also further the sad, "mystical" nature of AI that pervades the common knowledge of both the general public and industry people. This also allows for a means of rethinking your game AI problems into separate pieces that will then naturally fit into real AI solutions.
2. *Genre-by-genre breakdown of AI elements and solutions.* Too many books rely on one type of game, or one narrow demonstration program. This text will break apart the majority of the modern game genres and give concrete examples of AI usage in actual released titles. By seeing the reasoning behind the different genres choices of AI paradigms, the reader will gain greater understanding of the paradigms themselves.
3. *Implemented code for the majority of commonly used AI paradigms.* In the later parts of the book, real code will be given for each AI technique, both in skeletal form, and as part of a real-world example application. The code will be broken down and fully discussed, to help show the actual handling of the system.
4. *A discussion of future directions for improvement.* With each genre and AI technique, the text will give examples of ways the system could be extended. This will be done by pointing out common AI failings in current and classic games, as well as by detailing ways in which systems could be optimized for space, speed, or some other limitation.

TeamLRN

OVERVIEW OF THE BOOK

Part I of the book provides an overall look at game AI, covers the basic terminology that will be used throughout the book, looks at some of the underlying concepts of game AI, and dissects the parts of a game AI engine. Part II covers specific game genres and how they use the differing AI paradigms. Although the book cannot be all-inclusive (by detailing how each and every game "did it"), it will discuss the more common solutions to the problems posed by games of each genre. Part III provides the actual code implementations for the basic AI techniques, and Part IV covers the more advanced ones. In Part V, a variety of concepts and concerns are broken down, dealing with "Real Game AI Development": general design and development issues, "Distributed AI" as an overall paradigm that can help with the organization of almost any AI engine, debugging AI systems, and the future of AI.

AUDIENCE

This book was written to provide game developers with the tools necessary to create modern game Artificial Intelligence (AI) engines, and to survey the capabilities of the differing techniques used in some current AI engines. AI programming is a very challenging aspect of game production, and although many books have been written on generic game-related data structures and coding styles, very few have been written specifically for this important and tech-heavy subject.

This book is specifically written for the professional game AI programmer, or the programmer interested in expanding his area of interest into AI. If you are having difficulties determining which techniques to use, have questions about, or need working code for the engine best suited for a particular game, this is the book for you. This book will provide a clean, usable interface for a variety of useful game AI techniques. The book will emphasize primary decision-making paradigms, and as such will not delve into the important areas of pathfinding (at least, not directly; many of the techniques presented could be used to run a pathfinder) or perception, although they will be discussed.

This book assumes a working knowledge of C++, the classical data structures, and a basic knowledge of object-oriented programming. The demonstration programs are written in Microsoft Visual C++® under the Windows® platform, but

only the rendering is platform specific, and the rendering API used is the GLUT extension to OpenGL, so that you could easily port to another system if necessary. See the CD-ROM for information on GLUT and OpenGL.

After reading this book, you will be familiar with a good portion of the huge landscape of knowledge that a game AI programmer has to master. The genre discussions will supply the programmer with insights into how to build an AI system, from start to finish, given the realities of the product and the schedule. The code in the book is generic enough to build almost any type of AI system and provides clear ways to combine techniques into much more complex and usable game-specific AI engines.

TeamLRN

Part I Introduction

In Chapter 1, the phrase "game AI" will be defined, and the more ambiguous uses of the expression will be reassigned to more appropriate language. Some applicable theory will also be discussed, both from the psychological world and from the academic artificial intelligence (AI) world.

In Chapter 2, the foundation systems that make up a general game AI engine will be discussed, along with the elements that need to be considered when designing a new AI engine.

In Chapter 3, the basic example application that later parts of the book will use as an AI implementation test bed will be documented and discussed.

TeamLRN

1 Basic Definitions and Concepts

Welcome to *AI Game Engine Programming*. This book is meant to give the game artificial intelligence (AI) programmer the knowledge and tools needed to create AI engines for modern commercial games. But what exactly do we mean by "game AI"? AI as a science is relatively young; some of the earliest work was done in the early 1950s. Games have used real AI techniques for an even shorter time because of the computation and storage space limitations of earlier game machines. Because AI is a rather new concept in games, the definition of game AI is not clear for most people, even those who practice game production. This chapter will define the term *game AI,* identify practices and techniques that are commonly mistaken for game AI, and discuss areas of future expansion. Later in the chapter, relevant concepts from other fields, including mind science, psychology, and robotics, will be discussed regarding game AI systems.

WHAT IS INTELLIGENCE?

The term *intelligence* is fairly nebulous. The dictionary will tell you it is *the capacity to acquire and apply knowledge,* but this is far too general. This definition, interpreted literally, could mean that your thermostat is intelligent. It acquires the knowledge that the room is too cold and applies what it learned by turning on the heater. The dictionary goes on to suggest that intelligence demonstrates *the faculty of thought and reason.* Although this is a little better (and more limiting; the thermostat has been left behind), it really just expands our definition problem by introducing two even more unclear terms, *thought* and *reason.* In fact, the feat of providing a *true* definition of intelligence is an old and harried debate that is far beyond the scope of this text. Thankfully, making good games does not require this definition. Actually, this text will agree with our first dictionary definition, as it fits nicely with what we expect game systems to exhibit to be considered intelligent. For our purposes, an intelligent game agent is one that acquires knowledge about the world, and then acts on that knowledge. The quality and effectiveness of his actions then become a question of game balance and design.

TeamLRN

WHAT IS "GAME AI"?

Let us start by giving an academic definition to AI. In their seminal AI Bible, *Artificial Intelligence: A Modern Approach,* Russel and Norvig [Russel95] say that AI is the creation of computer programs that emulate acting and thinking like a human, as well as acting and thinking rationally. This definition encompasses both the cognitive and the behavioral views of intelligence and includes rationality and "humanity" (because being human is sometimes far from rational), but is still considered intelligent (like running into a burning building to save your child).

Game AI is the code in a game that makes the computer-controlled opponents (or cooperative elements) appear to make smart decisions when the game has multiple choices for a given situation, resulting in behaviors that are relevant, effective, and useful. Note the word "appear" in the last sentence. The AI-spawned behaviors in games are very *results* oriented, and thus, we can say that the game world is primarily concerned with the behavioralist wing of AI science. We're really only interested with the responses that the system will generate, and don't really care how the system arrived at it. We care about how the system *acts*, not how it *thinks.* People playing the game don't care if the game is using a huge database of scripted decisions, is making directed searches of a decision tree, or is building an accurate knowledge base of its surroundings and making inferred choices based on logical rules. The proof is really, as they say, all in the pudding as far as game AI goes. So, purely behavioral decisions, such as which attacks an opponent launches, how he navigates the terrain to get to you, how he uses the elements in the environment, and many other details can and are all considered decisions best left to the game's AI system.

Modern game development also uses the term *AI* to describe other game behavior, for instance, the way the interface works with input from the human. Even the algorithms that govern movement and collision (if the game uses animation-driven movement, rather than physics simulations) sometimes fall under this category. As you can see, the term *AI* is a broadly used moniker in the game development world. When discussing AI with someone else in the industry (or even within the company you work at), it's important to know that you both agree on the meaning and scope of the term; otherwise miscommunication can occur if your notion of AI is vastly different (be it simpler or more complex, or just at opposite ends of the responsibility spectrum) than the other person's. When this book refers to AI, it will use the rather narrow definition *character-based behavioral intelligence.* The term *character* refers to the character (or actor, or agent) driven nature of most games. True, many strategy or so-called "God" games have the notion of an "overseer," that is making decisions in the big picture sense, but this could again be considered just another character.

In the old days, AI programming was more commonly referred to as "gameplay programming," because there really wasn't anything intelligent about the behaviors exhibited by the CPU-controlled characters. See Figure 1.1 for an overall game AI timeline. Most coders in the early days of video gaming relied on patterns or some

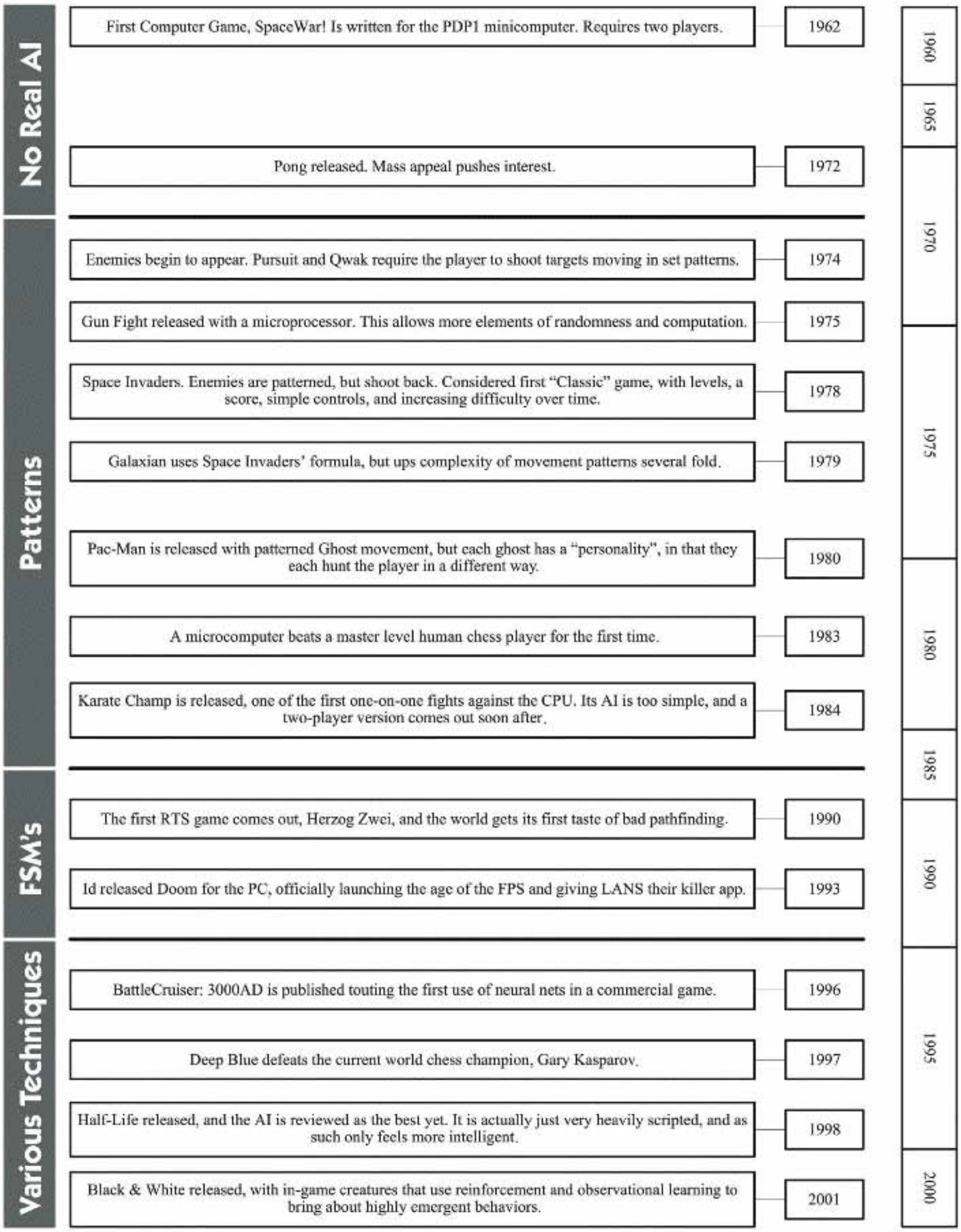


FIGURE 1.1 Game AI timeline.

TeamLRN

repetitive motions for their enemies (e.g., *Galaga* or *Donkey Kong*), or they used enemies that barely moved at all but were vulnerable to attack only in certain "weak points" (like *R-Type*). That was the game, to some extent: finding the predetermined behavior patterns so that you could easily beat that opponent (or wave of opponents) and move on to another. This was done because of the extreme restraints of early processor speed and memory storage. Patterns could be stored easily, requiring minimal code to drive them, and required no calculation; the game simply moved the enemies around in the prescribed patterns, with whatever other behavior they exhibited layered on top (for instance, the *Galaga* enemies shooting when you're beneath them while moving in a pattern). In fact, some games used supposed *random* movement, but because the random number generator in these early games used a hard-coded table of pseudo-random numbers, a pattern could eventually be seen in the overall game behavior.

Another commonly used technique in the past to make games appear smarter was to allow the computer opponents to cheat; that is, to have additional information about the game world that the human player does not have and, thus, make decisions about what to do next that seem remarkably smart. The computer reads that you pushed the punch button (before you've even started the punch animation) and responds with a blocking move. A real-time strategy (RTS) cheater might have its workers heading toward valuable resource sites early in the game, when they hadn't explored the terrain to find those resources. AI cheating is also achieved when the game grants *gifts* to the computer opponent, by providing him additional (and strategically timed) abilities, resources, and so forth that it uses outright, instead of planning ahead and seeing the need for these resources on its own. These tactics lead to more challenging but ultimately less satisfying opponents because a human player can almost always pick up on the notion that the computer is accomplishing things that are impossible for him. One of the easier to notice and most frustrating examples of this impossible behavior is the use of what is called *rubber banding* in racing games. Toward the end of a race, if you're beating the AI-controlled cars by too much, some games simply speed up the other cars until they've caught up with you, and then they return to normal. Sure, it makes the race more of a battle, but watching a previously clueless race car suddenly perform miracles to catch up to you borders on the ridiculous.

In modern games, the old techniques are being increasingly abandoned. The primary selling point of games is slowly but surely evolving into the realm of AI accomplishments and abilities, instead of the graphical look of the game as it was during the most recent phase of game development. This emphasis on visuals is actually somewhat causal in this new expansion of AI importance and quality; the early emphasis on graphics eventually led to specialized graphics processors on almost every platform, and the main CPU is increasingly being left open for more and more sophisticated AI routines. Now that the norm for game graphics is so

high, the wow factor of game graphics is finally wearing thin, and people are increasingly concentrating on other elements of the game itself. So, the fact that we now have more CPU time is very advantageous, considering that the current consumer push is now for games that contain much better AI-controlled enemies. In the 8-bit days of gaming or before, 1–2% of total CPU time was the norm, if not an overestimation, for a game's AI elements to run in. Now, games that require AI are budgeting 10–35% of the CPU time to the AI system [Woodcock01], with some games going even higher.

This means that today's game opponents can find better game solutions without cheating and can use more adaptive and emergent means—if for no other reason that they have access to faster and more powerful processors driving them. As such, AI as defined in modern games is increasingly showing more *real* intelligence (as defined by academic AI), instead of the old standby of prescripted patterns or behaviors mimicking intelligent behavior. Traditional game AI work is being progressively more infused with techniques from the academic world of AI (heuristic search, learning, planning, etc.), which will only continue as games (and gamer's tastes) become more complex.

WHAT GAME AI IS NOT

As shown in the previous section, game AI is a broad label that is often inaccurately used when referring to anything about the game other than graphics: the collision avoidance (or pathfinding) system, the player controls, the user interface, and sometimes the animation system for a game are all pushed into the AI corner. To some extent, these elements do have something to add to the AI world and are elements that, if done poorly, will make the AI seem "stupider," but they are not the primary AI system in a game. An exception to this rule might be a game in which the gameplay is simple enough that the entire smarts of the enemies are in moving around or choosing the right animations to play.

This book will emphasize this differentiation: game AI makes intelligent decisions when there are multiple options or directions for play. These secondary systems, while making decisions from a pool of solutions/animations/paths, are more "Find the *BEST*" (read: singular) solution for this particular input. The main AI, in contrast, might have many equally good solutions, but needs to consider planning, resources, player attributes, and so on to make decisions for the game's bigger picture.

An alternative way of thinking about this differentiation is that these support systems are much more low-level intelligence, whereas this book will focus more on the high-level decisions that an AI system needs to make. For example, you get out of your chair and walk across the room. The thought in your mind was, "I want

TeamLRN

a soda out of the fridge." But look at all the low-level intelligence you used to accomplish the task: your mind determined the right sequence of muscle contractions to get you out of the chair (animation picking), and then started you moving toward the fridge, threading you through all the things on the floor (pathfinding). In addition, you might have lost your balance but regained it quickly (physics) or scratched your head on the way there, in addition to a myriad other minor actions. None of these changed the fact that your entire plan was to go get a soda, which you eventually accomplished. Not all games use a tiered decision system, with an overseeing high-level AI, however. Most just split up the various levels of decision making into separate systems that barely communicate. The point is that these low-level systems do *support* the intelligence of the agent but, for this book's purposes, do not *define* the intelligence of an AI-controlled agent.

Another point to consider is that creating better game AI is not necessarily a result of writing better code. Many programmers believe that AI creation is a technical problem that can be solved purely with programming skill, but there's much more to it than that. When building game AI, a good software designer must consider balancing issues from such disparate areas as gameplay, aesthetics, animation, audio, and behavior of both the AI and the game interface. It is true that a vast number of highly technical challenges must be overcome by the AI system. However, the ultimate goal of the AI is to provide the player with an entertaining experience, not to be a demonstration for your clever code. Gamers will care not about your shiny new algorithm if it doesn't feel smart *and* fun. Game AI is not the best code; it is the best use of code and a large dollop of " whatever works (WW)." Some of the smartest looking games in the past have used very questionable methods to achieve their solutions, and although this book is not advocating poorly written code, nothing should be thrown away if it helps to give the illusion of intelligence and enhances the fun factor of the game. Plus, some of the most elegant game code in the world started out as a mindless hack, which uncovered a clever algorithm on retrospection and cleanup.

The reason the WW mentality has permeated the community is because of a common situation that comes into play very late in a game's development and is usually a sign of poor scheduling. Because AI concerns were so low on the totem pole in years past, the desperate AI programmers of old were somewhat forced to use these questionable methods to cram smarts into their systems. This *cramming* may have lead to better-looking behaviors, but it also led to difficulty debugging, maintaining, and extending game code. Luckily for us, this type of thing is becoming minimal because AI is becoming more and more important to the success of the game and is therefore given the forethought of extended technical design, as well as the time necessary to polish. If you are trying to use a generalized AI algorithm that is clean and elegant, but finding that it is either limiting the types of things your

engine can accomplish, or is causing undue pressure on the production staff by requiring a lot more secondary resources, then the issue of utility needs to be brought up. The world of game AI is just like the rest of the universe—there isn't always an elegant solution for everything.

On a less serious note, game AI is also not some kind of new life form, a disconnected brain that will eventually take over your PlayStation® and command you to feed it regularly. Hollywood tells us that this is what AI has in store for us, but the truth is likely far less dramatic. In the future years of AI research, we will most likely have access to a truly generic AI paradigm that could learn to competently play any game, but for now this is not the case. Nowadays, game AI is still very game specific, and very much in the hands of the coders that work on it. The field is still widely misunderstood by the nonprogramming public, however, and even by those people working in game development who don't regularly work with AI systems. As such, care must be taken not to fall prey to the whims, suggestions, and "game ideas" from management or game designers who don't fully understand the limitations of the AI system in a particular game.

HOW THIS DEFINITION DIFFERS FROM THAT OF ACADEMIC AI

The world of academic AI has two main goals. First is to help us *understand* intelligent entities, which will in turn help us to understand ourselves. This first goal is also the goal of more esoteric fields, such as philosophy and psychology, but in a much more functional way. Rather than the philosophical, "Why are we intelligent?," or the psychological, "Where in the brain does intelligence come from?," AI is more concerned with the question, "How is that guy finding the right answer?" Second is to *build* intelligent entities, for fun and profit, you might say, because it turns out that these intelligent entities can be useful in our everyday lives. The second goal mirrors the nature of the practical economy (especially in the so-called western world), in that the research that is most likely to result in the largest profits is also the most likely to win funding.

Russel and Norvig [Russel95] define AI as the creation of computer programs that emulate four things: thinking humanly, thinking rationally, acting humanly, and acting rationally. In academic study, all four parts of this definition have been the basis for building intelligent programs. The Turing test is a prime example of acting humanly—if you cannot tell the difference between the actions of the program and the actions of a person, that program is intelligent. Cognitive theories that are helping to blend traditional human mind science into AI creation will lead to more humanlike behaviors and, we can hope, to "think" in the same ways that humans do. Sheer logic systems try to solve problems without personal bias or

TeamLRN

emotion, by thinking purely rationally. But it is acting rationally—always trying to come up with the answer—that most defines academic AI methods. Acting rationally is in fact the primary leg of the four-part definition that researchers are striving for in labs across the world.

On this point, the major goal of game AI and the more traditional studies part ways. Academic AI is usually concerned with making rational decisions, that is, the best or most correct (given that there might not be a best) situation. In contrast, game AI focuses on acting "human," with much less dependence on total rationality. This is because game AI needs to model the highs and lows of human task performance, instead of a rigorous search toward the best decision at all times. This is for entertainment reasons, of course.

Say you're making a chess game. If you're making this chess game as part of an academic study, you want it to play the best game possible, given time and memory constraints. You are going to try to achieve perfect rationality, using highly tuned AI techniques to help you navigate the sea of possible actions. However, if you are building your chess game to give a regular human player an entertaining opponent to play against, then your goal shifts dramatically. Now you want a game that provides the person with a suitable challenge, but doesn't overwhelm the human by always making the best move. Yes, the techniques used to achieve these two programs might parallel in some ways, but because the primary goal of each program is different, the coding of the two systems will dramatically diverge. The people who coded *Big Blue* did not care if Kasparov was having fun when playing against it. But the people behind the very popular *Chessmaster* games surely spend a lot of time thinking about the fun factor, especially at the default difficulty setting.

Chess is a somewhat odd example because humans playing a chess program usually expect it to do pretty well (unless they're just learning and have specifically set the difficulty rating of the program to a low level). But imagine the same example used in an AI-controlled *Quake* "bot" deathmatch opponent. If the bot came into the room, dodged perfectly, aimed perfectly, and knew exactly where and when powerups spawned in the map, it wouldn't be very fun to play against. Instead, we want a much more *human* level of performance from a game AI opponent. We want to play against an enemy that occasionally misses, runs out of ammo in the middle of a fight, jumps wrong and falls, and everything else that makes an opponent appear human. We still want competent opponents, but because our measure of competence, as humans, involves a measure of error, we expect shortcomings and quirks when determining how intelligent, as well as how real, something is.

Academic AI systems, on the other hand, generally are not trying to model humanity, although some are in one way or another. They are mostly trying to model intelligence—the ability to produce the most rational decision given all the possible decisions and the rules. This is usually their one and only requirement and, as such, the reason why all our limitations (such as time or memory) are not given

thought. Also, by distancing themselves from the issues of humanity, they don't run into the sticky problems that we as game people have in dealing with questions about what constitutes intelligence and proper problem solving. They just happily chug along, searching vast seas of agreed-upon possibility for the maximum total value.

Eventually, computing power, memory capacity, and software engineering will become so great that these two separate fields of AI research may no longer be dissociated. AI systems may achieve the kind of performance necessary to solve even the most complex of problems in real time, and as such programming them might be more like simply communicating the problem to the system.

APPLICABLE MIND SCIENCE AND PSYCHOLOGY THEORY

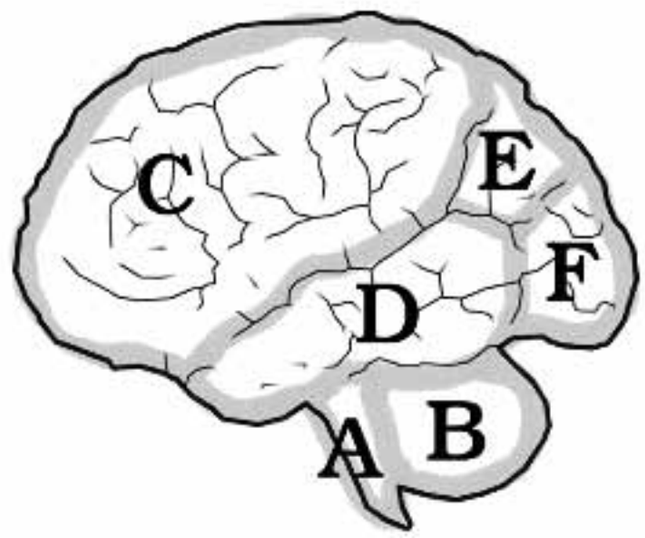
Thinking about the way that the human mind works is a great way to flavor your AI programming with structural and procedural lessons from reality. Try to take this section with a grain of salt, and note that different theories exist on the workings and organization of the mind. This section is more to give you ideas and notions of how to break down intelligence tasks in the same ways that the human mind does it.

Brain Organization

The brain is divided up into subsections, classically (and again, somewhat wrongly) broken into three main groupings: the hindbrain (or brain stem), the midbrain, and the forebrain. Most people have heard these divisions referred to as the reptilian brain, the mammalian brain, and the human brain, but recent research has shown this sort of clear-cut, species-related division to be false. Almost all brains have all three parts, just in different sizes and, in some cases, in dramatically different locations (thus, snakes have a *mammalian* brain region). These brain regions operate independently by using local working memory areas and accessing neighboring synaptic connections to do specific tasks for the organism (fear conditioning is centered in the amygdala, for example). But these regions also are interconnected, some areas heavily so, to perform global-level tasking as well (the amygdala, through the thalamus and some cortical regions, is also a primary first step collection spot for emotional data that will then be sent to the hippocampus for blending with other sensory input and eventual storage). This makes the brain in some way object oriented, with rampant accessor functions and a lack of a real "parent class," however.

The organizational model of the brain has merit when setting up an AI engine, as seen in Figure 1.2, which shows the relative tasking between brain and game systems. By breaking down your AI tasks into specific atomic modules that require

TeamLRN



Organization of:		
	The Brain	A Game AI System
A	Brain Stem - Reflex - Lower Functions/Survival	- Collision - Animation Selection
B	Cerebellum - Motor Center/Sensory Mixing and Coordination	- Physics - Navigation
C	Frontal Lobe - Higher Brain Functions - Emotions - Learning	- Decision Making
D	Temporal Lobe - Memory (Visual and Verbal)	- Learning
E	Parietal Lobe - Sensory Cortex	- Perceptions
F	Occipital Lobe - Visual Processing	- Perception

FIGURE 1.2 Object-oriented nature of the brain related to game AI systems.

little knowledge of the others, you allow other classes to collect the output of these smaller modules together and blend this knowledge into more complex representations that the game character can then use. This also represents the kind of efficiency we should be trying to achieve in our AI systems. Avoid single use calculations and code, or input conditions that are so rare as to be practically hard coded. Alas, inefficiency cannot be completely overcome, but most inefficiencies can be eliminated with clever thinking and programming.

Knowledge Base and Learning

Although the inner workings of the human memory system are not fully understood, or even agreed upon, the common idea is that information is stored in the

form of small changes in brain nerve cells at the synapse connection level (please excuse the heavily simplified neural description). These changes cause differences in the electrical conductivity of different routes through the network and, as such, affect the firing potential of specific nerve cells as well as whole subnetworks. If you use a particular neural pathway, it gets stronger. The reverse is also true. Thus, memory systems use a technique that games could learn a lot from (no pun intended), that of *plasticity*. Instead of creating a set-in-stone list of AI behaviors and reactions to human actions, we can keep the behavior mix exhibited by the AI malleable through plasticity. The AI could potentially keep track of its actions coupled with whether or not the human consistently chooses certain behaviors in response. It could then recognize trends and bias its behaviors (or the requisite counter measures, as a defense) to plastically change the overall behavior mix that the AI uses.

Of course, an AI system would require a dependable system for determining what is "good" to learn, whereas the human brain just stores everything, which can lead to misconception, miscommunication, and even delusion. Although very contextually complex, a filter on AI learning would keep the human player from exploiting a learning system by teaching it misleading behaviors, knowing that the system will respond in kind. Does the AI always use a low block to stop the next incoming punch after the player has punched three times in a row? Then the player will perceive that and punch three times followed by a high punch to get a free hit in on the low-blocking AI.

Another useful lesson from nature is that the rate of memory reinforcement and degradation in the human brain is not the same for all systems. Memories associated with pain aversion, for instance, may never fully extinguish, even if the person only experienced the relation once and never again. This is a good example of nature using *dynamic hard coding*. The plastic changes already mentioned could be "locked in" (by stopping the learning process or moving these changes into a more long-term memory) and thus not be allowed to degrade over time. But like the brain, too much hardcoding, or if used in the wrong place, can lead to odd behavior, turning people (or your game characters) into apparent phobics or amnesiacs.

A further concept to think about is long-term versus short-term memory. Short-term, or working memory, can be thought of as perceptions that can only be held onto for a short time, which are then filtered for importance, and then stored away into longer-term memories, or simply forgotten about. This creates such concepts as attention span, as well as single mindedness. Many games have somewhat digital memory, where the enemy will see you (or even worse, get shot in the arm by you), come after you for some set time period, but if you hide, will completely forget about you and go back to what he was doing. This is classic state-based AI behavior, but it is also very unrealistic and unintelligent behavior. By coding up more analog memory models for our opponent, or just giving him the ability to remember anything longer than a minute or so, he could still go back to his

TeamLRN

post, but would be much more sensitive to future attacks, and would probably make it a priority to call for backup, and so forth. For sure, some games do use these types of rules. But there is always room for improvement and expansion of realistic AI behaviors.

The brain also makes use of *modulators,* chemicals that are released into the blood that take a while to degrade. These are things like adrenaline or oxytocin. These chemicals main job is to inhibit or enhance the firing of neurons in specific brain areas. This leads to a more focused mind-set, as well as flavoring the memories of the particular situation in a contextual way. In a game AI system, a modulator could override the overall AI state, or just the behavior exhibited within a certain state. So, conventional state-based AI could be made more flexible by borrowing the concept of modulation. The earlier-mentioned enemy character that you alarmed could transition to an entirely different Alerted state, which would slowly degrade and then transition back down to a Normal state. But using a state system with modifiers, he could stay in his normal Guard state, with an "aggressive" modulator. Although keeping the state diagram of a character simpler, this would require a much more general approach to coding the Guard state.

The human brain learns (which can be thought of as extending your knowledge base of the world through bias and association) by storing things in the different memory centers of the brain. It usually does this in a few, separate ways: exploration or direct experience, imitation, or imaginative speculation. With the possible exception of speculation, which requires quite a sophisticated mental model, game characters may gather information in the same ways. But when you decide that your game is going to use learning techniques, carefully decide how you want the game to come up with its learned data. Keeping statistics on the behaviors that seem to work against the human is one way, and so is recording the things that the human player is doing against the AI opponent and either imitating or trying to deflect those human behaviors.

The problem that games have had with classical AI learning algorithms (as well as the physical model of the brain) is that they usually take many iterations of exposure to induce learning. It is a slippery slope to do learning in the fast-paced, short-lived world of the AI opponent. Most games that use these techniques do all the learning before hand, during production, and then ship the games with the learning disabled, so that the behavior is stable. This will change as additional techniques, infused with both speed and accuracy, are found and made public.

But learning need not be conscious. Influence maps, for instance, can be used by a variety of games to create unconscious learning that will make AI enemies seem much smarter and can be learned with as few as one or two applications. A simple measure of how many units from each side have died on any one spot on the map could give an RTS game's pathfinding algorithm all the information it needs to avoid these *kill zones* where an opponent, human or otherwise, has set up a

deadly trap along some commonly traveled map location. This learning could erode over time or be influenced by attacking units relaying back that they destroyed whatever was making an area a kill zone in the first place. Influence maps are also being used successfully in some sports games. For example, by slightly perturbing the default positions of the players on a soccer field to be better positioned for the passes the human has made in the past, as well as using this same system for the defensive team to allow them to be better able to possibly block these passes.

This type of system allows cumulative kinds of information to be readily stored in a quick and accessible way, while keeping the number of iterations that have to occur to see the fruition of this type of learning very low. Because the nature of the information stored is so specific, the problem of storing misleading information is also somewhat minimized.

Cognition

A flood of data coming from our senses bombards us at all times. How does the brain know which bits of information to deal with first? Which pieces to throw away? When to override the processing it is currently doing for a more life-threatening situation? It does this by using the brain's various systems to quickly categorize and prioritize incoming data. Cognition can be thought of as taking all your incoming sense data, called *perceptions,* and filtering them through your innate knowledge (both instinctual and intuitive) as well as your reasoning centers (which includes your stored memories), to come up with some understanding of what those perceptions mean to you. Think of logic, reason, culture, and all of your personal stored rules as merely ways of sorting out the perceptions you need to be aware of from the background noise. Think of the sheer volume of input coursing into the mind of a person living in a big city. He must contend with the sights, sounds, and smells of millions of people and cars, the constant pathfinding through the crowd, the hawkers, and homeless vying for his attention, and countless other distractions. If his brain tried to keep all this in mind, it would never be able to concentrate sufficiently to perform any task at all.

These perceptions are not all external. The pressures of the modern world cause stress and anxiety that split your attention and fragment your thoughts. Your mind also needs to try to distill the important thoughts inside your own head from the sea of transient, flighty ideas that everyone is constantly engaged in.

In game AI, we don't suffer as much from the flood of data because we can pick and choose our perceptions at any level in the process, and this does make the whole procedure a bit less mystical. In Figure 1.3, you can see a mock-up of a sports game using different perceptions for the various decisions being made by the AI player in the foreground. Make sure, when coding any particular AI subsystem that you only use those perceptions you truly need. Be careful not to oversimplify, or

TeamLRN

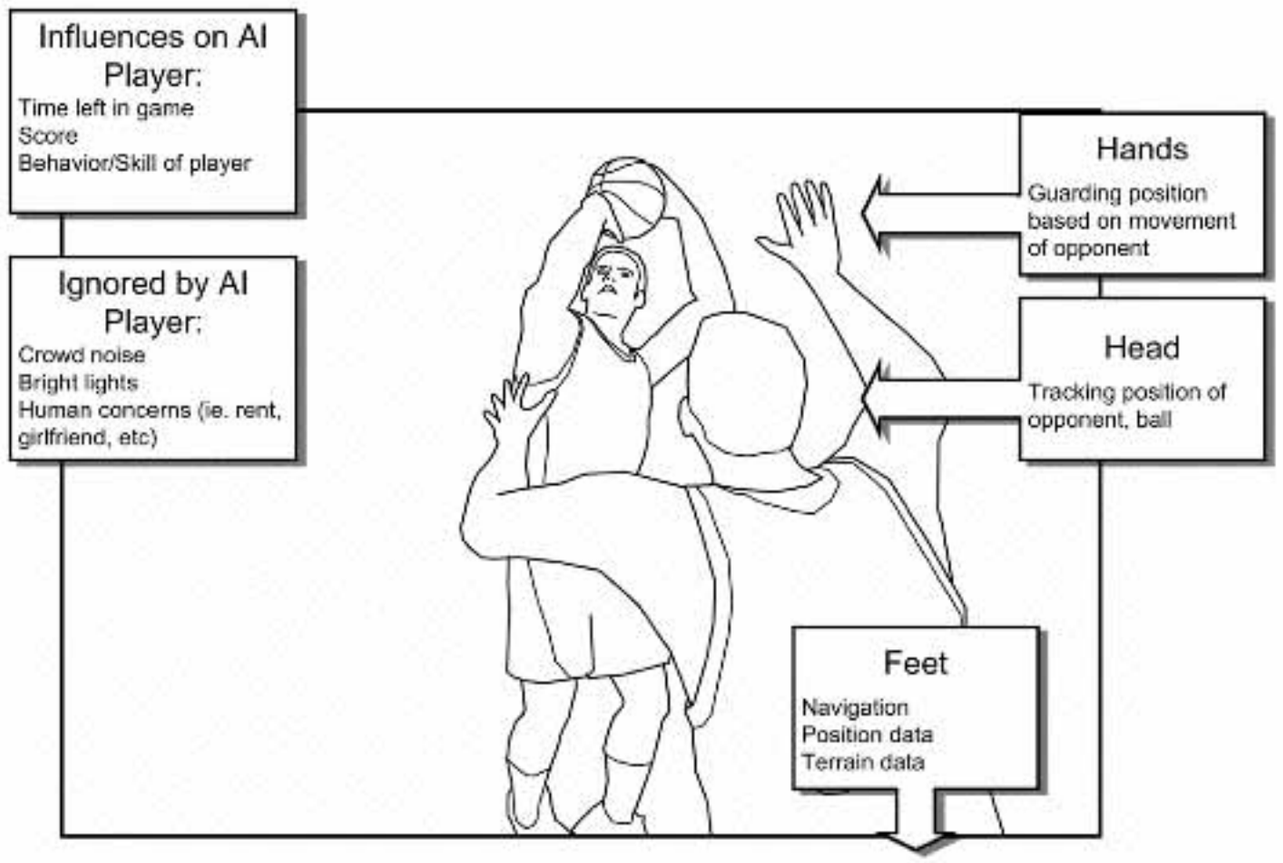


FIGURE 1.3 A visual depiction of various perceptions being taken into account by a game character.

you may make the output behaviors from this subsystem too predictable. An auditory subsystem that only causes an enemy character to hear a sound when its location is within some range to the enemy would seem strange when you set off a particularly loud noise just outside of that range. You should take into account distance and starting volume, so that sounds would naturally trail off as they traveled. You might also want to take into account the acoustics of the environment because sounds will travel much longer distances in a canyon than in an office building, or under water than in open air. These are very simple examples, but you see the notion involved. Perceptions are much more than a single value, because there are usually many ways to interpret the data that each perception represents.

We can think of the systems used in the AI world as filters as well. Whatever technique we are using as our primary decision-making system, to determine the right action to perform, is really just a method of filtering the current game state (by means of the perception variables as registered by the AI) through all the possible things that the AI can do (or some subset of these possibilities, as defined by some rule or game state). Thus, we see the primary observation many people make about AI in general—that it widely comprises focused searching, in some way or another. This is true to some degree. Most AI systems are just different ways of searching through the variety of possibilities, and as such, the topography of your game's possibilities can be used to conceptually consider the best AI technique to use. This topography is generally called the "state space" of the game. If your game's possible outcomes to different perceptions is mostly isolated islands of response, with no

real gray conditions, a state-based system might be the way to go because you're dealing with many more digital possible responses, an almost enumerated state space. However, if the full range of possible responses is more continuous, and would graph out more like a rolling hillside with occasional dips (or another metaphor with more than three dimensions, but you get the idea), a neural net-based system would probably be better because they tend to work better at identifying local minima and maxima in continuous fields of response. We will cover these and the other AI systems in Part III and Part IV of the book; this was merely for illustration.

Theory of Mind

One psychological construct that is again being embraced as a major field of investigation by both behavioralists and cognitive scientists is that of the so-called *Theory of Mind* (ToM). This concept has a good deal of merit in the field of game AI because our primary job is creating systems that *seem* intelligent. A ToM is actually more of a cognitive trait, rather than a theory. It fundamentally means that one person has the ability to understand others as having minds and a worldview that are separate from his own. In a slightly more technical fashion, ToM is defined as knowing that others are *intentional agents,* and to interpret their minds through theoretical concepts of intentional states such as *beliefs* and *desires* [Premack78]. This isn't as complicated as it sounds, however. You could think of this as having the ability to see intent, rather than just strict recognition of action. We do it all the time as adults, and humanize even the most nonhuman of environmental elements. Listing 1.1 shows a bit of code from a Java version (written by Robert C. Goerlich, 1997) of the very early AI program *Eliza,* which, in its time, did a remarkable job of making people believe it was a lot more than it really was.

What does this give us? In human terms, the ability to form a ToM about others usually develops at about the age of three. A commonly used test to determine if the child has developed this cognitive trait is to question the child about the classic "False Belief Task" [Wimmer83]. In this problem, the child is presented with a scene in which a character named Bobby puts a personal belonging, such as a book, into his closet. He then leaves, and while he's away, his little brother comes and takes out the book and puts it in a cupboard. The child is then asked where Bobby will look for his book when he comes back. If the child indicates the cupboard, he reveals that he has yet to develop the understanding that Bobby wouldn't have the same information in his mind that the child does. He therefore does not have an abstract frame of reference, or theory, about Bobby's mind, hence no ToM about Bobby. If the child gives the correct answer, it shows that he can not only determine facts about the world but can also form a theoretical, simplified model of others' minds that includes the facts, desires, and beliefs that they might have; thus providing a theory of this other's mind.

TeamLRN

LISTING 1.1 Some Sample Code from a Java Version of Eliza

```
public class Eliza extends Applet

      {

    ElizaChat       cq[];
    ElizaRespLdr    ChatLdr;
    static ElizaConjugate  ChatConj;
    boolean         _started=false;
    Font            _font;
    String          _s;

public void init()
     {

    super.init();
    ChatLdr = new ElizaRespLdr();
    ChatConj = new ElizaConjugate();

    //{{INIT_CONTROLS
    setLayout(null);
    addNotify();
    resize(425,313);
    setBackground(new Color(16776960));
    list1 = new java.awt.List(0,false);
    list1.addItem("Hi! I'm Eliza.  Let's talk.");
    add(list1);
    list1.reshape(12,12,395,193);
    list1.setFont(new Font("TimesRoman", Font.BOLD, 14));
    list1.setBackground(new Color(16777215));
    button1 = new java.awt.Button
          ("Depress the Button or depress <Enter> to send to
Eliza");
    button1.reshape(48,264,324,26);
    button1.setFont(new Font("Helvetica", Font.PLAIN, 12));
    button1.setForeground(new Color(0));
    add(button1);
    textField1 = new java.awt.TextField();
    textField1.reshape(36,228,348,24);
    textField1.setFont(new Font("TimesRoman", Font.BOLD, 14));
    textField1.setBackground(new Color(16777215));
    add(textField1);
    //}}
```

```
    textField1.requestFocus();
}

public boolean action(Event event, Object arg)
     {
    if (event.id == Event.ACTION_EVENT && event.target ==
                                                        button1)
         {
            clickedButton1();
            textField1.requestFocus();
            return true;
    }
    if (event.id == Event.ACTION_EVENT && event.target ==
                                                        textField1)
         {
            clickedButton1();
            textField1.requestFocus();
            return true;
    }
    return super.handleEvent(event);
}

public void clickedButton1()
     {
    parseWords(textField1.getText());
    textField1.setText("");
    textField1.setEditable(true);
    textField1.requestFocus();

}
public void parseWords(String s_)
     {
    int idx=0, idxSpace=0;
    int _length=0;      // actual no of elements in set
    int _maxLength=200;  // capacity of set
    int _w;

    list1.addItem(s_);
    list1.makeVisible(list1.getVisibleIndex()+1);
    s_=s_.toLowerCase()+" ";
    while(s_.indexOf("'")>=0)
        s_=s_.substring(0,s_.indexOf("'"))+
                s_.substring(s_.indexOf("'")+1,s_.length());
```

TeamLRN

```
                bigloop: for(_length=0; _length<_maxLength  &&
                                idx < s_.length(); _length++)
                     {
                    // find end of the first token
                    idxSpace=s_.indexOf(" ",idx);
                    if(idxSpace == -1) idxSpace=s_.length();

                   String _resp=null;
                   for(int i=0;i<ElizaChat.num_chats && _resp == null;i++)
                        {
                        _resp=ChatLdr.cq[i].converse

(s_.substring(idx,s_.length()));
                    if(_resp != null)
                         {
                         list1.addItem(_resp);
                         list1.makeVisible(list1.getVisibleIndex()+1);
                         break bigloop;
                    }
                }
                // eat blanks
                while(s_.length() > ++idxSpace &&
                           Character.isSpace(s_.charAt(idxSpace)));
                idx=idxSpace;

                if(idx >= s_.length())
                     {
                     _resp=ChatLdr.cq[ElizaChat.num_chats-1]
                                        .converse("nokeyfound");
                     list1.addItem(_resp);
                     list1.makeVisible(list1.getVisibleIndex()+1);
                }
            }
        }
        //{{DECLARE_CONTROLS
        java.awt.List list1;
        java.awt.Button button1;
        java.awt.TextField textField1;
        //}}
}
//---------------------
class ElizaChat
     {
```

```
static int          num_chats=0;
private String      _keyWordList[];
private String      _responseList[];
private int         _idx=0;
private int         _rIdx=0;
private boolean     _started=false;
private boolean     _kw=true;
public String       _response;
private String      _dbKeyWord;
public int          _widx = 0;
public int          _w = 0;
public int          _x;
private char         _space;
private char         _plus;

public ElizaChat()
     {
    num_chats++;
    _keyWordList= new String[20];
    _responseList=new String[20];
    _rIdx=0;
    _idx=0;
    _keyWordList[_idx]=" ";

    _space=" ".charAt(0);
    _plus="+".charAt(0);
}

public String converse(String kw_)
     {
    _response = null;
    for(int i=0; i <= _idx - 1;i++){
        _dbKeyWord = _keyWordList[i];

        if(kw_.length()>=_dbKeyWord.length()&&
                _keyWordList[i].equals
                    (kw_.substring(0,_dbKeyWord.length())))
             {

            _widx = (int) Math.round(Math.random()*_rIdx-.5);
            _response = _responseList[_widx];
            _x=_response.indexOf("*");
            if(_x>0)
                 {
```

TeamLRN

```
                        _response=_response.substring(0,_x)+

    kw_.substring(_dbKeyWord.length(),
                                                    kw_.length());
                        if(_x<_responseList[_widx].length()-1)
                            _response=_response+"?";
                        _response=Eliza.ChatConj
                                        .conjugate(_response,_x);
                        _response=_response.replace(_plus,_space);
                    }
                    break;
                }
            }

            return _response;
        }

        public void loadresponse(String rw_)
             {
            _responseList[_rIdx]=rw_;
            _rIdx++;
        }

        public void loadkeyword(String kw_)
             {
            _keyWordList[_idx]=kw_;
            _idx++;
        }
    }
```

It has been routine in philosophy and the mind sciences in general to see this ability as somewhat dependent upon our linguistic abilities. After all, language provides us a representational medium for meaning and intentionality; thanks to language, we are able to describe other people's and our own actions in an intentional way. This is also probably why Alan Turing gave forth his famous test as to a true measure of intelligence exhibited by a computer program. If the program could communicate successfully to another entity (that being a human), and the human could not tell it was a computer, then it must be intelligent. Turing's argument is thus that anything we can successfully develop a ToM toward must be intelligent—great news for our games, if we can get them to trigger this response within the people who play them.

Interestingly, further studies in chimpanzees and even some lower primates have shown their remarkable abilities toward determining intention and prediction

toward each other and us without verbal communication at our level. So, the ability to form ideas about another's mindset is either biologically innate, can be determined with visual cues, or possibly something else entirely. Whatever it may be, the notion is that we do not require our AI-controlled agents to require full verbal communication skills to instill the player with a ToM about our AI.

This is precisely the kind of inherent nature we want to instill in our games. If we can get the people playing our games to see not a creature in front of them with X amount of health and Y amount of strength, but rather a being with beliefs, desires, and intent, then we will have really won a major battle. This superb leap in suspension of disbelief by the human player can only be achieved if the AI system in question is making the kinds of decisions that a human would make, in such a way as to portray these higher traits and rise above the simple gameplay mechanic involved. In effect, we must model minds, not behavior. Behavior should come out of the minds that we give our AI creations, not from the programmers' minds

Note that this does not mean we need to give our creations perfect problem-solving abilities to achieve this state. People are instinctively seeking to create a ToM about other entities they are dealing with to try to anticipate other entities' actions and thoughts. As game AI programmers, we can use this trait to our advantage when creating entities that we want humans to attribute certain qualities to. In effect, knowledge of this fundamental, low-level goal (that of creatures trying to create a ToM about each other) can help give the programmers and designers guidelines about what types of information to show the player directly, what types to specifically not show, and what types to leave ambiguous. As the illusionists say, "The audience sees what I want it to see."

Take for example, an AI-controlled behavior from a squad combat game. In Figure 1.4, we see the layout of a simple battlefield, with the human player at the bottom of the map, and four CPU enemies closing in on him, moving between many cover points. The simple combat rules for these enemies are the following:

- If nobody is shooting at the player, and I'm fully loaded and ready, I will start shooting. Note that only one player can shoot at a time in this system.
- If I'm out in the open, I will head for the nearest unoccupied cover position, and randomly shout something like "Cover me!" or "On your left!" or even just grunt.
- If I'm at a cover position, I'll reload, and then wait for the guy shooting to be finished, maybe by playing some kind of scanning animation to make it look like he's trying to snipe the player.

Now imagine how this battle will look to the human player. Four enemy soldiers come into view. One starts firing immediately, while the other three dive for cover. Then, the one that was firing stops, shouts "Cover me!," and runs forward for cover

TeamLRN

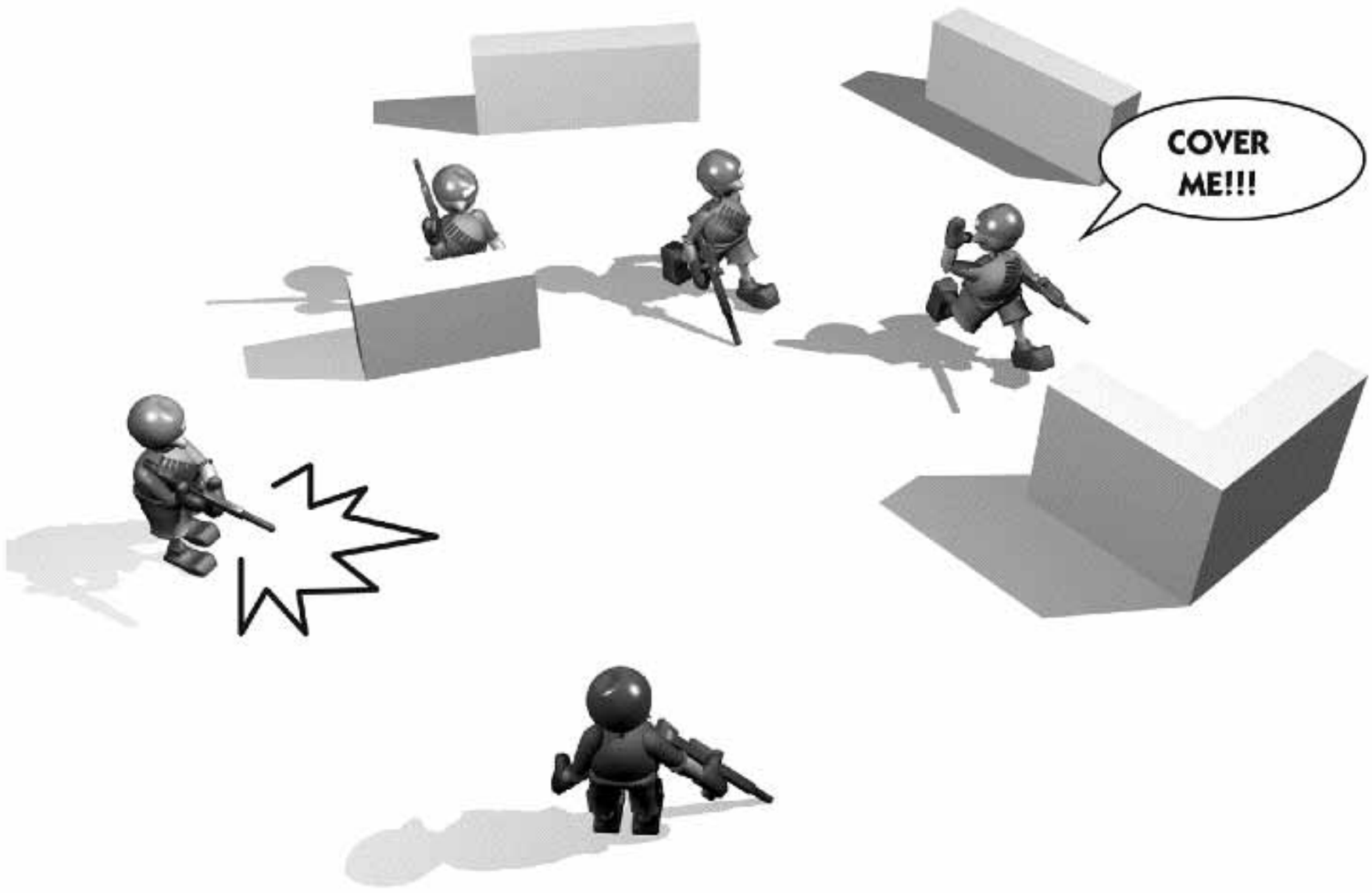


FIGURE 1.4 Emergent theory of mind in a loosely coordinated enemy squad.

as a different soldier pops up and starts firing. Here we have a system in which the soldiers are completely unaware of each other, the player's intentions, or the fact that they're performing a basic leapfrogging advance and cover military maneuver. But because the human player is naturally trying to form a ToM about the enemy, he is going to see this as very tightly coordinated, intelligent behavior. Therefore, the ruse has worked. We have created an intelligent system, at least for the entertainment world.

Bounded Optimality

In trying to achieve a level of rationality in our AI systems, the degree of rationality we are striving for must be declared and allowed to constrain the design of the system. If your goal is near perfect rationality, you had better be able to accept that your program is going to need vastly more time in which to run, unless the decision state space you are working with is very small indeed. For most entertainment games, perfect rationality is unwanted and unnecessary. As discussed earlier, the goal of game AI is to emulate human performance level, not perfect rationality. One of the reasons that humans make mistakes is the idea of bounded optimality (BO).

BO really just means that the system will make the best decision it can, given computation (as well as other resource) restrictions. The possibility of total rationality of the solution is directly linked to the number and amount of limitations. In other words, you get what you pay for. Given a limited view of the world, as well as time constraints, we can construct a decision-making paradigm that will make the best choice within its boundaries.

Like computers, the decision-making ability of people is limited by a number of factors, including the quality and depth of relevant knowledge, cognitive speed, and problem-solving ability. But that only covers the hardware and software. We also suffer from environmental limitations that might make it impossible to fully exploit our brains. We live in a "real-time" world, and must make decisions that could save our lives or make our careers in very short time frames. All these factors come together to flavor our decisions with a healthy dose of incorrectness. So, instead of trying to force our programs to find the ideal solution, we should be merely guiding our decision making in the right direction and working in that direction for as much time as we have. The decisions that come out will then, we hope, be somewhat more human (until, of course, computing power gets to the level that will dilute any time slice restriction to the point of zero) and work well with the limiting constraints of the platform and genre of game we are working on. In effect, we create optimal programs rather than achieve optimal actions.

The concept of BO is also starting to become somewhat prevalent in academic AI circles (as well as in game theory and even philosophy) because so-called optimal solutions to real-life problems are usually computationally intractable. Another reason is that very few real-life problems have no limitations. Given the realities of our world, we need a method of measuring success without requiring absolute rationality.

A problem with trying to use BO methods on many types of systems is that they require incremental solutions; that is, solutions get better by degrees as they are given more resources. Incremental solutions are definitely not universal to all problems, but the types of computationally challenging hurdles that require BO thinking can often be reduced in some way to an incremental level. Pathfinding, for example, can be given several levels of complexity. You might start by pathfinding between very large map sectors, then within those sectors, then locally, and then around dynamic objects. Each successive level is incrementally better than the last, but each level gets the player going in the right direction, at least at some gross level.

Lessons from Robotics

Robotics is one of the few fields that share a vast amount of similar tasking with the world of game AI. Unlike academic endeavors, which deal with large-scale problems that can use exhaustive searches to find optimal results, robots have to deal

TeamLRN

with real-time constraints like physics, computation speed, and perception of the environment. In effect, robots have to deal with the computational problems of solving problems intelligently and must master the skills necessary to house this technology into a physical construct that must deal with the real world at some level. This is truly an ambitious task and, as such, tends to take the academic theories and grind them against the stone of reality until finely honed. Because of this, many techniques coming from robotics end up in games because of the inherent optimizing and real-world use that robotics adds to the theoretical AI work done in research labs. The lion's share of the successful pathfinding methods we use in games, including the invaluable A* algorithm, came out of robotics research. Some of the prime lessons that robotics has given us include the following:

Simplicity of design and solution. Many robotics methodologies use the WW model, and games have fully embraced that paradigm. This is because robotics in general is a very hard problem, with an ambitious variety of challenges such as navigating undefined terrains, or recognizing general environmental objects. Every true sense that a researcher bestows on his robot translates to a tremendous amount of technology and study necessary to break down the system into workable parts. If the system can be made to work without the sense, then the solution is just as good, if not better, considering that the expense in both time and money was saved by not having to involve a complex perceptual system. Some of Rodney Brooks's robot designs illustrate this perfectly: instead of trying to navigate areas by recognizing obstacles, and either circumventing them, or calculating how to surmount them, some of his robot designs are largely mindless, insectile creations that blindly use general purpose methods to force their way over obstacles. The lesson here is that while others spend years trying tech-heavy methods for cleverly getting around obstacles and failing, Brooks's robot designs are being incorporated into robots that are headed to Mars.

Theory of Mind. ToM has also been advanced by robotics. Researchers have discovered that people deal better with robots if they can in some way attribute human attributes (if not human thought processes) to the robot. This natural human process, that of humanization, is a good thing for robotics researchers, in that it actually makes the robot seems more intelligent to people, and more agreeable in the eyes of the public. Imagine a robot built to simply move toward any bright light. Humans, when asked to describe this simple behavior, will usually report that the robot "likes lights," or even "is afraid of the dark." By continuing to give a robot the ability to show desires and intentions, instead of raw behaviors, researchers hope to make robots that people will not only toler-

ate but also enjoy working with in the real world. Robotic projects like Cog and Kismet [Brooks98] continue to push the realm of human-robot interaction, mostly through social cues that further people's ToM about the robot and enliven the interaction itself and the learning that the robot is engaging in.

Multiple layered decision architectures. Many modern robotics platforms use a system (sometimes called subsumption) whereupon the decision-making structure of the robot is broken down into layers, the array of which represents high-level to low-level decisions about the world [Brooks91]. This so-called bottom-up behavior design allows robots to achieve a level of autonomy in an environment because they will always have some fail-safe behavior to fall back on. So, a robot might have a very low-level layer whose only goal is to avoid obstacles or other nearby dangers. This layer would get fresh information from the world quite frequently. It would also override or modify behaviors coming from the top of the decision structure because it represents the highest priority of decision making. As you climb the layers, the priority lessens, the amount of interaction with the world lessens, and the overall goal complexity goes up. So, at the highest level, the robot formulates the high-level plan: "I need to leave the room." The layers within this system know nothing about each other (or as little as possible), they simply build on one another in such a way that the various tasks normally associated with the goal at large are specialized and concentrated into distinct layers. This layer independence also creates a much higher robustness to the system since it means that a layer getting confused (or receiving bad data) will not corrupt the entirety of the structure, and thus, the robot may still be able to perform while the rest of the system returns to normalcy.

A structure of this kind is very applicable to game genres that have to make decisions at many levels of complexity concurrently, like RTS games. By sticking to the formal conventions expressed (as well as experimentally tested) by robotics teams using subsumption techniques, we can also gain from the considerable benefits these systems have been found to exhibit, including automatic fault tolerance (between layers of the system), as well as the robustness to deal with any number of unknown or partially known pieces of information at each level. Subsumption architectures do not require an explicit, start-to-finish action plan, and a well-designed system will automatically perform the various parts of its intelligent plan in an order that represents the best way the environment will allow. This book will cover a general way of breaking down AI engine issues using a method something like this approach in Chapter 23.

TeamLRN

SUMMARY

This chapter covered some basic AI terminology that we will use in later chapters, some general psychological theory, and some concepts from some other fields that are applicable to AI system design.

- This book will use the term game AI to mean character-based behavioral decision making, further refined by concentrating on tasks that require choosing among multiple good decisions, rather than finding the best possible decision.
- Older games used patterns or let the computer opponent cheat by giving it clandestine knowledge that the human player didn't have; both methods are being increasingly abandoned because of the increasing power of AI systems being used in games.
- AI is becoming increasingly important in today's games, as players demand better opponents to more complex games. This is true even though many games are going online because most people still play single-player modes exclusively.
- Game AI needs to be smart and fun because this is primarily a form of entertainment. Thus, game AI needs to exhibit human error and personality, be able to employ different difficulty levels, and make the human feel challenged, but not overly so.
- Brain organization shows us the use of object-oriented systems that build on each other, in complexity order.
- Like the brain, our AI systems can employ long- and short-term memories, which will lead us toward much more real AI behaviors.
- Learning in a game, like in real brains, can be conscious or unconscious. By using both types, we can model more realistic behavior modification over time, while still focusing our learning to things we want to pay attention to.
- Cognition studies lead us to think of AI reasoning systems as "filters" that take our inputs and lead us toward sensible outputs. Thinking of the nature of the state space that a given game has, and contrasting that with the types of AI techniques available, the right "filter" can be found for your game.
- By making it our goal to give someone playing our game a Theory of Mind about the AI-controlled agents, we extend the attributes of the agent to basic needs and desires, and therefore extend the realism of his decision making to the player.
- Bounded rationality is a formal concept that we can use to visualize our game AI goals, in that we are not searching for optimal actions but, rather, optimal programs that give good solutions while working under many constraints.
- Robotics gives us the notions of design and implementation simplicity, extends our desire for giving our creations a ToM, and provides us with a generic subsumption architecture for designing and implementing autonomous agents from the bottom up.

2 An AI Engine: The Basic Components and Design

In this chapter, the basic parts of a common AI engine will be broken down and discussed. Although this list is neither all-inclusive or the only way to do things, almost all AI engines will use these foundation systems in some form or another: decision making and inference, perception, and navigation. See Figure 2.1 for a basic layout.

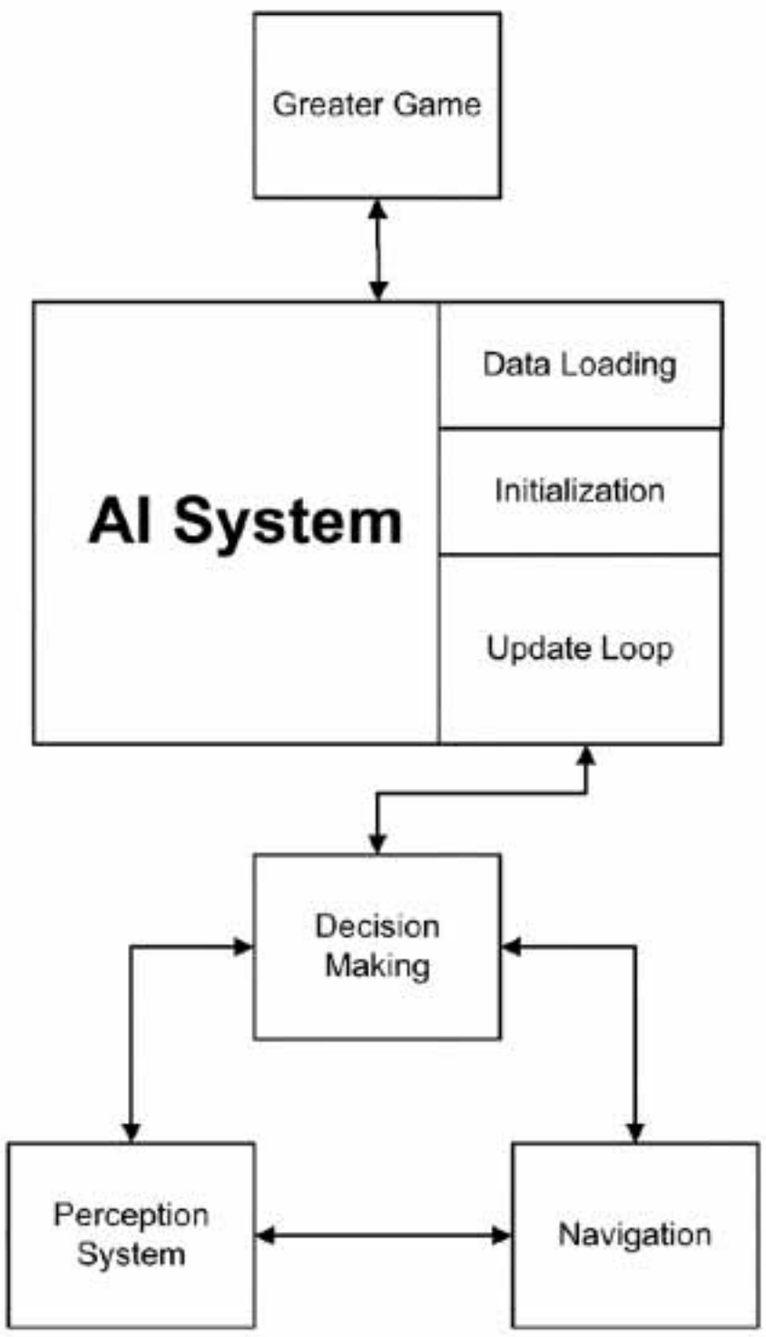


FIGURE 2.1 Basic AI engine layout.

TeamLRN

DECISION MAKING AND INFERENCE

The workhorse of the engine, the decision-making system is the main emphasis of this book. The decision-making system is the underlying structure that determines the type of AI engine (or engine subsection) you are building. Inference is defined as the act of deriving logical or reasonable conclusions from factual knowledge or premises assumed to be true. This means, in game terms, that the AI-controlled opponent gains information about the world (see "Perception Type," later in this chapter) and makes intelligent, reasonable decisions about what to do in response. Thus, your AI system is limited by the kind of information it can gain about the outside world, as well as the richness of the response set as defined by the game design. The more things the game allows the AI characters to do, the greater the response set, or state space, of the game. The technique you choose should be dictated, at least in part, by the size and nature of the state space of the game you are building. More information about this notion will be given in Parts III and IV, where the different techniques are described.

All the types of decision-making systems described in this book can be boiled down to the definition *using available inputs to come up with solutions.* The differences between these techniques will determine which one (or combination) you choose to use. The main differences we are concerned with are the types of solutions, agent reactivity, system realism, genre, content, platform, development limitations, and entertainment limitations.

Types of Solutions

The primary game solution types are strategic and tactical. Strategic solutions are usually long-term, higher-level goals that might involve having many actions to accomplish. Tactical solutions are more often short-term, lower-level goals that usually involve a physical act or skill. An example of the difference would be the two solutions `HuntPlayer`, and `CircleStrafe`, in a *Quake*-style game. Hunting the player is a high-level goal that involves finding the player, getting to the player, and engaging the player in combat. Circle strafing is merely a way to move while engaged in combat with an enemy. Many games require both strategic and tactical solutions, so developers use this division to split the problem into separate parts and combine different techniques for getting these solutions.

Agent Reactivity

How reactive do your game elements need to be? Scripted systems tend to create characters with much more stylized and contextual response, but they also tend to become locked into these behavior scripts and, thus, lose reactivity. Conversely, fully reactive systems (those that take the inputs, and change responses immedi-

ately, with little thought to what was being done before) tend to be considered either spastic or cheating, and are not very human feeling. Very responsive systems also require a fairly rich response set, or the behavior they exhibit will be very predictable and stale. However, this is great for arcade style, or what are called "twitch" games. This point needs to be addressed based on the type of game being created and the proper balance determined based on the gameplay experience you are looking to create.

System Realism

To be considered "realistic," the decisions and actions that an AI element comes up with need to be regarded as human. Each AI entity requires the intelligence to determine the right thing to do, given the limitations of the game. But being human also means making mistakes. Thus, AI characters need to show human weakness as well. Opponents that block all your punches or never miss a shot or a *Scrabble* opponent that knows the entire dictionary would not be enjoyable and would frustrate the player. The goal is to strike a balance between competition and entertainment, so that the player is drawn in by the challenge of the game, but also given a constant stream of positive feedback by beating the game. Other realism concerns involve the amount of actual adherence to physical laws the game uses. Can the player jump higher than in real life? Can he fly? Do players heal quickly? All these things are up to the developer. What this means is that "realism" can be defined as "real in this particular game world." Care must be taken in fantasy worlds because enemies that arbitrarily break rules will be considered cheating, not magical. You must take steps to ensure that the player knows the rules of your world and stick to them. Remember too, that Earth physical laws are usually known by most of the people playing your game, whereas special laws might provide your players with an initial stumbling block as they try to get used to the your new rules.

Genre

The different types of games require specific types of AI systems. See Part II of the book for an in-depth discussion of the genres. At the genre level, keep in mind the following:

- *Input (or perception) types.* Things to note include number of inputs, frequency, communication method (polled, event, callback functions, etc.), and any hierarchical relationships among inputs. Arcade-style games might have very limited inputs, whereas a character in a real-time strategy game might require quite a few perceptions about the world—to navigate terrain, stay in formation, help friendly units, take orders from the human, and respond to attacking enemies.

TeamLRN

- *Output (or decision) types.* Based on the values given to the decision-making part of the engine by the perception system, a decision is made, and an output is generated by the AI system. Outputs can be analog, digital, a series of events on top of ambient behavior, and involve the entire character (such as diving for cover), merely parts of the character (such as turning your head in response to a noise), or multiple characters (such as having your townspeople mine more stone). Outputs can be specific (affecting a single character in a certain way, like jumping into the air), or be high level ("we need to create Dragon units"), which could affect the behavior of many AI characters and change the course of many decisions.
- *The overall structure of the decisions needed for the genre.* Some games have fairly simple or single natured decisions. *Robotron* is a good example. The monsters head towards you, with a set speed and movement type, and try to kill you. But a complex game, like *Age of Empires,* requires many different types of decisions to be made during the game. You have team-level strategy, unit tactics, an array of pathfinding problems, and even more esoteric things, such as diplomacy. Each of these might represent a subsystem in the AI that is using an entirely different technique to get its job done.

Content

Over and above the game's genre are special-case gameplay concerns brought about by special or novel game content. Games like *Black & White* required very specialized AI systems for the basic gameplay mechanism, that of teaching your main animal behaviors by leading or showing it how to do things. This requires careful deliberation when designing the framework up front, but also is aided by early prototype work to flesh out design holes.

Platform

Will the game be made for the personal computer, a home console, an arcade architecture, or for a handheld platform? Although the lines between these differing machines are beginning to blur, each still has its own specific requirements and limitations that must be taken into account. Some AI considerations on each platform:

- *PC.* Online PC games might require user extensibility (in the form of included level or AI editors), so your AI system would need to handle this. Single-player PC games usually have fairly deep AI systems because PC game players are usually a bit older and want a tad more complexity and opponent realism. The standard input mechanism on the PC is the mouse (except for flight simulators [sims] or racing games), so remember that your human players will be per-

forming the specific commands of your game with this device, and if you make the AI perform things that would be either tedious or impossible with the mouse, they'll cry foul. Also, the constantly changing PC means that the minimum configuration for most games is going to keep climbing, so AI programmers need to predict the minimum configuration that the game will use (usually one to three years after the game is started) when making design decisions. PC game experiences are also usually longer (more than 30 hours of gameplay), and thus, the opponent AI needs to vary more often, so that playing against it doesn't get repetitious.

- *Consoles.* The realism constraints are lifted because console gamers are usually younger and more open to fantasy situations. However, now there is a much higher usage of difficulty settings because the overall range of player's skill is much greater. Memory and CPU budgets are usually much stricter because these machines (at least until now) have been very limited compared with their PC brothers. Console games have a much higher standard of quality, for the most part—from a quality assurance standpoint, rather than a quality of gameplay experience. Games on consoles usually don't crash, although this "PC only" problem has begun to creep into the console world. Because of this higher standard, however, your AI system has to endure much longer and more strenuous testing before it is approved for release. Many companies test their games internally, and then the maker of the console also tests the game before it gets to the shelves. Therefore, any "exotic" AI styles (such as learning systems) that are used in the game might make this testing process longer because of the inherent nonreproducibility of some of these advanced AI techniques.
- *Arcade.* The arcade platform was huge in the 1970s and 1980s when it was cost prohibitive to have advanced graphics hardware in everybody's home and home consoles were much simpler (like the Atari® 2600™ and the ColecoVision®) in what they could display. Because of today's increasingly powerful home machines, the arcade industry has had to make large changes. Nowadays, most arcade machines are one of three types: large, custom cabinets (such as sit-down racing games or skiing simulators), custom inputs (light gun games, music games), or small games that can be put in the corner of a bar or some other nondedicated arcade environment. *Golden Tee* golf is a good example of the last type. With the custom arcade machines, the sky is usually the limit in hardware. The entire package is customized, so the developer is free to put as much RAM and processing power as needed (within the limits of reason, of course). The smaller games actually tend to be sold as "kits," where the owner of the game can swap out parts from an old game with that of a newer game. Arcade AI is usually still "pattern based" because people assume that's what they're in for when they put in a quarter (or a dollar or more in some of the modern games). Tuning AI for the arcade environment usually involves putting

TeamLRN

a beta machine in a local venue, and getting statistics back from the machine to determine if areas of the game are too easy, too difficult, or whatever might be detrimental to the amount of money coming into the machine. So, AI for the arcade world is usually simple, but the tuning is difficult because you are trying to balance fun factor with cash flow.

- *Handheld.* The most restrictive platform, the handheld world has been almost exclusively ruled by the Nintendo® Gameboy®, but has recently become the hot area of game development, with PDAs, cell phones, and just about every other gadget you can think of now being turned into gaming devices. These machines usually have very little RAM, the number of input buttons is severely limited (this is true even on cell phones, which are not true game consoles and, thus, not designed to recognize more than one button being pressed at a time), and the graphical power of these mini machines is very small. In fact, people who used to work heavily in the 8- and 16-bit worlds are finding their talents are marketable again. AI on these platforms needs to be clever and space and speed optimized. As such, these machines usually use throwback techniques for their AI systems: patterned movement, enemies as mindless obstacles, or cheating (by using knowledge about the human that they only have because they're part of the program). However, this will change as more powerful handheld systems are developed, and the handheld/console line will blur.

Development Limitations

Development limitations includes budgetary concerns, manpower issues, and schedule length. Basically, the AI programmer needs to equate all these things into his one resource: time. The AI programmer really needs to have a good sense of time. How much time do you have to invest in the design phase, the production phase, and finally the test or tune phase? This last phase of the process is potentially the most important, as has been proven repeatedly when the best games are inevitably the most highly polished. True, designing the system is paramount as well because a well-designed engine will provide the programmer with the ability to add the necessary behavioral content to the game quickly and easily, but even the best designed games need extensive tuning to get proper feel.

Because the role of AI in a game is inherently higher level (rather than low-level engine code, such as the math library, or the renderer) and because new ideas and behaviors seem to almost inevitably come up late in the production, AI systems are notorious for "feature creep." This is defined as new features being added toward the end of the project, such that the final date keeps creeping out into the future. This indicates one of two things: a bad game that requires additional elements to be fun or playable or a good game that can be made just that much better. If you find yourself in the latter situation, good for you. If management is willing to take the

additional investment of time and money to really maximize the product above its initial design, that's great. But tacking on additional elements as quickly as possible to make a questionable or failing game better is a recipe for disaster. A good, up-front game design really is your best line of defense against this, but the production staff also needs to curtail this malady by keeping careful and strict accordance to the schedule.

As you will note in Part II, almost all games use some form of state-based AI, if not as the primary system. This is mostly because of the nature of games in general. People like at least some level of predictability in games—if you're constantly engaged in a never-ending, ever-changing fight, you'll burn out quickly. The AI (or gameplay experience in general) in most games needs to be somewhat cyclical, with phases of action, followed by a phase of rest, and then repeat. This pacing lends itself well to a state-based approach. However, most games use combination engines, with multiple decision-making sections devoted to the differing AI problems found during the span of the game, so don't feel that a state-based model is the only way to go.

State-based methods are so prevalent because they are a means of organizationally dividing the state space of the entire game into manageable chunks. Instead of trying to tackle the logical connections between decisions in the game as a whole, you in effect split the game into smaller subgames that can be solved more easily. That's why even games that don't lend well to the state architecture as a whole can still benefit from the partitioning of a high-level state machine that can divvy up the solution state space into convenient pieces by defining states that are really only internal states that provide this partitioning. For example, *Joust* is a very dynamic game, every level is pretty much the same (with the exception of the egg stages), and the AI system is more rule based than state based. But you could divide a normal level of *Joust* into three states: a Spawning state (where the enemies are instantiated), a Regular state (during normal gameplay), and an Extended state (where time has run out, and the Pterodactyl is after the human player). Optionally, you could divide the Regular state even further to refine the behavior more. So, you could determine that the AI character is on the bottom layer of the screen, or the middle, or the top, and actually make that a state, so the AI system could respond differently to this *location* state. This piece of information could obviously be used as a simple modifier in the Regular state (the regular state would have a switch statement dividing up the behavior determination based on the placement of the character, for example). By instead using it as another series of states, each resultant state is simpler on its own accord, rather than a more complex Regular state. Whether or not this upsets the balance between organizational simplicity and having repetitious code would have to be determined through implementation.

Another reason for the preponderance of state machines in game AI is for testing, tuning, and debugging purposes. The quality assurance staff (QA, or "testers") is going to have a heck of a time determining if the game AI is faulty, or too hard,

TeamLRN

or crashes the computer, if the game's AI system isn't reproducible in some way. Tuning a game based on nonstate techniques is much harder, and specific suggestions are very hard to implement (and we all know that producers are chock full of specific suggestions, sometimes dangerously close to product completion). These types of concerns will be discussed in more detail on a technique-by-technique basis in Parts III and IV of the book.

Entertainment Limitations

Games are also famous for some assumed entertainment elements, and many game players respond negatively to their absence, consciously or not. This includes things like the rock-paper-scissors (RPS) scenario. A commonly used notion in game design is that "everything that can be done should have a countermove," thus leading to the RPS comparisons. If your game opponents have abilities that cannot be countered by the human player, you'd better have a good reason, or your game isn't going to be much fun. But if the human can do something that the AI cannot counter, your game is going to be too easy, and you again lose out. This is the classic game *balancing* that is so crucial to the final success of a game.

The question of difficulty levels is another entertainment question that must be answered by your AI system. Static skill levels (which are set before the game begins, usually by the player) are typically considered better than dynamic skill levels (levels that change in real time as the player progresses). This is because most players want to know the challenge level they are trying to beat, although you could set up a "static" difficulty level that the player would know is going to adjust as the game progresses. It is also because people's skill levels vary so much, both from person to person, and at the specific task level, that dynamic skill level adjustments are very hard to do and still have the game feel like it's balanced and not cheating. Some people enjoy being very anxious about the game, loving the feeling of being just on the edge of their seats, but others just want to sit back and sail through like a tourist, noting the sights and such. Another problem with dynamic skill levels is that you have to somehow filter out exploratory or nonstandard behavior that the human does from behavior associated with being "stuck" or frustrated because of the difficulty.

Because we are making video games, and not movies, there is also a problem with getting across emotion or intent of the AI characters to the player, without being heavy handed or trite. In movies and TV, this can be done with dramatic camera angles, lots of dialogue, and the inherent expressivity of the human face. In a game, it's much harder to use camera angles because (especially in 3D games) the control scheme might be tied to the camera, or you might need a wide angle camera in order to play the game (for example, you might need to see most of the field in a football game, and gameplay would be hurt by even a fairly short close up of somebody's face). Therefore, we are left with a somewhat limited set of tools to get

this type of information across. We can caricature the emotion, which is useful for more cartoonish games, like *Crash Bandicoot* or *Ratchet and Clank*. The use of classic cartoon stretch and squash when animating moves in these games helps to really bring emotion into the characters from afar, without having to use a close up camera. Dialogue can help, but can get repetitive and also requires some level of lip-synching to look good. A character with a sad look on his face, but a generic flapping lower jaw while talking, isn't going to convey the level of emotion you're going for. We need to realize that most actions have to be fairly obvious to be perceived. Better graphical power in today's platforms are leading to this problem being a bit easier to deal with because we can actually model more complex characters and use more subtle animations to enliven them, but home consoles are still suffering from the limited resolution of regular TV, which means that small details are mostly blended into nothingness on non-HDTVs.

INPUT HANDLERS AND PERCEPTION

AI perception can be defined as the things in the environment that you want the elements in your game to respond to. So, this might be as simple as the player's position (in *Robotron*, this was the only input to the AI of note, besides the enemy's own position), to as complex as a careful record of the units that the computer has seen the human use in a real-time strategy (RTS) game. Usually, these types of data registers are encapsulated into a single code module, if possible. Doing this makes it easier to add to the system, ensures that you are not repeating calculations in different parts of the AI system, helps in tuning, and distills the computations into an easily optimized central location.

A central perception system can also tag additional data or considerations on each input register, including perception types, update regularity, reaction time, thresholds, load balancing, and compilation cost and preconditions.

Perception Type

The various types of inputs might include Boolean, integer, floating point, and so on. They might also include static perceptions (a perception needed for logic in a basketball game might be "Ball Handling Skill is greater than 75," which really only needs to be determined once, unless your game allows for that skill to be adjusting during a game.

Update Regularity

Different perceptions might only need to be updated only so often because they don't change that often or are expensive to recalculate all the time. This could

TeamLRN

somewhat be considered a form of reaction time, but isn't really. It's more like a polled perception that you don't mind being slightly out of date. Continuing our basketball example, this could be used with line of sight check that determines if the ball holder has a clear lane to the basket. That's a pretty expensive check, especially if you use prediction on all the moving characters to determine if they will move out of the corridor in time to allow for passage. So, you might only want to check this every so many ticks, instead of all the time.

Reaction Time

Reaction time is the pause before an enemy acknowledges a change in the environment. With a reaction time of zero, the computer seems just like that, a computer. But, by giving a slightly random (or based on some skill attribute) amount of pause time before things are acknowledged by the enemy, the overall behavior of the system seems much more human and fair. This can also be tweaked for difficulty level, to make the overall game more or less difficult as desired. Reaction time can also give a modicum of personality to characters, so faster characters will respond quicker than slower ones will.

Thresholds

Thresholds are the minimum and maximum values that the AI will respond to. This can be for simple data bounds checking but could also simulate a slightly deaf character (his minimum auditory threshold might be higher than that of other characters), or an eagle eye enemy (who sees any movement at all, instead of large or fast movement). Thresholds can also go down or up in response to game events, again to simulate perception degradation or augmentation. So, a flash grenade would temporarily all but blind an opponent, but a patrol guard started by an unidentified sound might actually become *more* acute because he's paying so much more attention for a short while.

Load Balancing

In some games, the amount of data that the AI needs to take into account might be too numerous or too calculation-heavy to evaluate on any one game tick. Setting up your perception system so that you can specify the amount of time between updates of specific input variables is an easy way to load balance the system so that you don't end up needing far too much CPU time for something that rarely changes.

Computation Cost and Preconditions

In addition to load balancing the calculations as just described, you should also consider raw computation cost. Thus, design your system with any hierarchically

linked computations in mind, so that simple precondition calculations are done first, such that any more complex determinations might not have to be done at all if they can be somewhat based on the more simple computations. To give an oversimplified example, let us say that in the game of *Pac-Man,* an AI routine for running the main character around needs to make (among others) two calculations: the number of power pills, and the distance to each power pill location. *Pac-Man* would be better off checking the total number of power pills first (by checking some sort of Power Pill Count variable, or polling the various pills to see how many are still active), to make sure there is one, before he recalculates his distance to all the power pills because this is a more costly calculation.

The perception system you choose for your game will most likely be game specific because the inputs to which your AI system will respond depend heavily on the type of game, the emphasis of the gameplay, any special powers that the characters or enemies have, and many other things. Some data your AI systems will require are simulated human sensory systems (such as line of sight or hearing radius), whereas others will just use the information straight from the game. Make sure you don't go too far with this latter group, or you run the risk of cheating. More likely, you will use extended information to craft an algorithm for this kind of input because it would be too costly computationally or space wise (such as a detailed map of everywhere the AI has been, or modeling a sense that someone is behind you).

The main two paradigms for updating the perception registers are the following:

Polling. Polling involves checking for specific values to change, or making calculations, on a game loop by game loop basis—for example, checking to see if a basketball player is open for a pass every tick. This is necessary for much of the data that your AI will respond to, but is also the kind of data that is much more likely to need load balancing (see earlier). Use this method for analog (continuous or real values) inputs or inputs that may be present in some form all the time.

Events. Using events is in some ways the opposite of polling; the input itself tells the perception system that it has changed, and the perception system notes that change. If no events are shunted to the perception system, it does nothing. This is the preferred method for digital inputs (on/off, or enumerative states) that don't change often (rather than 30 times a second or more like the human player's position, for example). The reason for this is that if you're going to have a constant stream of events being registered, queued, and then acted upon, you're really just adding overhead to a polling system (for that particular input).

TeamLRN

Some games—stealth games in particular—make extensive use of advanced perception systems. This is because the senses of your enemies become a weapon against you, and a large part of the game experience is about beating the perception system, in addition to the objectives of the game. See Chapter 5, "Adventure Games," for more on this.

NAVIGATION

AI navigation is the art of getting from point A to point B. In our search for more realistic/thrilling/dramatic games, the worlds of modern games commonly involve large, complex environments with a variety of terrains, obstacles, movable objects, and the like. The reason we have well-researched AI algorithms for solving problems like this is because of the field of robotics, which has had to deal with trying to get robots to maneuver through tougher and tougher environments. Navigation involves two main tasks: *pathfinding* and *obstacle avoidance.*

Pathfinding is an interesting, complex, and rather frustrating problem. In the old days, pathfinding was almost nonexistent, as environments were simple or wide open (like that in *Defender,* where the enemies simply headed in your exact direction), or the enemies really didn't head in your direction but, rather, random directions that you had to avoid (like the barrels in *Donkey Kong*). But, when games started having real worlds to move around in, all this changed. To have an AI character move from point A in the world to point B, you're going to need a system to help him find the way. Several different schemes have come about to do this, including grid based, simple avoidance and potential fields, map node networks, navigation mesh, and combination systems.

Grid Based

In a grid-based system, the world is divided up into an even grid, either square or hexagonal, and the search algorithm A* (the heavyweight champ of pathfinding) or some derivative is used to find the shortest path using the grid. Each grid square has a "traversal possibility" value, usually from 0 (cannot pass through at all) to 1 (totally open for travel). Simple systems might use just binary values for the grid, where more complex setups would use the full analog values to show the height of the grid (to make it possible to simulate going uphill being harder than going downhill) or special attributes of the grid squares, such as "water" or "someone is standing here." See Figure 2.2. Concerns with grid-based solutions are sheer memory size of the grid, as well as storage of the temporary data as the system finds the shortest path. This can also be cost prohibitive if the grid is fairly high resolution because the amount of work the search algorithm has to do goes up dramatically with finer grids.

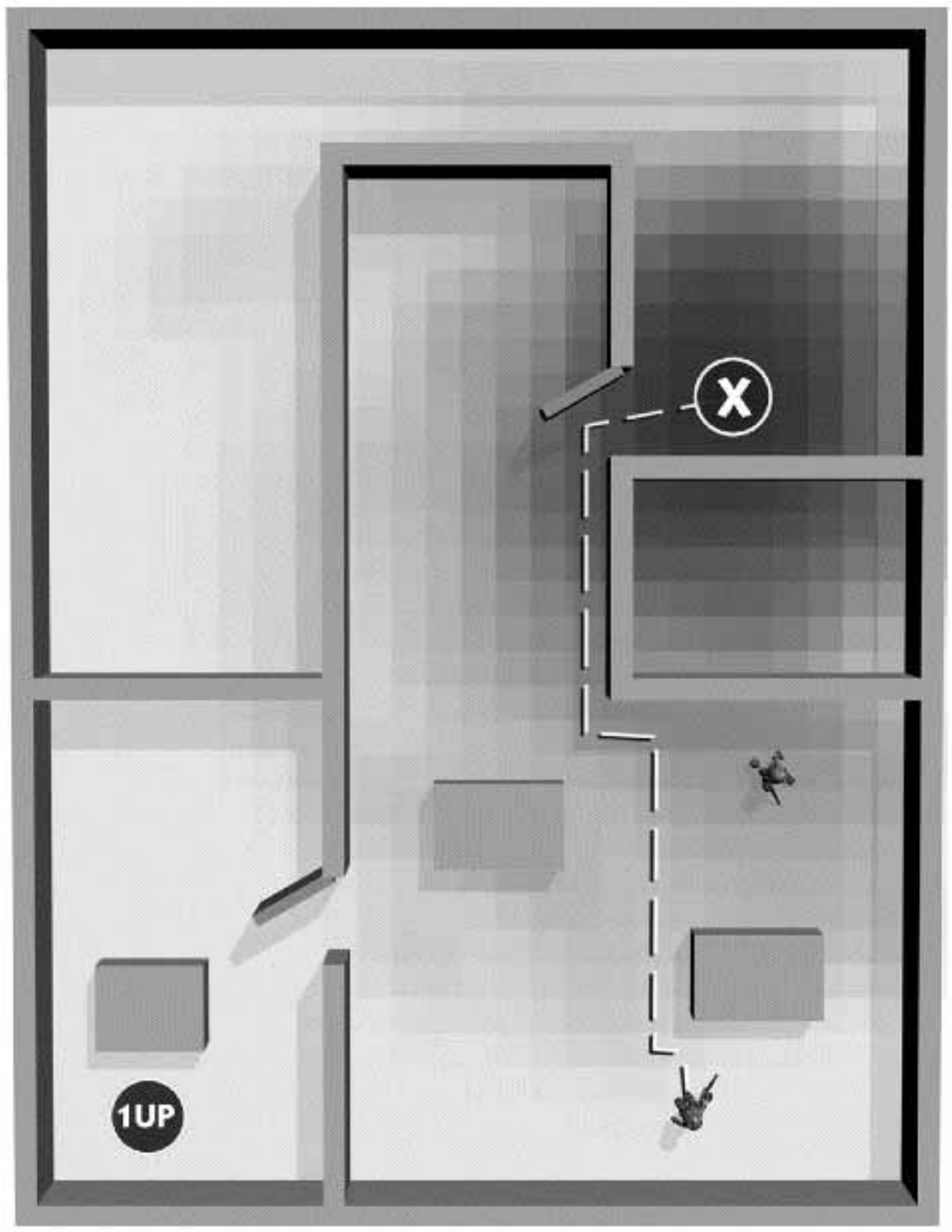


FIGURE 2.2 Example of grid squares.

Simple Avoidance and Potential Fields

In the simple avoidance and potential fields method, you again separate the map into a grid and associate values with each grid area that push or pull the AI character from areas of high potential to areas of low potential value. In an open world with convex obstacles, this technique can be preprocessed, leading to an almost optimal Voronoi diagram of the space (i.e., a mathematically sound optimal "partition" of the space—plenty of research has been done with Voronoi methods) providing good quality pathfinding (and fast because the paths are extracted from existing data by simply following the line of decreasing potential as opposed to heavy searching). With convex obstacles, however, you cannot preprocess because

TeamLRN

you don't know the direction of travel. See Figure 2.3. In this case, the pressure is now on the potential field generator.

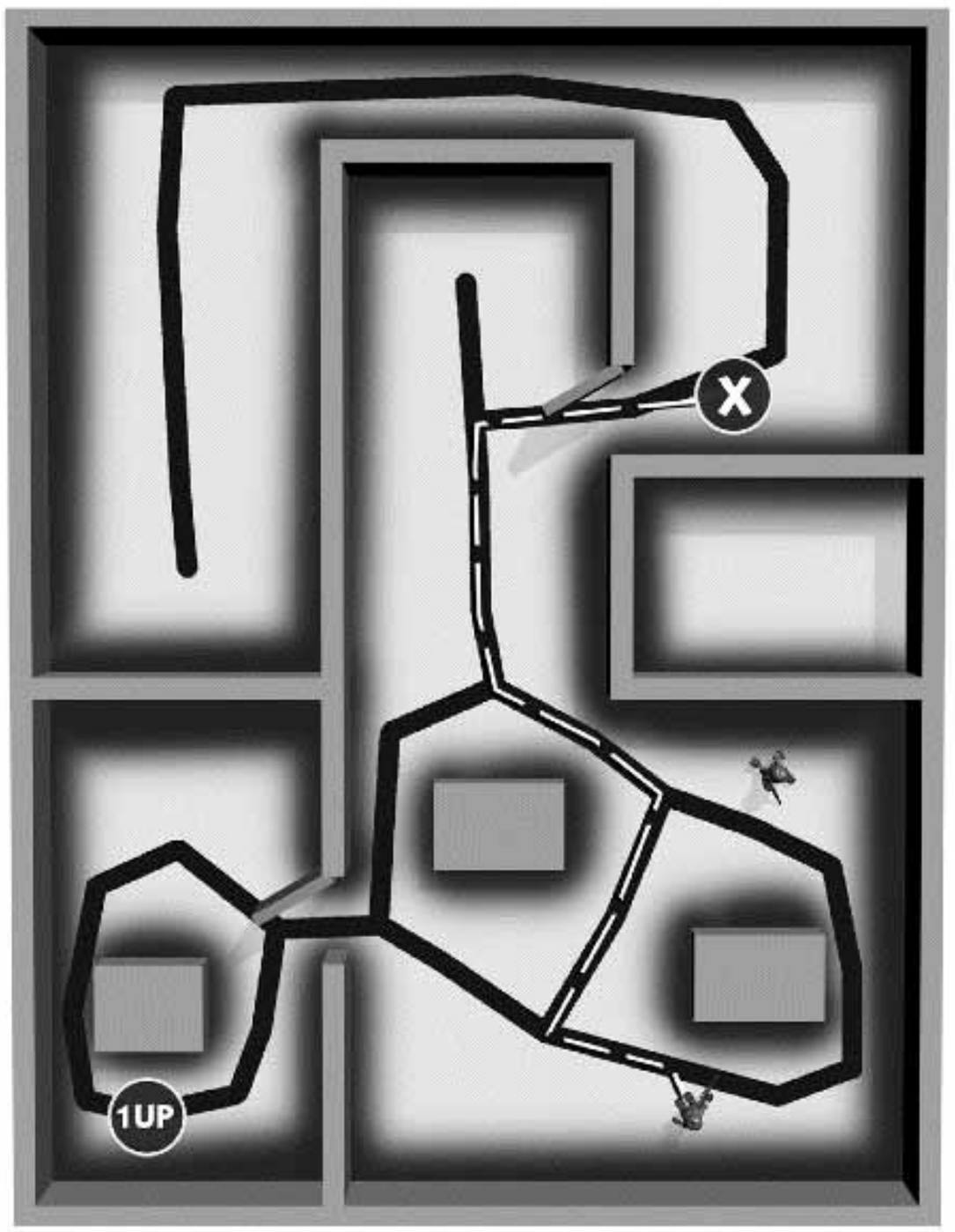


FIGURE 2.3 Preprocessed potential fields.

Map Node Networks

Map node networks are for more expansive worlds, or more three-dimensional (3D) worlds, than grid systems can comfortably portray. With this method, the level designers, when making the levels, actually lay down a series of connected waypoints that represent interconnectedness among the rooms and halls that make up the level. See Figure 2.4. Then, just like the grid-based method, a search algorithm (most like A*) will be used to find the shortest connected path between the points. In effect, you are using the same technique as described earlier, but are reducing the state space in which the algorithm can operate tremendously. The memory cost is much less for this system, but there is a cost. The node network has to be created correctly, to make for good-looking paths, and maintained if the level is changed. Also, this

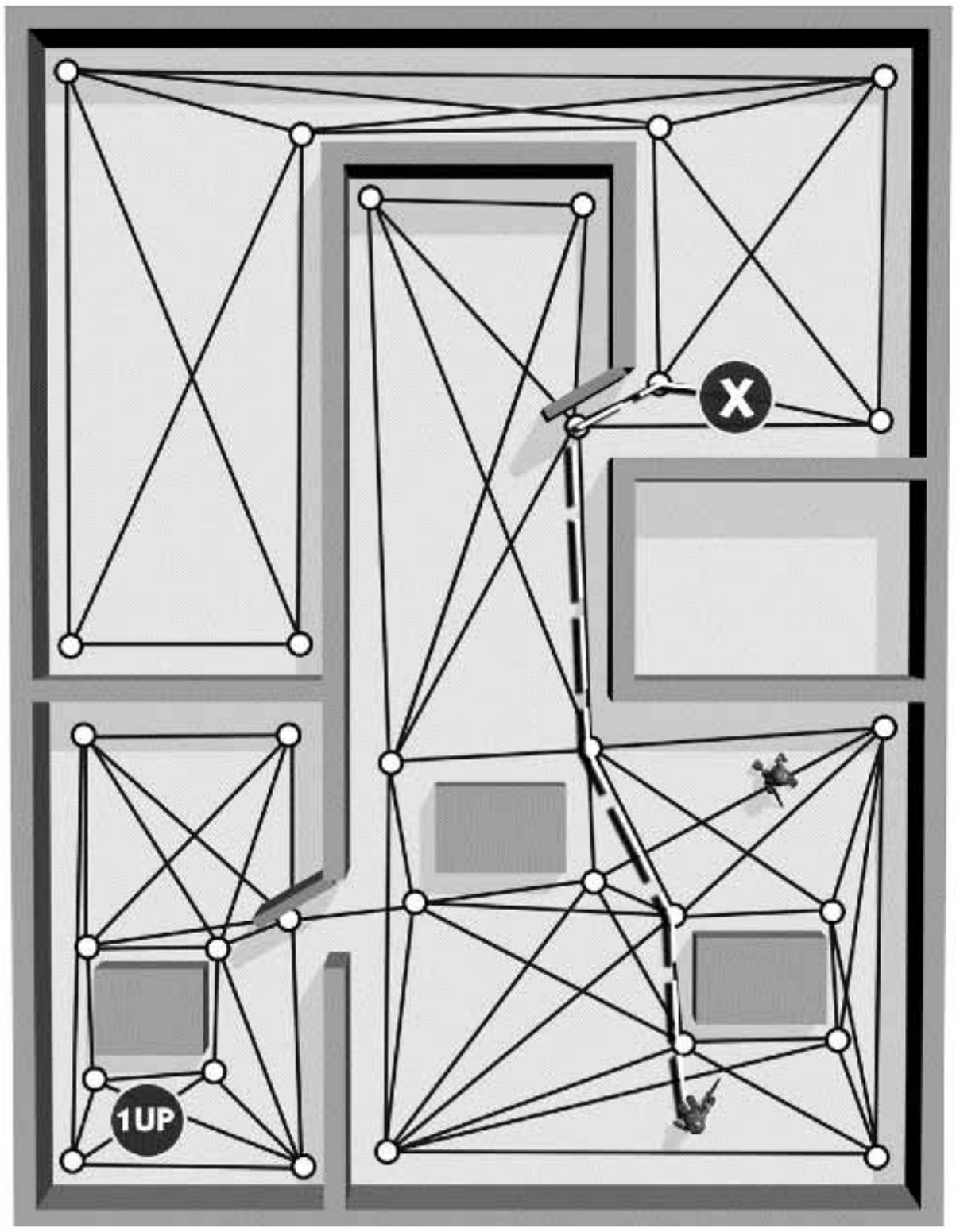


FIGURE 2.4 Map node network systems.

method doesn't lend itself well to dynamic obstacles, unless you don't mind inserting the dynamic object locations into the node network. A better way is to use some form of obstacle avoidance system to take care of moving objects, and use the node network to traverse the environment. The obstacle avoidance system would kick in when you get too close to something and would just perturb the direction of travel to the next path node as long as the local obstacle was in the way.

Navigation Mesh

A navigation mesh system tries to get all the advantages of the map node system, without having to create or maintain the node network. By using the actual polygons used to build the map, this system algorithmically builds a path node network

TeamLRN

that the AI can use. See Figure 2.5. This is a much more powerful system, but can lead to some strange looking paths if the method of constructing the navigation mesh isn't fairly intelligent itself, or the levels were not built with the knowledge that this process was going to be performed. A system of this sort is best used for simple navigation because gameplay specific path features (such as teleporters or elevators) would be difficult to extract with a general algorithm, unless the level designers lay down specific connection data that would be associated with these special case gameplay elements and used in building the network. If you're trying to spare the level designers the worry of dealing with navigation issues, this step would somewhat defeat the purpose of autogenerating a navigation mesh in the first place.

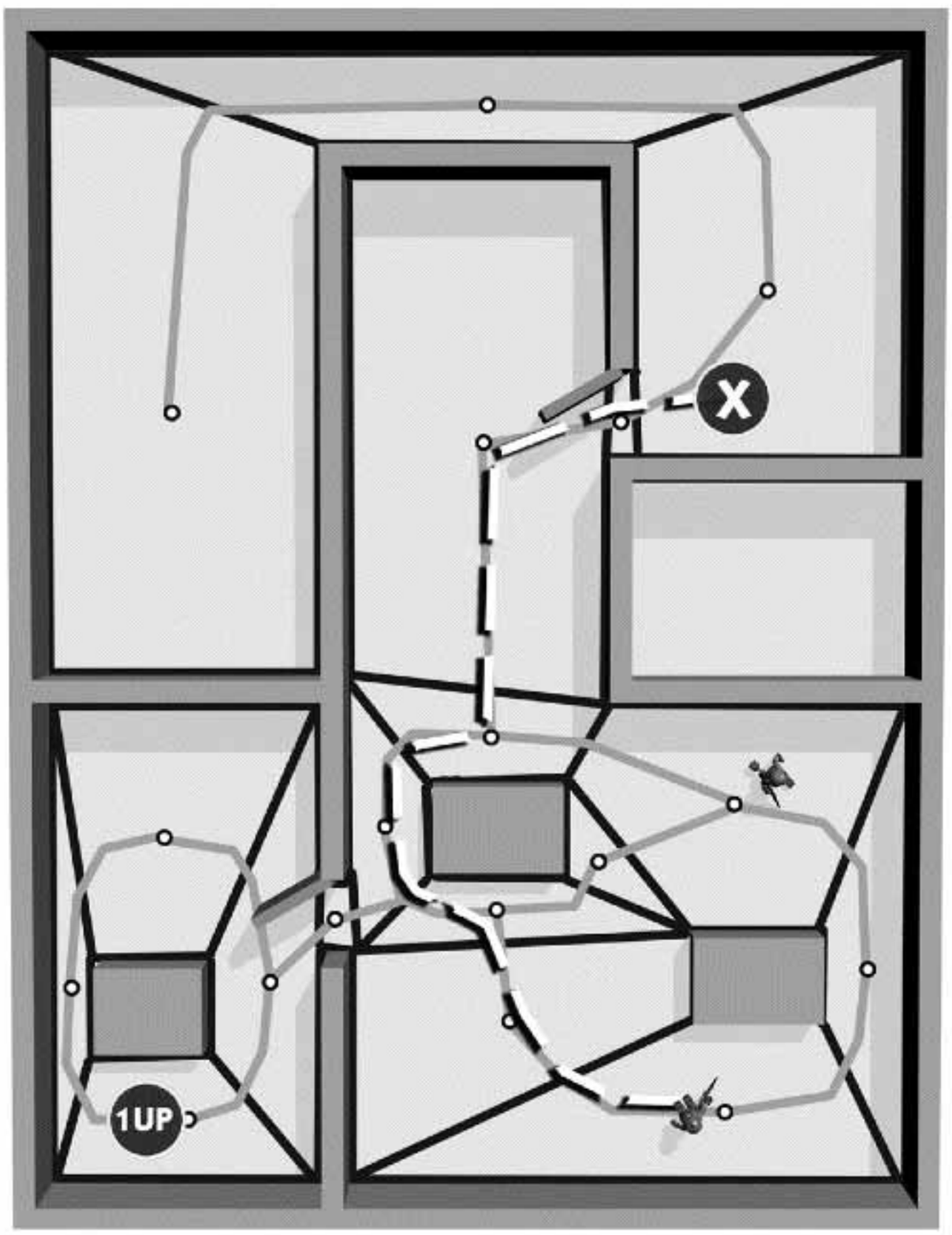


FIGURE 2.5 Navigation mesh systems.

Combination Systems

Some games use a combination of these techniques. Relatively open worlds might use a navigation mesh, but have underground passages that rely on path node networks. Games with lots of organic creature movement (like flocks of birds, or herds of animals) might use a potential fields solution to accentuate the group behavior, but have a fixed pathfinding system for more humanoid creatures, or a special network of nodes that only UFOs can use when flying in the air. By combining, you get the advantage of not having to overtax any one part of the system because you're using that system only for what it does best. You can then rely on another technique when the first one breaks down.

Obstacle avoidance, on the other hand, is a much simpler navigation task. It involves getting around objects that are in your direct line of travel. Avoidance is akin to *dodging,* in that you are temporarily changing your path to get around objects. The pathfinding system has found you a legitimate path to get you to your target location, but you need to adjust that heading for now because something just got in the way. In this way, dynamic obstacles that appear in the world can be handled separately from the static pathfinding system.

Avoidance is commonly done in a couple of different ways:

Potential fields. If you are already using the potential fields method of pathfinding, you could use a similar method for avoidance, by just having the various dynamic obstacles apply an additive *repellant* force away from themselves in all directions centered on the obstacle's current position, so that anybody getting too near to them will turn away. Make the force get stronger as you get closer, until it finally stops all movement of the invader at some minimum distance.

Steering behaviors. Back in 1987, Craig Reynolds released a paper [Reynolds87] detailing a system of behaviors for what he called "boids," creatures that moved in groups and had somewhat organic behavior without complex planning. In 1999, he updated his research by releasing another paper entitled, "Steering Behaviors for Autonomous Characters," [Reynolds99] and games have been borrowing from it ever since. In it, he illustrated that with only a few mathematical forces you could very easily simulate realistic motion patterns for AI controlled characters. Although the most popular application of Reynolds's techniques have been in the implementation of "flocking" systems (dealing with large groups of creatures, such as birds and fish), the same system can be used for general movement including avoidance. The system uses very simple sensors to determine possible collisions, and then reacts accordingly with basic vector math that is quite easy to understand and put into practice.

TeamLRN

There are many, many articles and papers on pathfinding. So many early games did this task poorly, and were taken to task by critics, that this AI task is actually one of the more heavily explored problems in the AI world. This book will not be delving into implementation of specific pathfinding systems, but see the companion CD-ROM for links to materials concerning this important AI engine subsystem.

BRINGING IT ALL TOGETHER

By taking all of these considerations into account, and noting the strengths and weaknesses of the different AI techniques (as described in later parts of this book), you will assuredly find a solution to your game's AI needs.

The basic steps involved in AI engine design are thus:

- Determine the different sections to your AI system, and allow that you might have to treat these different parts as completely separate pieces to your engine, each needing a specific AI technique to solve. Some of this is genre specific. If you will be coding on a straightforward fighting game, you might only need one real type of AI system. Maybe you are working on a fighting game, most work is data driven, and the character scripters will be doing the brunt of that. But if you're going to be coding a large RTS, you might need several subsystems to accomplish the many levels of AI that encompass this genre.
- Determine the types of inputs to the system, be they just digital (on/off), some series of enumerative states, full floating-point analog values, or any combination of these.
- Determine the outputs that the system will use. Along the same lines as the inputs, you may have very distinct outputs, like playing a specific animation or performing in a very constrained behavior. You could have a number of analog outputs, such as speed, where you can be at 1.5 mph or 157.3 mph. But you might also have layered outputs; an example would be characters that can play different animations for the upper and lower parts of their body. This character's lower half might be connected strongly with movement, whereas his upper body could then be concerned mostly with holding a weapon, and aiming, or playing some taunt animation. In effect, you are now governing two outputs concurrently, and they are being layered onto the character in some way.
- Determine the primary logic you are going to need to link the inputs to the outputs. Do you have real, hard, and steadfast rules? Do you have very general rules and a ton of exceptions? Do you have no rules at all, and merely modes that can layer onto each other to convey an overall logic? All of these are prevalent in today's games.

- Determine the type of communications between objects in your game, between the AI systems you might need to code, and between the other game systems. Are you going to need continuous communication, or a more event-driven situation? Are you going to be getting back multiple messages from things within any particular game tick every so often, or almost always? These types of considerations will give you a list of additional requirements that you will need from your individual AI entities.
- Take note of all the other limitations that your game will endure. Platform specific concerns are a big category here. But so is schedule length, which is a hard one to deal with when you're first tackling an AI project. There are so many places to get tangled, and the high-level nature of AI work means that you're also relying on other people in the team to provide you with technology or art resources along the way. You have to be reasonable about the amount of work that you can accomplish, given these types of concerns, but also remembering that if you work yourself into the ground, you'll go crazy or burnout.
- At this point, you can consider the pros and cons of each AI technique, as detailed in Parts III and IV of the book, and we hope you will find something that you can use to implement your system. If you can't, it might be because you haven't broken the problem down enough and are trying to tackle too large a chunk at a time. Try looking at the system (or subsystem) that you are designing, and ensure that it is atomic enough so you aren't trying to pack too much functionality into a single AI technique, and choking it with complexity or exceptions.

Theory will only get you so far, however. Take the skeletal code and do some prototyping in your game. You might find specific failings from a particular method, discover that it is difficult to scale the technique to the level you require, or need additional elements for side AI issues. Consider this prototyping to be a part of the design phase of your AI engine. It will help you find holes in your plan, as well as break up the somewhat tedious task of class and structural design. Your final product will be better for it.

SUMMARY

This chapter covered the foundation systems inherent in a game AI engine and described the primary points to consider when designing and building an engine.

- The three main portions of an AI engine are decision making, perception, and navigation.

TeamLRN

- The type of decision-making technique you use should rely on game-specific factors like types of solutions, agent reactivity, system realism, genre, special content, platform, and development and entertainment limitations.
- Perception systems are usually central locations for input data calculations for the AI characters. By keeping it central, the AI system prevents excessive recalculation and aids debugging and development.
- Perception systems can also take into account low-level details, including update regularity, reaction time, thresholds, load balancing, and computation cost and preconditions.
- Navigation systems for game AI usually fall into one of four main paradigms: grid based, simple avoidance and potential fields, map node networks, and navigation meshes. Some games use combinations of these hierarchically. Obstacle avoidance is a more local system dealing with short-term goals.
- When designing your AI system, use the process of: breaking down the overall system into sections, determine inputs and input types, determine outputs and output types, determine logic needed to unite the two, determine communication types needed, determine other system limitations, and *then* consider the attributes of each AI technique. If you're having trouble fitting a system into a technique, you might need to simplify (by subdividing) the current system you're working on, or maybe a different technique will be better.
- Prototyping your AI system as part of the design phase will help to ensure that your system is flexible enough to handle everything you will need from it, and quickly point out holes in design or implementation, which will be much more easily fixed before the full production cycle is under way.

3 AIsteroids: Our AI Test Bed

In this chapter, you will be introduced to the small application that will become the test bed for the various AI techniques, AIsteroids. As the name implies, it is a very simplified version of an *Asteroids*-style game, with only rocks (represented by circles), an AI or human-controlled ship (represented by a triangle), and powerups that increase your shot power (represented by squares) to begin with. The ship can turn, thrust (forward and reverse), use "hyperspace," and shoot. Later we will incorporate additional elements, such as other alien craft and different weapons and powerups, as the need arises to show off AI techniques. This application was picked because of its simplicity and because the various AI methods could be implemented within the program easily.

To begin with, we will cover the basic program, including the class diagram, and a clear dissection of the code involved.

Figure 3.1 shows the layout of the various classes used. It is a fairly flat hierarchy, with only one major base class, the `GameObj`. The dynamic objects in the game—asteroids, bullets, explosions, powerups, and ships—are all `GameObj` children. This allows the `GameSession` class, which is the main game logic class, to have a complete list of `GameObjs` that it can act on. There are three other main files: `Aisteroids.cpp`, which is the main loop as well as the OpenGL Utility Toolkit (GLUT) initialization code, and the `utility.cpp` and `utility.h` files, which include some global functions. These functions include some useful math functions, several game-related defines, and functions for drawing text to the screen under GLUT.

THE `GameObj` CLASS

As shown in Listing 3.1, the `GameObj` class is very straightforward. The class encapsulates object creation, collision (both checking for collisions and special code that needs to run in the event of a collision), basic physical movement, and `Draw()` and `Update()` functions. `Explode()` handles spawning of explosions for object types that explode when they collide.

TeamLRN

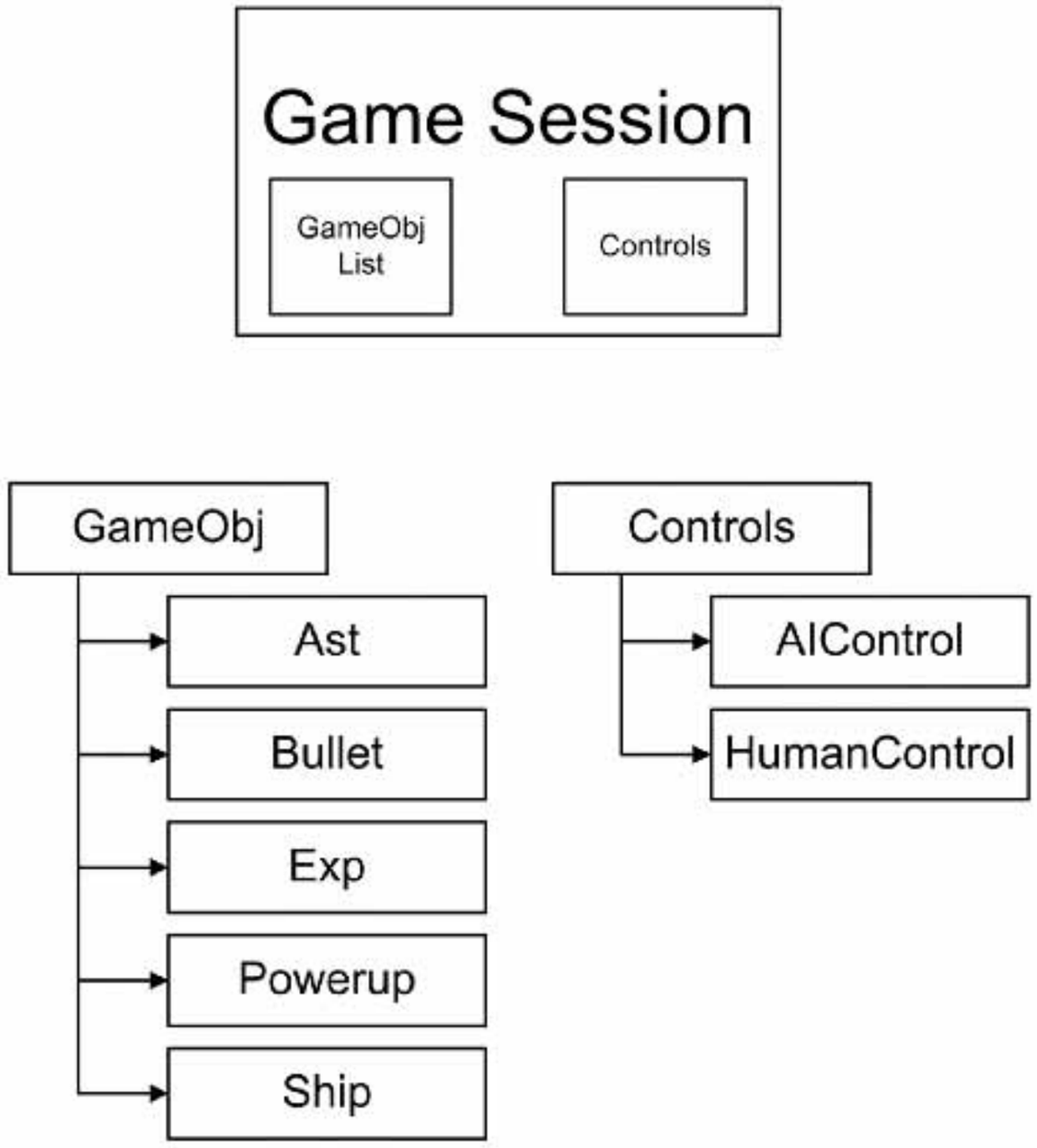


FIGURE 3.1 Alsteroids class structure.

Note the enumeration for object types. They have been made bitwise values instead of a straight integer enumeration so that the code can also use these types for collision flags. Each object must register for the specific object types it will collide with, and this bitwise representation allows an object to register collisions with multiple object types. Collisions for all game objects are handled with simple collision spheres that test for intersection.

Also, notice that by default a plain `GameObj` does not draw, explode, or perform any special code at collision time. Children of this class must override these member functions to facilitate each action.

LISTING 3.1 Header for the `GameObj` Class

```
class GameObj
{
public:
     //constructors/functions
     GameObj(float _size = 1);
     GameObj(const Point3f &_p, const float _angle,
             const Point3f &_v);
```

```
virtual void Draw(){}
virtual void Init();
virtual void Update(float t);
virtual bool IsColliding(GameObj *obj);
virtual void DoCollision(GameObj *obj) {}
virtual void Explode() {}

//unit vector in facing direction
Point3f UnitVectorFacing();
Point3f UnitVectorVelocity();


enum//collision flags/object types
{
     OBJ_NONE       = 0x00000001,
     OBJ_ASTEROID   = 0x00000010,
     OBJ_SHIP       = 0x00000100,
     OBJ_BULLET     = 0x00001000,
     OBJ_EXP        = 0x00010000,
     OBJ_POWERUP    = 0x00100000,
     OBJ_TARGET     = 0x01000000
};

//data
Point3f        m_position;
Point3f        m_axis;
float          m_angle;
Point3f        m_velocity;
float          m_angVelocity;
bool           m_active;
float          m_size;
Sphere3f       m_boundSphere;
int            m_type;
unsigned int   m_collisionFlags;
int            m_lifeTimer;

};
```

THE GameObj UPDATE FUNCTION

Listing 3.2 is the base class update function, which updates the base physics parameters, `m_position` and `m_angle`, and decrements the optional `m_lifeTimer`, which is a generic way of having game objects simply last for a set period of time, and then

TeamLRN

automatically remove themselves from the world. This is used for bullets, explosions, and powerups.

LISTING 3.2 The Base Game Object `Update()` Function

```
//--------------------
void GameObj::Update(float dt)
{
    m_velocity  += dt*m_accelleration;
    m_position  += dt*m_velocity;
    m_angle     += dt*m_angVelocity;
    m_angle      = CLAMPDIR180(m_angle);

    if(m_position.z() !=0.0f)
    {
        m_position.z() = 0.0f;
    }

    if(m_lifeTimer != NO_LIFE_TIMER)
    {
        m_lifeTimer -= dt;
        if(m_lifeTimer<0.0f)
            m_active=false;
    }
};
```

THE SHIP OBJECT

The `ship` object is a `GameObj`, with the addition of controls and the ability to fire bullets. Listing 3.3 shows the class header. Some functions constitute the controls of the craft, and others h;andle the bullet firing and bookkeeping. The `m_invincibilityTime` integer is for the initial period of invincibility when a level starts, or the main ship respawns. The variable `m_shotPowerLevel` is an accumulator for powerups that affect your shooting power level. If you were to create additional powerup types, you would probably want to also give the structure accumulator variables for those as well. The `Update()` function is only mildly different from the base class; the `Update()` function checks to see if `m_thrust` is true, and if so, calculates an acceleration, and then updates velocity, position, and angle. The function also updates the `m_invincibilityTime`, if it still has time left.

LISTING 3.3 The Ship Class Header

```
class Ship : public GameObj
{
public:
     //constructor/functions
     Ship();
     virtual void Draw();
     virtual void Init();
     virtual void Update(float t);
     virtual bool IsColliding(GameObj *obj);
     virtual void DoCollision(GameObj *obj);

     //ship controls
     void ThrustOn(   {m_thrust=true;}
     void ThrustOff() {m_thrust=false;}
     void TurnLeft()  {m_angVelocity=180.0;}
     void TurnRight() {m_angVelocity=-180.0;}
     void StopTurn()  {m_angVelocity=0.0;}
     void Stop();
     void Hyperspace();

     //Powerup Management
     virtual void GetPowerup(int powerupType);
     int GetShotLevel() {return m_shotPowerLevel;}
     int GetNumBullets(){return m_activeBulletCount;}
     void IncNumBullets(int num = 1){m_activeBulletCount+=num;}
     void MakeInvincible(float time){m_invincibilityTimer = time;}

     //bullet management
     virtual int MaxBullet();
     void TerminateBullet(){if(m_activeBulletCount > 0)
                              m_activeBulletCount--;};
     virtual void Shoot();
     virtual float GetClosestGunAngle(float angle);

     //data
     Control* m_control;
private:
     int     m_activeBulletCount;
     Point3f m_accelleration;
     bool    m_thrust;
```

TeamLRN

```
        bool    m_revThrust;
        int     m_shotPowerLevel;
        float   m_invincibilityTimer;
};
```

THE OTHER GAME OBJECTS

`Exp` (explosions) and `Powerup` are very simple objects that simply instantiate, last for some period of time, and then disappear. If a ship collides with a powerup, however, that ship's `GetPowerup()` function will be called. Asteroids don't have a maximum lifetime, and the only additional logic is that they split apart when struck by a bullet, if they're big enough. The target object is for debugging (unless you wanted to implement it for something else, such as homing missiles possibly), and is simply a game object with no logic that displays itself as an X.

Bullets require one further collision step, as shown in Listing 3.4.

LISTING 3.4 The `Bullet` Special Collision Code

```
void Bullet::DoCollision(GameObj *obj)
{
     //take both me and the other object out
     if(obj->m_active)
     {
          obj->Explode();
          obj->DoCollision(this);
     }

     m_active=false;
     if(m_parent)
     {
          Game.IncrementScore(ASTEROID_SCORE_VAL);
          m_parent->TerminateBullet();
     }
}
```

In this simple function, the bullet also increments the score, and calls its parent's `TerminateBullet()` function (this depends on whether you set this bullet to have a ship parent because bullets can be freely instantiated as well), which just decrements the number of shots that ship has active. The bullet will also kill off the other object that it collides with. The general collision system only calls the `Explode()` and `DoCollision()` functions for the first object in the collision, for

optimization reasons. Therefore bullets, which require both objects to run collide code, need this special case consideration.

THE GameSession CLASS

The overall game structure is shown in Listing 3.5. Most of the class is public because it will just be accessed by the main game functions anyway. The game is divided into a few high level states, `STATE_PLAY`, `STATE_PAUSE`, `STATE_NEXTWAVE`, and `STATE_GAMEOVER`. These are very basic game flow states and serve only as modifiers to the draw and control codes. For this demonstration program, there are two `Control` classes that are instantiated, a `HumanControl` class that handles the keyboard events, and an `AIControl` class, which for right now does nothing but will eventually be where we put our AI code for the game.

LISTING 3.5 The GameSession Class Header

```
typedef std::list<GameObj*> GameObjectList;
class GameSession
{
public:
     //constructor/functions
     GameSession();
     void Update(float dt);
     void Draw();
     void DrawLives();
     void Clip(Point3f &p);
        void PostGameObj(GameObj*obj)
          {m_activeObj.push_back(obj);}

     //game controls
     enum
     {
          CONTROL_THRUST_ON,
          CONTROL_THRUST_REVERSE,
          CONTROL_THRUST_OFF,
          CONTROL_RIGHT_ON,
          CONTROL_LEFT_ON,
          CONTROL_STOP_TURN,
          CONTROL_STOP,
          CONTROL_SHOOT,
          CONTROL_HYPERSPACE,
          CONTROL_PAUSE,
```

TeamLRN

```
        CONTROL_AION,
        CONTROL_AIOFF
};
void UseControl(int control);

//score functions
void IncrementScore(int inc){m_score += inc;}
void ResetScore()           {m_score = 0;}

//game related functions
void StartGame();
void StartNextWave();
void LaunchAsteroidWave();
void WaveOver();
void GameOver();
void KillShip(GameObj *ship);

 //data
   Ship*         m_mainShip;
   HumanControl* m_humanControl;
   AIControl*    m_AIControl;

bool m_bonusUsed;
int  m_screenW;
int  m_screenH;
int  m_spaceSize;
float  m_respawnTimer;
float  m_powerupTimer;
int  m_state;
int  m_score;
int  m_numLives;
int  m_waveNumber;
int  m_numAsteroids;
bool m_AIOn;

enum
{
     STATE_PLAY,
     STATE_PAUSE,
     STATE_NEXTWAVE,
     STATE_GAMEOVER
};
```

```
private:
     GameObjList m_activeObj;
};
```

The list of dynamic objects for the game is stored in a Standard Template Library (STL) `list` structure called `m_activeObj`. This program was written for simplicity, so it does things like `new` and `delete` memory while in game, whereas most real games try to achieve a solid memory allocation beforehand (possibly by allocating a large pool of the different `GameObj` structures, and just using them as needed), to prevent memory fragmentation. By placing all the game objects in this structure, the `Update()` function for `GameSession` is generic and straightforward. The discussion of this function will be shown split into eight parts, so that each part of the update can be discussed separately. See Listings 3.6.1 through 3.6.8.

Primary Logic and Collision Checking

Listing 3.6.1 is the primary part of the update loop. It sets up a `for` loop to iterate through all the game objects, and then for each object, runs its `Update()`method and clips its position to the viewport (which also wraps the position around, asteroids style). The function then checks for any collisions with other objects, by looping through the objects and calling `IsColliding()` on them. These calculations are optimized by the following:

1. An object must be registered to collide by having its `m_collisionFlags` variable not contain the `GameObj::OBJ_NONE` bit.
2. The object will only do collision checks against objects of the types it is registered for.
3. An object cannot collide with another object that isn't active.
4. Objects cannot collide with themselves.

LISTING 3.6.1 GameSession's Update Loop, Section 1: Update and Collision Checking

```
void GameSession::Update(float dt)
{
     GameObjectList::iterator list1;
     for(list1=m_activeObj.begin();
         list1!=m_activeObj.end();++list1)
     {
          //update logic and positions
          if((*list1)->m_active)
             {
```

TeamLRN

```
            (*list1)->Update(dt);
            Clip((*list1)->m_position);
        }
        else continue;

        //check for collisions
        if((*list1)->m_collisionFlags !=
           GameObj::OBJ_NONE)
        {
            GameObjectList::iterator list2;
            for(list2=m_activeObj.begin();
                list2!=m_activeObj.end();++list2)
            {
                //don't collide with yourself
                if(list1 == list2)
                  continue;


                if((*list2)->m_active          &&
                   ((*list1)->m_collisionFlags &
                    (*list2)->m_type)          &&
                  (*list1)->IsColliding(*list2))
                {
                    (*list1)->Explode();
                    (*list1)->DoCollision((*list2));
                }
            }
        }
        if(list1==m_activeObj.end()) break;
    }//main for loop
}
```

Object Cleanup

Listing 3.6.2 shows the code necessary so that objects that have had their `m_active` field set to false (by either colliding with something they are destroyed by, or by outliving their life counter variable) are simply removed from the list by means of the `remove_if STL` operator, and then erased. The functor that checks for the inactive condition (`RemoveNotActive`) is also in charge of deleting the memory taken up by the object; the `erase` function just takes it out of the object list.

LISTING 3.6.2 GameSession's Update Loop, Section 2: Killed Object Cleanup

```
//get rid of inactive objects
     GameObjectList::iterator end    = m_activeObj.end();
     GameObjectList::iterator newEnd =
                remove_if(m_activeObj.begin(),
                    m_activeObj.end(),RemoveNotActive);
     if(newEnd != end)
          m_activeObj.erase(newEnd,end);
```

Spawning Main Ship and Powerups

Listings 3.6.3 and 3.6.4 are simple parts of the update function that check a couple of timers, `m_respawnTimer` and `m_powerupTimer`. The respawn timer is used when the main ship has been destroyed; it makes a small pause before respawning. This is so the player has time to realize he exploded. Later, when we're using an AI-controlled main ship, we'll probably reduce this value because the AI does not need this down-time. The powerup timer provides for the pause between each powerup spawning. If this pause time is up, the game spawns a new powerup with random position and velocity and adds it to the main object list.

LISTING 3.6.3 GameSession's Update Loop, Section 3: Respawn Main Ship

```
//check for no main ship, respawn
     if(m_mainShip == NULL || m_respawnTimer>=0)
     {
          m_respawnTimer-=dt;
          if(m_respawnTimer<0.0f)
          {
               m_mainShip = new Ship;
               if(m_mainShip)
               {
                    PostGameObj(m_mainShip);
                    m_humanControl->SetShip(m_mainShip);
                    m_AIControl->SetShip(m_mainShip);
               }
          }
     }
```

LISTING 3.6.4 GameSession's Update Loop, Section 4: Spawn Powerups

```
//occasionally spawn a powerup
     m_powerupTimer -=dt;
     if(m_powerupTimer <0.0f)
```

TeamLRN

```
{
        m_powerupTimer = randflt()*6.0f + 4.0f;
        Powerup* pow = new Powerup;
        if(pow)
        {
                pow->m_position.x()= randFlt()*m_screenW;
                pow->m_position.y()= randFlt()*m_screenH;
                pow->m_position.z()= 0;
                pow->m_velocity.x()= randFlt()*40 - 20;
                pow->m_velocity.y()= randFlt()*40 - 20;
                pow->m_velocity.z()= 0;
                PostGameObj(pow);
        }
}
```

Bonus Lives

Listing 3.6.5 does a simple score check, and every 10,000 points, it awards the player another life. This is pretty straightforward code and is a very common practice in these kinds of games.

LISTING 3.6.5 GameSession's Update Loop, Section 5: Bonus Lives

```
//check for additional life bonus each 10K points
        if(m_score >= m_bonusScore)
        {
                m_numLives++;
                m_bonusScore += BONUS_LIFE_SCORE;
        }
```

End of Level and Game

The next two listings (3.6.6 and 3.6.7) check for two important game conditions, the end of the current level (determined by there being no asteroids left for the player to shoot), and end of the game (determined by not having any more lives left). Each of these conditions calls a function, `WaveOver()` or `GameOver()`, which sets some critical flags and sets the game state to either `STATE_NEXTWAVE` or `STATE_GAMEOVER`.

LISTING 3.6.6 GameSession's Update Loop, Section 6: End of Level

```
//check for finished wave
        if(!m_numAsteroids)
        {
```

```
            m_waveNumber++;
            WaveOver();
        }
```

LISTING 3.6.7 GameSession's Update Loop, Section 7: Game Over

```
    //check for finished game, and reset
        if(!m_numLives)
            GameOver();
```

THE Control CLASS

To give commands to a ship, the game requires the use of a `Control` class. The base class contains the barebones structure, including `Update()`, `Init()`, and an `m_ship` pointer to the controlled ship. This class will be the parent to both the human control system (`HumanControl`) and to the AI (`AIControl`). For now, because the human control scheme is always the same, the `HumanControl` class is a bit different in that it doesn't use its update function but, rather, is just the depository for the global callbacks that the program passes to GLUT to perform keyboard checks. If the game were more complex, we could implement a state-based control scheme (or some other way of separating the system functionality) for the human player and, thus, require the full functionality of a `Control` class. Later in the book, when we implement the various AI methodologies, we will start by creating specific `AIControl` classes to house the particulars of each AI method.

THE AI SYSTEM HOOKS

In listing 3.6.8, the `GameSession` checks to see if the AI system is turned on, and if so, the `Update()` function for the `AIControl` class is called. This update function is stubbed out in `AIControl.cpp`, so the AI system does nothing now. Again, this is just the framework for the future implementations of each AI technique. We will later make child classes of this barebones `AIControl` class that will run specific code for each technique.

The only other additions to this base class are some debug data fields (which were used in developing the demo programs in this book and left to serve as a good start for any additional debugging information). You should always include debugging system hooks in your design right from the start because you will spend precious time during development trying to reverse engineer your AI engine to simply output text strings or on screen visual debugging elements.

TeamLRN

There are two update functions, the regular `Update()` and `UpdatePerceptions()`, which is where system-level data objects should be updated. These functions are separated to give the greatest flexibility to the system, as well as being more organizationally sound. Figure 3.2 shows a screenshot of the test bed running the finite state machines (FSM) AI system from Chapter 15.

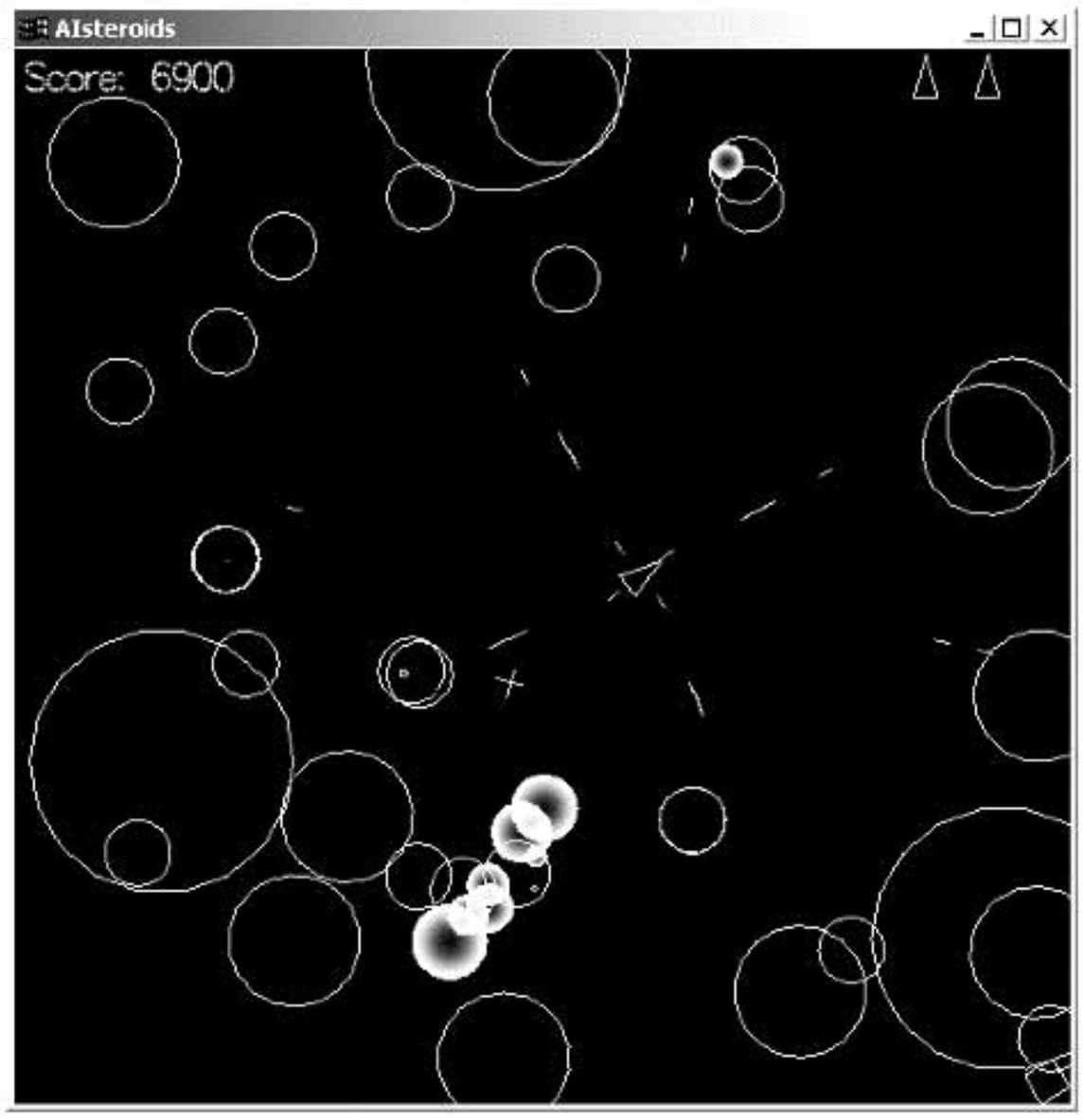


FIGURE 3.2 AIsteroids screenshot.

LISTING 3.6.8 GameSession's Update Loop, Section 8: AI Update Hook

```
        //update AI control, if turned on
         if(m_AIOn)
              m_AIControl->Update(dt);

    }//end of GameSession::Update()
```

GAME MAIN LOOP

`AIsteroids.cpp` is the main game file for the project. It initializes GLUT and sets up the callback pointers for updating the game, drawing the game, and handling all the input from Windows or the user (the global functions that handle the keyboard are in the `HumanControl.cpp` file).

SUMMARY

This chapter described the primary test-bed application the book will use for implementing each AI technique in Parts III and IV. The overall class structure was discussed, as were the notable sections of the code.

- `GameObj` is the basic game object class. It takes care of physics and handles object drawing and updating.
- The current objects in the game include asteroids, bullets, explosions, powerups, ships, and a debugging target object.
- `GameSession` is the singular game class. It takes care of all the variables and structures needed to run a game. It is the primary update and draw functions for the game. It spawns all additional game elements and manages object-to-object collision checking.
- `Aisteroids.cpp` is the main loop file, and it includes all the initialization of GLUT and all callbacks for running the game.
- The `Control` class handles the logic for a ship object. This logic can be in the form of an AI technique or keyboard functionality for a human player.
- The `AIControl` class will be the branching point for our AI to hook into the system. By overriding the class with a specific AI method class (e.g., `FSMAIControl`, discussed in Chapter 15), we can use this game application with CPU-controlled opponents. The keyboard control will still be enabled, but this is to facilitate the application as a test bed because we still want to be able to send keyboard events to the game when the AI system is running.

TeamLRN

Part II Game Genres

In the following chapters, the various game genres will be dissected, and we will discuss the various AI techniques that work best with each genre, and why. The following aspects of each genre will be considered:

- The elements of each type of game that require some form of AI.
- The common AI techniques used with each genre, including examples of finished products. The specifics of the technique will be left for the later parts of the book.
- Real-world examples.
- Notable exceptions, if any.
- Areas that need improvement, or that have traditionally been lacking.

Again, although definitely not all-inclusive, the following will give you a good idea about the AI requirements of different types of games and the commonly used solutions that AI programmers have employed to satisfy those requirements.

TeamLRN

4 Role-Playing Games (RPGs)

As soon as personal computers started to become more popular, so too did the flood of games that took advantage of the more complex control scheme inherent in having a keyboard, instead of a joystick and a button. RPG games were among the first of the new game genres made possible by the personal computer. Before then, arcade games were the prevalent game type, with the early home systems basically just porting over all the popular arcade games. The gameplay in arcade games was designed to be over quickly, to get more quarters, so the notion of a game that takes a very long investment in time and effort was a complete departure from the norm.

The earliest RPGs were either text based (like *Adventure* or *Wumpus*) or were crafted art out of ASCII characters (like *Rogue* and *NetHack,* see Listing 4.1 for a code snippet from *NetHack*—notice how the listed function, which is a generic method for determining and defining missile attacks from an AI controlled enemy, takes into account different perceptions, makes case based determinations based on attributes and skills, and makes final behavior decisions). Finally, the graphical RPG came out and the rest is history. Newer consoles that had increased in power (as well as having some kind of memory system, either by way of a card the player could store information on, or some kind of on-cartridge storage) jumped on board, and the console RPG began to branch off. Console RPGs were primarily more real-time combat-based because consoles have more action-oriented input controllers (although games like the *Final Fantasy* series or *Phantasy Star* games were turn based), and they tended to be less mature in nature (meaning, the main character was usually 15 or so, instead of the hardened warriors found in computer RPGs) because the primary audience for console games is generally younger than that of computer games. Nowadays, both console and computer RPGs have blurred the platform line, with games like *Diablo* being a computer game with simple, console-like action-oriented gameplay; and the new online persistent RPGs on the consoles are all but identical to their personal computer brothers.

TeamLRN

LISTING 4.1 Code Snippet from the Open Source ASCII RPG, *NetHack*. Distributed under the NetHack GPL.

```
/* monster attempts ranged weapon attack against player */
void
thrwmu(mtmp)
struct monst *mtmp;
{
    struct obj *otmp, *mwep;
    xchar x, y;
    schar skill;
    int multishot;
    const char *onm;

/* Rearranged beginning so monsters can use polearms not in a
        line */
if (mtmp->weapon_check == NEED_WEAPON || !MON_WEP(mtmp)) {
    mtmp->weapon_check = NEED_RANGED_WEAPON;
    /* mon_wield_item resets weapon_check as appropriate */
    if(mon_wield_item(mtmp) != 0) return;
}

/* Pick a weapon */
otmp = select_rwep(mtmp);
if (!otmp) return;

if (is_pole(otmp)) {
    int dam, hitv;

    if (dist2(mtmp->mx, mtmp->my, mtmp->mux, mtmp->muy) >
            POLE_LIM ||
    !couldsee(mtmp->mx, mtmp->my))
return;    /* Out of range, or intervening wall */

if (canseemon(mtmp)) {
onm = xname(otmp);
pline("%s thrusts %s.", Monnam(mtmp),
      obj_is_pname(otmp) ? the(onm) : an(onm));
}
```

```
    dam = dmgval(otmp, &youmonst);
    hitv = 3 - distmin(u.ux,u.uy, mtmp->mx,mtmp->my);
    if (hitv < -4) hitv = -4;
    if (bigmonst(youmonst.data)) hitv++;
    hitv += 8 + otmp->spe;
    if (dam < 1) dam = 1;

    (void) thitu(hitv, dam, otmp, (char *)0);
    stop_occupation();
    return;
}

x = mtmp->mx;
y = mtmp->my;
/* If you are coming toward the monster, the monster
 * should try to soften you up with missiles.  If you are
 * going away, you are probably hurt or running.  Give
 * chase, but if you are getting too far away, throw.
 */
if (!lined_up(mtmp) ||
    (URETREATING(x,y) &&
        rn2(BOLT_LIM - distmin(x,y,mtmp->mux,mtmp->muy))))
    return;

skill = objects[otmp->otyp].oc_skill;
mwep = MON_WEP(mtmp);        /* wielded weapon */

/* Multishot calculations */
multishot = 1;
if ((ammo_and_launcher(otmp, mwep) || skill == P_DAGGER ||
    skill == -P_DART || skill == -P_SHURIKEN) && !mtmp->mconf) {
/* Assumes lords are skilled, princes are expert */
if (is_prince(mtmp->data)) multishot += 2;
else if (is_lord(mtmp->data)) multishot++;

switch (monsndx(mtmp->data)) {
case PM_RANGER:
    multishot++;
    break;
case PM_ROGUE:
    if (skill == P_DAGGER) multishot++;
    break;
```

TeamLRN

```
case PM_NINJA:
case PM_SAMURAI:
    if (otmp->otyp == YA && mwep &&
    mwep->otyp == YUMI) multishot++;
    break;
default:
break;
}
/* racial bonus */
if ((is_elf(mtmp->data) &&
    otmp->otyp == ELVEN_ARROW &&
    mwep && mwep->otyp == ELVEN_BOW) ||
(is_orc(mtmp->data) &&
    otmp->otyp == ORCISH_ARROW &&
    mwep && mwep->otyp == ORCISH_BOW))
    multishot++;

    if ((long)multishot > otmp->quan)
            multishot = (int)otmp->quan;
    if (multishot < 1) multishot = 1;
    else multishot = rnd(multishot);
}

if (canseemon(mtmp)) {
    char onmbuf[BUFSZ];

    if (multishot > 1) {
    /* "N arrows"; multishot > 1 implies otmp->quan > 1, so
       xname()'s result will already be pluralized */
    Sprintf(onmbuf, "%d %s", multishot, xname(otmp));
    onm = onmbuf;
    } else {
    /* "an arrow" */
    onm = singular(otmp, xname);
    onm = obj_is_pname(otmp) ? the(onm) : an(onm);
    }
    m_shot.s = ammo_and_launcher(otmp,mwep) ? TRUE : FALSE;
    pline("%s %s %s!", Monnam(mtmp),
      m_shot.s ? "shoots" : "throws", onm);
    m_shot.o = otmp->otyp;
} else {
    m_shot.o = STRANGE_OBJECT;
         /* don't give multishot feedback */
}
```

```
    m_shot.n = multishot;
    for (m_shot.i = 1; m_shot.i <= m_shot.n; m_shot.i++)
        m_throw(mtmp, mtmp->mx, mtmp->my, sgn(tbx), sgn(tby),
            distmin(mtmp->mx, mtmp->my,
                        mtmp->mux, mtmp->muy), otmp);
    m_shot.n = m_shot.i = 0;
    m_shot.o = STRANGE_OBJECT;
    m_shot.s = FALSE;

    nomul(0);
}
```

RPGs in general follow a simple formula: start with nothing, perform tasks for treasure and money (mostly killing monsters and going on quests), train your skills, and eventually build yourself into a powerhouse figure that can then right the ultimate wrongs of the land. Some games include a whole party of adventurers, so the player is in effect building up a whole team of characters. Whatever the technical details, the name of the game is immersion: getting the player to identify with the main character, and caring enough to invest the vast amount of time necessary to build the character up and eventually finish the game.

The enemy-filled, constantly hostile world of most RPGs might seem odd, but not if you're a teenager. In a way, young people somewhat *relate* to a character who is solitary in the world, against everyone, universally misunderstood and attacked. It's what gives RPGs their appeal to many of the youth who play them. The inclusion of a small band of party members ties nicely into the clique-ish world of most teens, where you form a small group of intense friends, and extend the "me against the world" fight to include these people as well. This argument is not to say that older or younger people cannot enjoy RPGs but, rather, speaks to a theoretical reason why some people find these types of games popular.

RPGs are fairly AI intensive, mostly because they are usually expansive games, with varying types of gameplay and many hours of gaming experiences per title. As such, the intelligence of the varying game elements has to be higher than most (or at least more heavily scripted) because the sheer number of hours people invest in the game will make any behavioral repetition much more obvious, as well as making small annoyances in AI behavior appear larger. Also, especially on home computers, users demand a minimum of 40 or so hours of gameplay from an RPG. Consoles are a bit lower, usually 20–40. This formula seems to be somewhat fixed in the minds of game players (a strange mix of the approximate amount of time a game can keep a player's interest, and marketing education about how much gameplay you can expect for your money), but there are exceptions, like *Baldur's Gate* for the PC having 100+ hours of play.

Because of these hefty gameplay *quantity* demands, the game had better have a variety of gameplay types (such as puzzles, combat, some kind of crafting, different

TeamLRN

types of travel, etc.) or your primary combat system had better be *very* fun and addicting. The *Diablo* games fall into the latter category. The gameplay is very repetitive, but also very addictive. Some have theorized that the game somehow awakens our inherent "hunter-gatherer" lineage, and we just can't stop clicking the mouse.

COMMON AI ELEMENTS

Enemies

Most populations of most RPG worlds are enemies. An almost endless supply of enemies is needed to provide the player with something to dispatch and get experience points, money, and powerful new items. RPGs in the past used almost exclusively what can be described as *statistical* AI, in that the attributes of the monsters determined everything about them, the attacks they use, the way they fight, how tough they are in general, what treasure they drop when they die, and so on. Today's games go a bit farther and have usually have enemies that are a bit more hand tailored. These enemies also use more complex behavior patterns, including running away, healing themselves, fighting in groups by surrounding you and using complementary attack methods, and so forth.

But because enemies in RPGs usually come in such numbers during a game, the AI is specifically set up to be pretty much *A* and not much *I*. Turn-based RPGs of the past (*Bard's Tale, Phantasy Star, Chrono Trigger*), the so-called real-time combat RPG (*The Legend of Zelda,* the later *Ultima*™ games, *Diablo, Terranigma*), and the fusion variants brought about recently (*Baldur's Gate* or *Icewind Dale,* which are real-time games that can be paused and, thus, made to act turn based) all pretty much rely on the enemies to be combination *containers* (of wealth and experience points) and *obstacles* (by being "walls" of a certain number of hit points that the hero must destroy to get by).

This is done by design, of course. When a player who has spent 60 or more hours playing your game goes into a room and sees a monster approach that looks like an enemy character he has seen before, he should feel one of three ways:

I can beat this guy. I know what attacks he uses, approximately how many hit points he has, and that I have a weapon that affects this enemy.

I think I can defeat this guy. He looks a lot like an enemy I've already fought, but is a different color, or a special name, that makes him unusual and possibly more advanced. In effect, I believe he belongs to an enemy "type," but I'm not sure about his toughness.

I cannot beat this guy. He's too tough, or I don't have the weapon necessary to get through his armor. I know because I've tried before, and failed, or somebody in the game has warned me.

This is just another way of immersing the player in the game and making him feel a part of the world, in that he "knows" the enemies by experience. If a lowly Orc suddenly pulls out a grenade (after futilely running up and using a rusty dagger in the last 50 encounters) and nukes the player, said player is going to feel somewhat cheated. However, this basic guideline can be somewhat sidestepped, if the player is allowed to save the game whenever he wants, or the game actually autosaves quite frequently. In this way, a highly unusual encounter with a special enemy might kill the player, but he won't have lost any time if he has a save. Yes, this leads to more "save, then round the corner, kill one monster, then save" behavior from the player, but it also gives you more freedom to put more of elements of surprise into your random encounters.

Bosses

Bosses are larger, more complex game characters, either humanoid or creature, found at the end of each level (or game world, or subsection) after defeating a horde of lesser enemies. They are usually equivalent to monster *leaders*, the Kings of the Monsters. These are specific, usually unique enemies that break all the previous rules. Players expect to be surprised by the power, skills, weapons, and so forth used by these guys. Bosses are even thought of as treats in the RPG world, and a good boss creature can make up for a lot of game shortcomings, either in the areas of average gameplay, or merely a period of tedious leveling necessary to continue on in the game world.

As such, Boss monsters are usually heavily scripted, with specialty attacks and behaviors that only they perform. Boss monsters also usually communicate with the player, in the form of plot advancing information, or pure invectives. So the AI for these creatures needs to include use of the dialogue system for the game. The *Final Fantasy* series' Boss monsters are a wonder of specialized coding, with encounters that might take hours of real time, complete with various stages of battle and conversation. These encounters are strictly paced by the developers, with planned volleys of your advantage, followed by the enemy's advantage, scripted interruptions with other enemies or special game events, and whatever else they can think up.

Another tried and true Boss tactic involves the "can't be killed . . . yet" Boss. This involves a Boss that can be taken to near death, only to miraculously escape, shouting "I'll be back!" and promising to be bigger and badder next time. Although somewhat trite, this is the gaming equivalent of simple character development, with the Bad Guy developing over the course of the game as much as you are.

Some games use the designation of "sub-boss" to further stratify the monsters in the game, although they are usually just very tough versions of regular creatures, like the unique creatures that heavily populate the *Diablo* series. But even *Diablo*, which many considered an "RPG-lite" click fest, also uses much more specialized

TeamLRN

Boss creatures that employ additional dialogue, animations, spell and weapon effects, and special powers.

The Boss designation also includes the final creature (wizard/god/evil doer) that you will need to defeat to win the game, also called the *End* Boss. This character is very important indeed, and many a good game has received bad marks for having a disappointing or anticlimatic End Boss. The player should have to perform every trick he has learned during the game, and stretch his skills to the limit to destroy this guy, and the End Boss himself should be able to do things that the player has never seen before in the game. He should be tough from a statistics point of view, of course (with lots of hit points and immunities to weapons or spells), but he should also be capable of behaviors beyond the typical. That's why he's the End Boss in the first place.

Nonplayer Characters (NPCs)

NPCs are defined as *anybody in the game that is not a human player.* Usually, however, the term *NPC* refers to people in the game that the human can interact with in ways other than combat. NPCs are the people who inhabit the towns, the half-dead soldiers on the trail who give you valuable clues to the danger ahead, and the occasional old man who offers you money to rescue his daughter. They are usually either of one-shot usage, like the people that are involved in a quest of some sort, or are information dumpers, in that you can keep conversing with them at different points during the game, and they might know something additional about whatever is currently "new" in the game flow.

NPCs, in any case, are generally not very intelligent; they don't have to be. Anything they add beyond information or story advancement is just flavor for the game. However, they also represent one of the largest interfaces the player has to information about the flow of the storyline, as well as being in-game help that can bring a stuck or lost player back into alignment with the objectives of the game. As such, many games have experimented with differing ways of approaching the NPC conversation issue. Some games give the player keywords that represent questions you are posing to the NPC (as in the *Ultima* games), others give the player a choice between a number of sentences that represent the different attitudes you can take against the NPC (like the recent *Baldur's Gate* for the PS2). The evolution of these systems will continue as grammar systems become better, faster, and more generally accepted. Some day, players may converse directly (by speaking) with a general AI NPC who can give wide-ranging responses by indexing his knowledge base and forming sentences on the fly. Until then, we do what we can.

Shopkeepers

Shopkeepers are special NPCs that do business with the player—buying and selling gear, teaching the player new skills, or what have you. Shopkeepers usually aren't much smarter than regular NPCs, but they get special mention because they usu-

ally have extended interfaces, which in turn require special code so they seem intelligent and usable. Sometimes shopkeepers might be part of a scripted quest or game sequence, in that they only become shopkeepers later in the game, or after a task has been completed. A shopkeeper thus might have a notion about whether or not he likes the player, which would then affect his attitude, and prices, when dealing with that player. Some games have a general charisma attribute for characters within the game (or some derivative; the meaning is "How well other people perceive you naturally," considering first impressions, your looks, and your speaking ability), as well as some form of a reputation system that represents a sort of total number depicting the amount of good versus evil deeds you have, as well as flags representing specific things you've done that NPCs can notice and respond to.

There is a natural human tendency to give inanimate things human qualities, and this tendency is tied directly to the amount of time we have to spend dealing with something, and in correlation with how much that something has cost us. Very few people would attribute human qualities to their shoes, but most people have named their cars, know its gender, know how to identify if it's having a bad day, and will even plead with it if it isn't running well. Both objects (shoes and cars) do roughly the same thing: help protect our bodies from the rigors of traveling, so why the disparity? The answer is obvious. With no moving parts, and a simple procedure that we learned when we were three years old, we put on our shoes in the morning, and forget about them. Buying a new pair doesn't require a credit check. Our cars are exactly the opposite. The same is true with Shopkeeper AI. If you have a one-shot NPC within your game, you can pretty much do whatever you want with his behavior, dialogue, and interactions with the player. The player isn't expecting much and will take most things at face value. But with a shopkeeper, especially one that the player will have to use for a large part of the game, every nuance, reply, and animation frame will be carefully watched, memorized, and humanized. Have a bartering system (which in reality takes the player's charisma score, adds in a random factor, and determines a small discount that a player can bargain for) within your game? Over time, a human player will start to imagine intricate rules involving the order of the items he does business with, the time of day, the shopkeepers' moods, and a host of other factors that don't exist. It is precisely this human tendency that allows game makers to get away with so little detail in their games because the human will fill in all the complexity where there is none. The lesson is that shopkeepers do more than provide your players with an economy interface; they also give richness to the world and provide the player with other facets of the game to consider.

Party Members

Members of your adventuring party are also special NPCs, except that they travel with the player, and are either completely player controlled (in turn-based RPGs, or in later games that allow you to pause the action so you have time to give detailed

TeamLRN

commands) or have AI code associated with them. These AI-based party members need careful coding because stupid party members will drive potential players away quickly. Many of the real-time combat games use simple party AI, so that the player can predict (and rely on) what each party member is going to do during a fight. Another factor to remember with real time combat RPGs is pathfinding. In turn-based combat systems, your party members are just attached to you, or follow you around directly (like the *Final Fantasy* games, or even the early *Bard's Tale*), but in real-time games, they actually have to pathfind to follow you. This can cause trouble if you're in a semi-enclosed space (such as an underground dungeon, for instance), and because they can't squeeze by somebody, certain party members go running off to take some extra-long scenic route around that the pathfinder managed to find. This is supremely frustrating to the player, and it can cause these erroneous party members to run through packs of monsters in other parts of the map, and bring them running into the room behind your "helpful" friends to join in the fight. Here's a place where an intelligent party member might say, "Hmm, I can't get around that guy directly to use my sword. But I do have a bow and arrow in my pack, and I'm decent at archery, maybe I'll try attacking that way," or even, "Can't get around directly, so I can't attack. Maybe I should tap my weaker buddy on the shoulder that's being mauled by a creature and *replace* him on the front line." These kinds of "smarts" (rather than blind pathfinding and script following) are the difference between party members, and other guys that the player needs to babysit.

If the characters you adventure with commonly screw up, do the wrong thing, or are constantly getting themselves or you killed, you're not going to want to continue playing with them. *Baldur's Gate* (and its descendents) even allow users to edit the simple scripts that govern the Party members AI, so that users have even more control over this crucial game element. See the section on "Scripting" that follows.

Adding a scripting system to edit party AI is a careful balance. If you make it too easy to use and don't provide enough complexity and functionality, it's worthless. But if the system is too powerful, then it overwhelms the casual gamer, and again becomes worthless to a large part of your audience. A technique that many sports games use to allow players to adjust the AI in their games is to expose specific *tendencies* of behavior as "sliders" (scroll bars that tie to a variable) that the player can set. For sports games, this means that the players could set up a basketball game where the AI never tries to steal the ball, doesn't guard as well, and is better at three point shots all by setting sliders to certain points. A similar system could be used to give more casual gamers access to AI editing without having to write script code. Even some of the complexity of a scripting system, like setting up when specific spells would be cast by an AI mage character, could be represented as sliders that are specific to that spell. This does translate to many potential sliders, but again, it's definitely more accessible to a larger audience than script files are.

USEFUL AI TECHNIQUES

Scripting

Most RPGs are heavily scripted because most of these games follow a very specific storyline. Scripts are used for a variety of game constructs, including dialogue, game event flags, specific enemy or NPC behavior, how the player can interact with the environment, and many others. Scripting is used because most RPGs are linear, or at most branching linear, and as such, work well with the scripted interface. You can design parts of the game to play out almost exactly as designed, with choke points and flags embedded into the scripts so that the players are forced to follow the game flow from point A to point B, even if they first wandered over to points C, D, E, and F in the meantime. Plus, the conversational nature of many RPGs also lends itself to this technique. You can think of scripts as a somewhat data based way of hardcoding the assorted events that come up during the overall story. See Listing 4.2 for an example of a short script from the Black Isle game, *Baldur's Gate.* Here you can see a very basic attack script, which simply determines whether to attack an enemy based on the enemy's distance to the character, and then also determines whether to use a ranged or melee weapon. It does perception checking (the range calculations) as well as perception scheduling (by saying how often the script should be repeated). It also has some randomness, in that the determination for ranged or close combat is determined by a random number (33% of the time, it chooses melee, the rest of the time, it chooses ranged).

LISTING 4.2 Sample Warrior AI User Defined Script from *Baldur's Gate*

```
IF
    // If my nearest enemy is not within 3

    !Range(NearestEnemyOf(Myself),3)

    // and is within 8

    Range(NearestEnemyOf(Myself),8)
THEN
    // 1/3 of the time

    RESPONSE #40
       // Equip my best melee weapon
       EquipMostDamagingMelee()
          // and attack my nearest enemy, checking every 60 ticks
           // to make sure he is still the nearest
```

TeamLRN

```
        AttackReevalutate(NearestEnemyOf(Myself),60)

        // 2/3 of the time

    RESPONSE #80

        // Equip a ranged weapon

        EquipRanged()

        // and attack my nearest enemy, checking every 30 ticks
            // to make sure he is still the nearest

            AttackReevalutate(NearestEnemyOf(Myself),30)
    END
```

Finite-State Machines (FSMs)

The staple of game development, FSMs are useful in RPGs just as they are useful in any game—they allow the developer to split the game into explicit states, in each of which specific characters will perform differently, and manage these with discrete code blocks. Thus, you could have an NPC who first meets you and gives you a quest (for example; state before meeting you: `state_intro`, changing to `state_quest` after giving you information about a quest) and could then, after you finish the quest, become a shopkeeper and sell you things at a discount as a reward (`state_shopkeep`). Note how earlier the script from *Baldur's Gate* is only applicable if an enemy is close by. Any other game state would require additional scripting, or it would simply follow the default script, which is most likely to follow the human around.

By having a state-based system, but scripting the entry and exit to those states, many RPGs hide the hard state transitions. Other games do not, like Nintendo's classic *The Legend of Zelda,* in which the game was split into two distinct states: the overworld and the dungeons of the underworld. The game's music would change, the character himself would look a little different (because of the "lighting"), and if you died, the game acted a little differently (by allowing you to continue in the same dungeon, if you wanted), all because of this basic state change.

Messaging

With so many elements in an RPG world, the need to communicate between entities is very high, so a messaging system is useful in this genre. Information can be passed between party members quickly and easily, facilitating group combat or dialogue. Keys, or whatever your game is using, can message locks to open, and out of place wall stones can cause entire sequences of events to occur when pushed. Be-

cause of the sheer number of uses within an RPG, messaging systems can really give you a lot of flexibility and ease of implementation.

One thing that should be watched for, because it breaks the illusion of play in some titles, is for "instant" messaging being used by the game. Your party kills some creature on the far side of the world. They then teleport back to town (because of a special magic item), and *everyone* back in town already knows about the battle, that you won, and that you're the hero. The townspeople obviously got the message and have switched on the game state specific behavior for it. Wouldn't a better reaction be that the first guy you talk with *doesn't* know (unless you took the long way home, and gave everybody time to find out on their own), and you have to tell him? Then, *that guy* runs into the streets and spreads the good news? Yes, build messaging into the game, and use it to set game flags that change game behavior. But don't overuse it, or abuse the system by allowing game states to change instantaneously that in ways that couldn't possibly have occurred. If the mayor of the town has his own wizard who saw everything happen through his crystal ball, that's a different story, and should be portrayed as such.

EXAMPLES

Classic games like *Wizardy*, the early *Ultimas, Phantasy Star, Might and Magic*, and the *Bard's Tale* had mostly statistic-based enemies, with very little special case code. They also usually used a puzzle system (using some sort of key or jewel or Skull of Muldark or what have you) that had to be found and used in the right place at the right time. This was most likely coded as a system of flags that the elements of the game would access to determine the particulars of the game progression.

Usually, the gameplay diagram for these games would include a town state, a travel state, and a combat state. These states divided up the core experience of most of these types of games, and the differences between these games were usually the overall game's graphic quality, how the player conversed with NPCs, and the combat interface.

Strangely enough, some massively multiplayer online RPGs (MMORPGs) of today are using this exact game style to create huge worlds for people to play in. The only real gameplay addition has, of course, been the vast number of people that are also playing the game at the same time, leading to more player-to-player interactions.

More modern RPGs such as the later *Final Fantasy* games, *Neverwinter Nights, Baldur's Gate*, and the *System Shock* are much more scripted affairs, with some of the attribute-based enemies of these games, but with a large portion of hand-tailored encounters and environments along the way to provide the player with a more crafted gameplay experience. Only recently have the online RPGs tried this

TeamLRN

tactic (such as the *Final Fantasy* online game) because of the enormous amount of work associated with creating custom quests and encounters for a world that may be inhabited by thousands of people at all hours of the day. But, the demand is there for higher quality content, so game companies will provide it.

EXCEPTIONS

Bethesda Softworks makes the excellent *Elder Scrolls* series of RPGs (see Figure 4.1 for a screenshot from *Elder Scrolls: Arena,* the first in the series), which it touts as being "open ended," meaning that you can solve the game and perform the various quests in a nonlinear fashion. The games do deliver this promise to a much larger degree than any other RPG. A large amount of freedom is granted by there being no (or very few) time limits on the quests you receive, so you can go around collecting quests, and do them in any order. The quests are still mostly scripted (a number of quest types are used as templates, with different characters and locations) and usually of a fairly simple nature to facilitate this (although the newer games in the series have vastly improved the variety and complexity of quests). The main quest is still linear, facilitated by scripted encounters with unique NPCs, but it allows the player to take time completing many other side quests as well.

FIGURE 4.1 *Elder Scrolls: Arena* screenshot. © 1993. Bethesda Softworks LLC, a ZeniMax Media company.

Neverwinter Nights is another recent game that was supposed to change everything. By allowing players to control a character in the game and actually be in the Dungeon Master role (as borrowed from the pen and paper world), the game was supposed to be *Dungeons and Dragons (D&D)* fully brought to the computer. To some degree it succeeded, but in many ways, all it really showed was that the average person is pretty bad at coming up with good game content. Patches have fixed some of the problems, and the title is nothing if not created for longevity, so this will surely change, and good modules will make their appearance on the Net.

SPECIFIC GAME ELEMENTS THAT NEED IMPROVEMENT

Role-playing Does Not Equal Combat

In most RPGs, right from the very start, most of the time spent "role-playing" is actually time spent *killing*, because of two major influences: really old pen-and-paper RPGs (such as *Dungeons and Dragons* and, earlier than that, *Chainmail*) centered on this aspect and because combat is much easier to model and invent than is an actual story with plot, characters, drama, and so on.

Consider this: nonkiller classes in most RPGs are only useful for the small set of contrived circumstances that the designers have included to justify these classes. Thieves are one of the more classic types with problems, even in paper *D&D*. If you allow thieves to *really* do what they do, they're too powerful (just like in real life, the Mafia is more powerful than a police officer, at least locally) because they don't have to follow the rules like everybody else does. So games hobble them. Thieves can disarm traps, and pickpocket. But, if they disarm wrong, they generally die, and if they pickpocket unsuccessfully, they are generally imprisoned forever. Fun is nowhere to be seen. Think also to the myriad "wonderful professions that you can choose from" in your average MMORPG game. In *Ultima Online*, you could be a baker if you wanted. Unfortunately, you could spend months playing the game, become a Master Baker, a true *King* of Baking, and then be almost instantly killed the second you stepped outside of town by an extremely low-level fighter with a rusty spoon. In today's MMORPGs, people tend to be tanks (meaning fighter types with huge amounts of health and armor; human walls that absorb damage), or casters (someone who stands behind a tank and can either damage creatures with spells, or heal the tank so he can continue to bash and be bashed). Specialty classes have somewhat dissolved into these two basic groups.

Huge areas of impelling gameplay are inherent within RPG worlds, but that involves thinking about ways of creating content that doesn't involve killing and that takes advantage of nonlethal skills in a meaningful way, not just to affect your prices

TeamLRN

for new swords. The task involved here is not an easy one, and writing AIs to support these new quest types will also be hard. But our RPGs will definitely be the better for it.

Grammar Machines (GMs)

GMs make better conversation engines. A lot of the interaction with other characters in RPGs is through conversation, usually in the form of choosing from a list of responses, and then reading the character's scripted response. *Ultima* used a keyword system, so you would say "thieves," and the other character would tell you about the local thieves, mentioning toward the end that someone named Blue runs them. A new keyword, "Blue," would show up in your list, and you could ask for additional information in this way. Old text-adventure games actually had rudimentary grammar engines that could handle semicomplex sentences. No fully functional grammar system that could be used to converse with NPCs in a modern RPG has yet been implemented. This might change because of the advent of better and better speech recognition software. Eventually, RPGs might use this system instead of a slow, clumsy text interface to allow the user to really ask questions. Our job as AI programmers will then be to fully flesh out a grammar engine, and fill a text database with enough knowledge to dutifully answer those questions.

Quest Generators

The real quest (for developers, that is) is quest generators that don't churn out derivative or repetitive content. Sort of the Holy Grail of large-scale RPGs, a quest generator could make up new quests that the player could tackle without having to be explicitly set up and scripted by a game designer. Games like *Everquest,* which are played around the clock online, could benefit greatly from a system that could come up with novel challenges for any number of party members, and of any skill level. As of now, only a few games have "random" quests, and they usually fall into the "fedex" quest realm. That is, go find this guy, get something from him, and bring it back to me. An improvement might be a system set up ad-lib style, using templates to create custom quests (or strings of connected quests) that included multiple characters, locations, rewards, and different actions to be done. These templates, connected to a database of potential ad-lib names and locations, as well as some way of scoring quests for skill level and such, could make RPG games truly unique experiences (at least for side quest interactions). The game could even keep track of which quests the player liked (by keeping records of quests turned down or never finished versus successful and repeated types) and adjust the kinds of quests given to a specific player. Also, by making the ad-lib machine extensible, you could add content (either to the online game, or through mods, patches, or expansion

packs to individual products) continually, and the ad-lib system would just incorporate it into the mix.

Better Party Member AI

Party AI that can be extended and modified, both implicitly and explicitly, is another big area in need of concern. Early real-time RPGs (like *Ultima 7*, pictured in Figure 4.2) had simple party AI that mainly just followed you around the map and tried to help during combat. *Baldur's Gate* has contributed heavily to real-time RPG party AI becoming a greater priority. The level of adjustment that can be accomplished within their simple script form is pretty astounding, but it could be better. The character could keep track of the sorts of actions you (as a player) have the character do, and could incorporate them into automatic behavior.

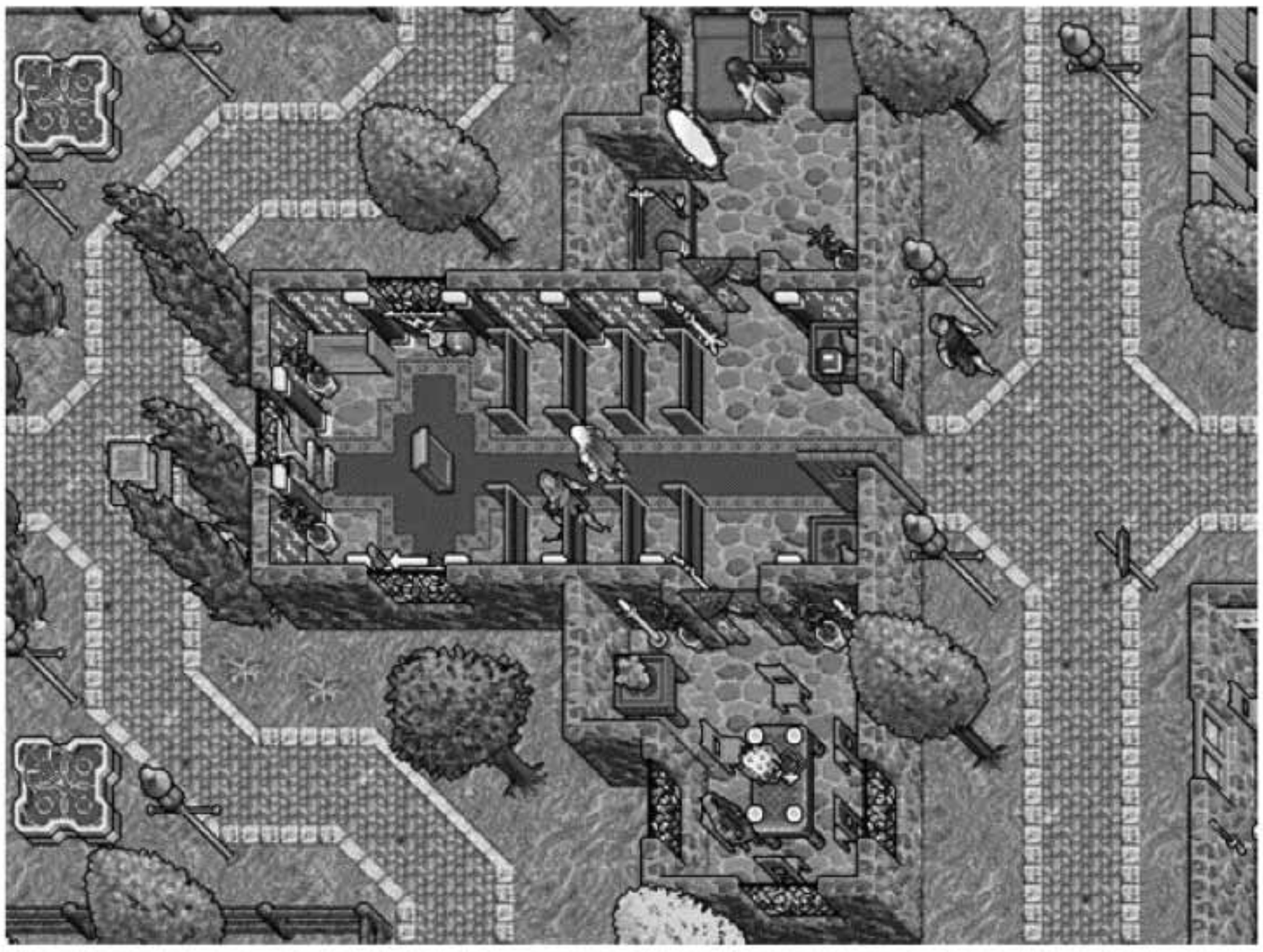


FIGURE 4.2 *Ultima™ 7* screenshot. Populous, SimCity, SimCity 2000 and Ultima 7 screenshots © 2004 Electronic Arts Inc. Populous, SimCity, SimCity 2000, SimAnt, SimEarth, SimFarm, Dungeon Keeper, The Sims and Ultima are trademarks or registered trademarks of Electronic Arts Inc. in the U.S. and/or other countries. All rights reserved.

TeamLRN

Think of this as simple learning by imitation. Do you always retreat a certain character (like a weak mage, perhaps)? After two or three times of doing this manually, the mage could retreat automatically. Do you drink a health potion whenever you get to one-third health, but only after the battle is over or after running away from immediate danger? The characters should perceive this and parrot these simple behaviors.

Imagine how the player's game experience is going to evolve and change as the game progresses, instead of micromanaging very tedious actions again and again during hours of gameplay. It might even be possible to show the player this learned behavior list and allow the player to edit it by deleting things, or changing the priorities of these behaviors.

Better Enemies

Instead of just mobs, groups of monsters that turn toward the player, advance until in range, and attack, enemies should work together from multiple fronts, using plans and the environment to their advantage. They should set ambushes, make traps, find your weakness and try to exploit it, and do everything else that a human player would do. This is, of course, a universal problem. As stated earlier, most RPG enemies are supposed to be relatively mindless, so the player can quickly kill enough of them to rise in rank at a rate that feels good. The problem is that this need creates very monotonous battles, one after another, with exceedingly stupid monsters. One popular answer to this is sub-bosses or mildly scripted and slightly more strenuous enemies that will make the player feel like the whole of creation is not filled with senseless drones, all attacking in the same manner as the last. *Dungeon Siege* (Figure 4.3) and the *Diablo* games used this technique relatively successfully, as areas of the map would always have a native type of creature, and some larger, stronger version of that creature type would be leading them. This unique creature would not be tied to any quest (although some were) but, rather, just provided a bit of variety to the constant stream of cannon fodder.

These sub-bosses could be taken further than just tougher versions of regular monsters, to a level where they are truly small boss monsters that rule that part of the game world. One of the great things about pen and paper RPGs was the concept of the Dungeon Master (DM), the human player in charge of setting up and running the game. Sub-bosses could be little DMs, giving better orders to their armies, and doing things that a leader would do. By killing this creature, you would weaken the attack of all the creatures he led, until another leader is found.

An aside about *Dungeon Siege,* however, is that the game did too many things automatically for the player, and the game could sometimes be played by itself, with hardly any input from the user. If this automatic behavior could have been modified or tweaked (maybe even just a slider so that the player could set the level of automation he liked), the game might have done better.

FIGURE 4.3 *Dungeon Siege* screenshot. © 2002 Gas Powered Games Corp. All rights reserved. Gas Powered Games and Dungeon Siege are the exclusive trademarks of Gas Powered Games Corp. Reprinted with permission.

Fully Realized Towns

The towns that constitute the trade and information centers of these games are usually pretty dull, filled with people either standing around, or moving between two locations. These townsfolk usually say the same thing over and over and don't appear to have a life at all. Obviously, this is not reality. By using simple rules, and a data-driven approach to town creation, even large towns could be populated with people who have jobs, go to school, shop for groceries, or whatever it is that people do in your RPG world. If you employ a system like this, you have to also make it somewhat easier for the human player to find people in the town (this is why most games have people standing in one place, so that the user knows where to find them). But this is a problem that can be solved (perhaps you have certain important NPCs that can be found in one of three different places, based on time of day), and the overall effect of a living, breathing town would make the game world much more interesting and immersive.

TeamLRN

Implementing this kind of town could be done a few different ways. You could use a *need-based* system, where each NPC would have a number of needs and would determine how to fulfill those needs. As an arbitrary example, let's say that a certain part of town contains 100 NPCs. Each NPC has three needs: hunger, business, and family. Each need is satisfied when the NPC performs tasks that are suited to the particular need (eating to hunger; trading, training, talking, and so forth to business; and parenting, providing, and so on to family). The game could then use a "need pathfinding" system to give information on how to fulfill its needs to each NPC. The streets would be busy with people, going to and fro, buying bread, painting fences, or looking for their kids. The given action of each townsperson is defined by what need is the highest. Another way to write this system would be to write a number of different scripts, each of which would define a chain of actions, and just assign these little scripts to each NPC in the map. The second method saves a lot of computation (because you don't have to do any sort of planning, or need tracking), but isn't as general (you could implement a hundred different places for a need-based NPC to satisfy his hunger and the AI would use them all, you'd need to write a hundred different scripts in addition to creating the hundred different places in the scripted system).

SUMMARY

As a game genre, RPGs have been here a long time and show no sign of going anywhere. They provide people with an escape from their ordinary lives, by taking on the persona of another person or creature. The AI systems in this genre are quite complex, with many different AI needs across the entire game.

- Enemies and Boss Enemies are necessary to give the player something to fight against, and to provide story motivation.
- NPCs and Shopkeepers provide the player with more personal interactions (other than combat), and give the world a feeling of being alive, complete with an economy.
- Party member AI needs special attention, especially in real-time combat-based RPGs.
- AI Scripting is a prime weapon to use in developing RPGs, but FSMs, and messaging systems are also staples for this genre.
- Some areas in which RPGs need improvement include grammar machines for better conversations, quest generators for more varied and long lasting gameplay situations, the always-present push for better enemy and party member AI, and fully realized towns to give the player a greater sense of immersion in the world.

5 Adventure Games

Adventure games and early personal computers were made for each other. In the late 1970s and early 1980s, adventure games were some of the first games to make entertainment use of the clunky PCs that were just starting to become popular.

The so-called *text-based* adventure games (the original being *Dungeon,* which eventually became the classic *Zork* series) were our first taste of the genre and got their names because they had no graphics whatsoever—a text description of the room you were in and your imagination were all that you had to go by. The player would type commands into a parser, and the game would either respond in kind with the result of the action the user had entered, or inform the user that it didn't know what he was talking about (if he typed something in that wasn't in the games command language). The player traveled from room to room, collecting elements used to unlock puzzles, which would in turn allow him access to other areas and further the story.

Eventually, people started attaching pictures to these puzzle-filled stories, including games like the *King's Quest* series, LucasArts'® seminal *Day of the Tentacle* and *Monkey Island* games, and the *Leisure Suit Larry* games. LucasArts also did away with full text parsers, instead relying on a highly simplified keyword and iconic interface.

In 1993, a little company called Cyan released a game called *Myst. Myst* took the adventure game and removed almost the entire story, leaving a very pretty world (it was one of the first CD-ROM games and used prerendered backdrops that looked amazing compared with the simplistic real-time 3D worlds that people were used to seeing in other games at the time) and a large number of puzzles to solve. You couldn't die, but there was also no help to guide you through the game; it was pure exploration mixed with trial and error. Although this sounds like a simple premise, *Myst* was *the* runaway hit of its time and is still widely credited as the best-selling computer game of all time, with more than 12 million copies sold. It spawned five sequels and countless similar games trying to follow its formula.

TeamLRN

Today, the *classic* adventure game has all but disappeared. Nobody seems to know why. The *Myst* games may have given the genre sales (adventure games had never been very big sellers), but they also may have been the reason for the current dearth of new titles. People started to associate the adventure game title with slow, casual gaming that was merely a collection of puzzles and forgot (or had never heard) about the well-written, rich storylines of the earlier titles. They have instead headed for the instant gratification of the more action-oriented adventure game variants that have begun to take over the genre today.

This book will not concentrate on the classic style of adventure game, which has also been called *interactive fiction*. The level of AI elements inherent in these games is pretty low. They are usually coded with state-based characters; most have only static elements, and only certain games even have actors that can move from room to room. Also, because the human could solve the puzzles in many of these games in any order, the AI for the characters is something more akin to a database of flags then to actual code.

Instead, this book will focus on the modern alternatives that have all but taken over the genre. These new takes on the adventure game are usually variations of the first-person shooters/third-person shooters (FTPS) genre that focuses on noncombat-based gameplay situations: a mostly exploratory game (like *Tomb Raider*), or the more recent *stealth* games. The stealth game involves a main hero who cannot shoot his way out of the primary situations in the game but must, instead, use elements of stealth and guile to slip past the guards (such as the recent *Metal Gear* games, or the *Thief* series). Stealth games have proven hugely popular because of the varying gameplay elements, and the heightened sense of tension that comes from having to come up with alternative means of traversing the level and solving problems other than "pull the trigger." This transcends the FTPS roots of the games, bringing us back to the feeling of constant puzzle solving and a great storyline, but in a real-time game environment, so these are now considered adventure titles.

Another variation, which does contain some combat elements, is called the survival horror game. Titles such as *Resident Evil* still have a lot of combat, mostly projectile attacks, but these are mostly 3D exploration titles with lots of puzzle elements to drive the player around the map.

COMMON AI ELEMENTS

Enemy AI

For the most part, enemies in stealth games tend to be implemented with scripted movement sequences or very simple rules. The player needs to sneak by guards and

other enemies and, thus, has to be able to identify patterns of movement to determine ways of exploiting these patterns. Once alerted to the player's presence, however, the enemy's behavior can get a whole lot smarter, and enemies can become quite involved. Guard characters usually employ multiple stages of attention (from "Did I hear something?" to *pretending* he didn't hear you as he slowly patrols in your direction while taking the safety off his gun) and perform basic gameplay behaviors like calling for backup, hunting you down, and so forth. One limitation on enemy behavior for this genre to remember is that you don't want the enemies to be too diligent, or you'll wake up the whole complex by setting off one guard, which would be frustrating to the human player.

For other types of adventure games, pretty much anything goes. Some games use somewhat mindless hunter-style enemies, as in the simpler FTPS games. Other games have smart enemies that are constrained to zones (as in the *Thief* games), so you might find yourself being tracked down by an alerted guard, but he won't set off the whole world if you can escape his territory within a reasonable time.

The survivor horror titles use *very* simple enemy AI, usually because the monsters involved are zombies, or the like. The combat interface is mostly secondary to the exploration and puzzle interaction, so the enemies are a bit slower, and the action isn't so *twitch* oriented (reliant on fast reflexes).

Nonplayer Characters (NPCs)

Just like RPGs, NPC characters are noncombatant inhabitants of the game world. They are placed there to give the player information, or to bring the world to life for visual support. The AI used in these characters is quite varied, from both an ability level and an implementation level, and can be anything from a static dialogue and actions to a much more complex system involving paths, goals, and a conversation engine to engage the player with. This is all determined by the design of your game.

Cooperative Elements

Cooperative elements characters go beyond the realm of NPCs. In RPG games, they would be members of your party. These people assist you directly, by showing you places or helping you fight against the creatures in the game, or are even secondary main characters. The last group is for games in which the gameplay involves the player constantly switching primary control back and forth, in episodic or mission-based chunks of time, between different game characters. Switching control like this is a great way to decrease the perceived linearity of your game and to break the action into manageable chunks for the player.

Because of the touchy nature of stealth games, the programmer must make sure that an AI helper in that genre isn't going to do anything really stupid that would set off the guards, or else we're back to frustration.

TeamLRN

Perception Systems

For stealth games, most of the complexity of the AI model is contained within the perception system. Different techniques have been developed for each of the senses—to model the sense but model such that it translates well to the video game world.

Thief, from LookingGlass™ Studios, took the stealth game to an entirely new level, with the main thrust of the gameplay being constant sneaking, hiding in shadow, pick pocketing specific characters when they're not looking, and so on. A good breakdown of the perception system of *Thief* was given by one of the programmers who worked on the game at the 2002 Game Developer's Conference; the paper can be found online at the following site: *http://www.gamasutra.com/gdc2003/features/20030307/leonard_01.htm* under the heading *Building an AI Sensory System.* This is highly suggested reading if you plan to do a system of this complexity. Also, see the CD-ROM for additional links and materials.

Camera

Most adventure games are 3D (a notable exception is the two-dimensional [2D] *Commandos* series) and third person, so we again see the problems associated with bad camera placement. However, because of the much slower pace of these types of games, this is usually a much easier problem to fix, and cinematic-style camera cuts with precise camera placement are usually the norm. Certain sections of the game may require a free-form camera system, however, and thus, need programmer attention. The stealth games also frequently require an *around the corner* camera angle for hiding behind cover and watching a guard walk by. This can be an algorithmic camera that comes up when you crouch next to a corner, or specific camera parameters can be set up in the level editor for particular cover positions.

USEFUL AI TECHNIQUES

Finite-State Machines (FSMs)

Many elements of stealth and exploration adventure games lend themselves well to FSM-based AI systems. If the game is digitally triggered, such as guards having an alerted state of *yes* or *no,* or has an enumeration of states (like neutral, annoyed, alert, mad, berserk), then state machines provide the best bang for the buck. Because of the nature of state machines, you can make parts of your AI fairly simple, but other parts could have many more states and much more complexity. Thus, for games with only a few more complicated AI tasks and a large amount of very straightforward AI tasks, you might want to stay with this system.

Scripting Systems

Some adventure games use very cinematic camera placement, lots of in-game dialogue, and sequences that show the results of solving a particular puzzle somewhere else in the level. Scripting systems allow the programmers (and designers) to put extra tailoring into specific parts of the game, and this technique is readily used for the linear story that these games employ. The combination of triggered events setting off scripted sequences, and having the trustworthy game mechanic of having to "unlock" later parts of the game by accomplishing tasks (thus setting certain game state flags) gives the best of both worlds. It allows game designers to have many places within a game in which to get specific things to happen, while still giving the player some feeling of being able to roam around uncontrolled.

Messaging Systems

Given the event-driven nature of puzzles (push lever A, door goes up; move three stones into certain pattern, hidden chamber lights up; and so forth), messaging makes a lot of sense in these games, so that the disparate elements in the game don't need access to each other to communicate (although the flag coordination could be done completely within the perception routines). The advanced perception systems of stealth games can use messages for determining perceived sounds and the like, as well as providing enemy guards an easy method for alerting others or calling for help.

Fuzzy Logic

Given the complex nature of many stealth games' perceptions, AI opponents need fuzzy decision making when dealing with sensory data. This is to make guard states feel forgiving to the player (you can sneak by if you don't push the boundaries *too* much—like being able to push on a pinball table: some movement is legal, but if you overdo it, you tilt). Frequently, part of the gameplay is having the guards deal with situations such as player-initiated distractions, diversions, ambushes, and other kinds of slighting to the enemy. These sorts of interactions are scripted quite often, but could also be coded by using fuzzy logic to allow the guards a model of the world, to deal with the kind of imperfect information that a diversion might provide. So, a guard's model of his territory might be fairly clear—he hasn't seen or heard anything suspicious in a while. Then, you throw a rock into a dark corner. He hears it, his suspicion level goes up a bit, he adds a suspicion target to his internal list, and he focuses most of his attention on it because it's his only area of concern right now. You toss another rock; he reacts by getting more suspicious. He yells, "Who's there?" and cocks his weapon, moving slowly toward the corner. You get the picture. The ebb and flow of suspicion, directed toward however many targets,

TeamLRN

is determined by his *very* unclear, sparse picture of the world, which is determined by his perceptions. Of course, this kind of system would be much harder for the player to figure out; scripted systems are usually quite telegraphed, in that you watch the guard for a bit, and notice that every two minutes, he gets up and goes to the balcony to look outside, giving you a window of time to make your move. In practice, most of this fuzziness would better used within the perception system itself, rather than in the decision structure. An FSM with fuzzy transition logic is much easier to program then a full fuzzy logic system is.

EXAMPLES

After the classic adventure games started to slow down in popularity, the crossover genres started to appear. *Tomb Raider* was the early hit that started us off on the crossover from shooter to adventure game. Other earlier games included *Alone in the Dark,* and *Shadow Man,* which eventually gave us *Resident Evil,* which spawned a slew of horror-based exploration titles like *Silent Hill,* American McGee's *Alice in Wonderland,* and *Nightmare Creatures.* Note that these action/adventure games still had lots of combat involved. This was because the AI systems were still borrowing heavily from their FTPS brothers, and the designers just increased the exploration and item-gathering challenges to encompass more of the overall experience.

As the AI got better, and perception systems became complex and had gameplay depth, the stealth games came out, with *Thief, Deus Ex,* and *Metal Gear Solid* initially leading the pack. These games made it fun to *not* kill your enemies but, rather, to never even let them see you. *Commandos* was an overhead 2D stealth game: your job was to accomplish missions by infiltrating increasingly complex enemy bases and sneaking from spot to spot unseen. The game was spectacularly hard, but very well done. The line of sight of all the guards was actually shown as moving cones on the ground, so you could much more intimately time your movements to ensure your secrecy.

Another notable hybrid adventure game was *Blade Runner,* which touted real multiple endings and storylines, and a somewhat alive world (meaning that the NPCs in the game were engaged in semi-autonomous behavior, moving through the city to get to stores, jobs, and so forth, but the overall effect was mostly cosmetic because interacting with the NPCs was still very state- and event-based).

Although the classic adventure game is rare, it is not fully extinct. Some great examples of these games in recent years include *Full Throttle, Grim Fandango,* and *Circle of Blood.* These games have expanded the old formula, with better (and more involved) puzzles, great graphics, and much more varied gameplay elements (*Full Throttle* even included a motorcycle combat stage). The interaction system that these games use has gone up and down in complexity over the years. With the ini-

tial text adventures, the player could type pretty much anything, and the game's parser would either recognize the command or say otherwise. Players would eventually learn the commands that the parser knew. Later, with LucasArts' SCUMM system (which stands for *Script Creation Utility for Maniac Mansion*, a great example of a tool being built for a specific game becoming the cornerstone of an entire suite of games, as the SCUMM engine eventually was used in no less then 18 games), the possible commands were given to the player as buttons on the graphical interface, and the player could apply these commands to various elements on screen. *Full Throttle* went even more abstract, with icons depicting your eye, mouth, or hand being used as context-sensitive commands to apply to game objects. So, if you used your mouth with an NPC, you'd talk to him, whereas if you used it with a beer, he would drink it. Because of this simplification of input from the human to facilitate ease of interfacing with the game, the NPCs in the games have in some ways become much more simplistic as well. They can't really communicate with the player beyond a certain level, simply because the player no longer has any means by which to respond intelligently. If an NPC asks you for the time, do you click on him with your mouth icon or with the hand icon? If you choose the wrong response, and the NPC asks what's wrong with you, then what?

AREAS THAT NEED IMPROVEMENT

Additional Types of Stealth Goals

In addition to the classic stealth mechanic of patterned movement that has to be circumvented, *Deus Ex* gave players many different ways to accomplish key story goals. For example, to get through a particular door, you could shoot the guard and take his key, and then have to fight the other four guards that would come when they heard the shot. You could also cause some kind of diversion, and then use your hacking skill to open the unguarded lock. Or, you could climb through a ventilation shaft and find a different way in. You could even find a guard uniform and use it to walk right by the guard. By doing this, the game designers made each encounter and area of the world into a puzzle. You had to really experiment with the situation to uncover the hidden gameplay gems. The player didn't have to sneak down one particular hallway and open one particular door. This forced *Deus Ex*'s guard AI to be more open ended, instead of being heavily scripted because there were potentially so many ways to get around them.

A Return to Traditional Adventure Roots

Traditional interactive fiction provided computer gamers with some of the most popular games in the early days. Many of the classic LucasArts and Sierra games

TeamLRN

have loyal followings, even today. Today's exploratory and more action-oriented games must meld with classic roots of the genre to bring adventure games alive again.

Better NPC Communication

The inherent noncombat nature of modern adventure games lends itself well to having additional story-driven elements included as part of the experience. By giving NPCs in adventure games real grammar systems, or even allowing branching storylines within the full umbrella of the greater game story, the world in which the adventure is occurring could become more real, and much more personal to the player. This, of course, would require an immense amount of additional work in story design to make up for branching and consistency problems.

User Interface

When we lost the full-text parsers of the original text adventures, we also lost the ability to have very rich interactions with in-game characters. After going to a graphical interface, the complexity was gradually degraded until eventually some traditional adventure games had as few as three or four basic commands that could be used with elements in the world. Nowadays, with the more action-oriented variants, there is barely any interaction other then positioning yourself well and using quiet weapons or tools when necessary.

Imagine *Sam and Max* with a full voice interface, or some other kind of general interface where you could get a much richer kind of connection to the game if you spent the time to explore the capabilities of the interface. Eventually, a new interface might help adventure games regain some of their traditional depth, without having to resort to typing long sentences into a computer.

SUMMARY

Adventure games are continuing to evolve from their initial roots, which was more a string of puzzles wrapped into a story, and were definitely not played in real time. The modern stealth games and the more action-oriented exploration games are modern variants of the classic adventure formula that will continue to give game players challenges and new worlds to explore.

- The first adventure games were text based and required the user to type commands to a parser. These eventually gave way to the graphical adventure game, which was the same type of game, but with a graphical user interface.

- Modern adventure games are are variants on the FTPS genre and emphasize noncombat situations such as exploration and stealth.
- Enemy AI in stealth games can be somewhat pattern based because the object of the game is to note patterns and circumvent confrontations. In the more exploratory games, enemy AI can be much more varied.
- Most adventure games have a number of NPCs, as well as cooperative characters, that give the player information or new gear. The AI level of these agents varies greatly.
- Perception systems are paramount for stealth games because overcoming the guards' perceptions is the goal of the game.
- Camera AI is usually necessary for these adventure games because they usually are done in 3D.
- FSMs, scripting, fuzzy logic, and messaging AI systems are very commonly used with the adventure genre.
- New stealth challenges (possibly by infusing the current game schemes with more intelligent enemies) is an area of improvement for this genre.
- A return to the classic adventure game roots is needed to help revive the lineage of the genre.
- Increased NPC communication and story branching might give adventure games additional personal connections to the player.
- An advanced user interface could help give back the richer interaction level of more traditional adventures to modern games.

TeamLRN

6 Real-Time Strategy (RTS) Games

The AI systems used in RTS games are some of the most computationally intensive of games, simply because they usually involve numerous units that must coordinate and technology trees that must be navigated to perform goals. They must also share CPU time with collision detection and drawing routines, which also have numerous units to contend with. Although RTS games have been around for years (the 1990 game *Herzog Zwei* for the Sega® Genesis™ console is usually considered the first), AI in these games has been nowhere near the level of good human players. This is because the AI in RTS games has to contend with huge numbers of objects, are supplied typically with incomplete information (like the fog of war), are heavily concerned with micro actions, and have to run in real time. By contrast, most games in which the AI is considered expert (or just very good) mostly exist for turn-based games, with perfect information, in which most moves have global consequences and human planning abilities therefore can be outsmarted by mere enumeration. This type of game includes chess and the like. Thus, almost every aspect of RTS games is considered nonoptimal for AI performance. The burden lies on us to overcome these problems.

COMMON AI ELEMENTS

Individual Units

The real *player* in RTS games is the overseeing General of the game, either the CPU or the human user. The goals each player is fighting for involve the entirety of thier society. However, this doesn't mean that individual units are worry free. Individual behaviors in RTS games are usually considered secondary, by temporarily overriding the primary order given by a user. Most of this local intelligence falls into the categories of pathfinding and obstacle avoidance, concentrating attacks, and falling back when you cannot win. The question of how much intelligence to put at this tactical level is tricky. The amount of micromanagement your RTS is trying to

TeamLRN

achieve will likely determine this. The more individual intelligence a unit has, the less you have to check every unit in your army. However, for games with low individual AI, if the CPU opponent micromanages its individual-unit AI too much (giving it the appearance of better individual AI), it will be seen as cheap because it isn't possible for the human to replicate the computer's efforts as fast or easily. One easy example of this is the archer behavior in the *Age of Empires* games. The computer will send in many weak units, which will shoot, retreat, shoot, retreat. This very simple micromanagement of behavior leads to making these weak units become much more powerful because they will string out and separate guards in all directions, a behavior that would be very difficult (or at least tedious) for a human to do. Reliance on this simple individual behavior has also made the *Age of Empires* games not attempt more common strategic techniques, such as setting up a wall of melee fighters and putting the archers (or other long-range attacker) behind them for support, which is something that almost all human players do.

Economic Individual Units

Sometimes called *peons* (the "builders" and "gatherers"), economic individual units are units that usually do not fight but are, instead, employed as the economy on which the player gains resources for creating his armies. Much like other individual units, the level of AI has to be carefully tuned to the level of micromanagement the game is shooting for. *Age of Empires* recently addressed common dislikes about this area of the game's AI by making peons automatically start gathering resources after building a resource-associated building, and also making food gathering easier by the ability to "queue up" farms instead of having to check back and replant them manually. Other common techniques include:

- *Order queues.* In most RTS games, the interface allows you to tell a unit to perform multiple actions, one after another. This is a very powerful addition to the genre because it allows smart players to plan the behavior of their economic units ahead, so the player can then rest assured that their economic units will be busy during more battle-oriented points of the game. However, the interface still requires the player to set it up, so the AI of each individual unit doesn't have to be bloated with special case code designed to make the peons appear smart.
- *Auto-retreating.* Peon units can rarely fight (or aren't any good at fighting), so most RTS games have some sort of autoretreat AI for these units. Usually it's just leaving the attack range of the enemy, however. It could definitely be improved by getting to a building for protection, or running to the nearest military unit (while shouting "Help!"). Also, noticing when the danger is over and going back to work would be another welcome addition.

Commanders and Medium-level Strategic Elements

Some games directly use "commanders" to bolster groups of units (such as *Total Annihilation*, which used its commander unit as a primary builder in addition to a super unit), or commanders are used internally by the AI system to group units into fighting elements and control them in a larger war sense. This is a medium level of AI, in that it requires much more than simple individual actions (such as shoot or go somewhere) and is not full high-level strategy (take command of a particular resource, or defend the base) but, rather, is right in between. A simple example is a commander choosing a new destination for a group of units, but the individual units decide how to stay in formation and use the terrain features to get there. A more complex example would be having a high-level directive to attack player #3, and the commander level would then direct 20 infantry to attack from the west, followed by a group of ranged weapon units and send some tanks in from the south, taking out some towers that could harm the infantry along the way. This level of RTS game AI is usually sorely lacking, by and large because it is the most involved. It deals with coordinating strategic combat elements (usually more than 5 units, possibly as many as 30, but anything bigger could be considered a separate army) toward greater efficiency and performance by directly setting destinations, actions, and enemy targets that correspond to local and battle-level goals.

High-level Strategic AI

Think of this as the general of a real army. Performing commands and plans from this level of direction, might involve numerous units, or require whole sections of the economy to shift, and include actions at many different levels of AI to complete. The perceptions at this level are usually built on information feedback from the lower levels to determine what its enemies are up to. Given all this feedback, the high-level AI would make plans to deal with threats exposed in the perception data. In this way, the Strategic level affects everything from the individual soldier (as part of a larger group of soldiers who are told by a commander level to respond by moving) to the entire economic system for the AI player (when shifting the allocation of units that are retrieving resources to bias a particular type that will support the high-level plans).

Quite frequently, the high-level AI is multifaceted, in that it is running resource allocation between several different aspects of the game (defense versus offense versus research versus economy), and thus represents most of a given RTS civilization's *personality*. Race #1 might value offense and have a strong economy. Race #2 might be cautious and studious. Coupled with specialty units for a given AI type, and some tunable parameters, you can differentiate different types of AI opponent races easily, just from this level of the AI.

TeamLRN

Town Building

Setting up the initial headquarters, as well as advance bases for the AI, is a difficult problem in its own right. You want to place structures somewhat close together, for ease of protection. But you also want to spread them out a bit, to get better visibility and guard against area effect weapons. Finding this balance, while keeping a fluid economy running, is quite challenging. Many games use hard rules for town building (which are broken up into difficulty levels) that start out fine, but may or may not be able to cope with changing world conditions and as such look pretty silly by the end of the game. The decisions about where to place key structures need to account for many different elements. Economic structures need to be placed next to the resource they are going to store; military structures need clear exit lanes and proximity to the front line (if possible). Guard structures need to maximize visibility effects and be able to back each other up and guard the largest possible number of other units.

Indigenous Life

Most RTS games also include native elements to their game worlds. Games like *Warcraft* have sheep walking around in them, and *Age of Empires* actually uses native fauna as a resource that can be gathered. Other games treat the locals as a hazard, or even a source of powerups and such. AI for these entities is usually minimal, but some games give them a certain degree of "smarts." Depending on the nature these elements will play in your game (be it resource or hazard), you need to balance the distribution of these elements, as well as the randomness involved, otherwise your players will not have fun. *Age of Empires* games using random maps can sometimes be thrown off by having a wolf too close to a player's initial town, and this random element can diminish the starting capabilities of that player tremendously if the wolf inadvertently kills one or more of that player's starting peons.

Pathfinding

Pathfinding is one of the biggest CPU concerns for RTS games. Because there are a huge number of units, which conceivably could be ordered to go to different locations about the map, that pathfinding system for a game must correctly find quality paths, load balance the CPU cycles necessary to find these paths, and use other optimizations to make pathfinding feasible for so many separate entities. Things like formations, flocking techniques, and follow-the-leader-type systems will vastly improve the speed of per-unit pathfinding. Other pathfinding concerns include friendly units blocking paths, special case choke points like bridges, and dynamic path elements such as user-constructed walls or level debris.

Tactical and Strategic Support Systems

Many RTS games are increasingly using extended AI techniques to make the actions taken by their games smarter. These advanced support systems include the following:

- *Terrain analysis.* By dividing the terrain into manageable chunks and breaking down various aspects of each chunk, the AI can glean huge amounts of data that can be useful for strategic decision making. Terrain bottlenecks and odd landscape features can be identified and recorded for the pathfinding system, so that the pathfinder can more easily (and quickly) develop with intelligent paths. The AI can keep track of enemy base locations and resources and find holes in his (and the other player's) defenses. Most of this is done by using an influence map, which is really just a fancy name for grid-based map attributes, or data specifically describing certain aspects of each grid-sized piece of the map. This data can be updated as additional scouting information comes in, but might prove costly, so make sure your budget allows this level of recalculation. Some RTS games have a special multiplayer mode in which most of a certain resource is centered on the map, leading to a vicious fight over this precious resource by all the players. Human players can see that this is the only way to win in this style of map, but AI opponents, unless specifically analyzing the terrain for features like this, are quite bad at seeing the long-term problem with this map. They will only head for the far-off resources when local ones are depleted and will usually be overrun by human players who have already taken control.
- *Opponent modeling.* In games with imperfect information, like RTS games (or poker, for another example), you cannot use standard AI opponent assumptions, such as "My opponent will make roughly the same decisions as I do, because we both use the same optimal search algorithms for the state space of this game." The reason is that at any given time, the AI might not know the abilities of the other players, and thus has no basis on which to make predictions about his opponents. By observing and noting both physical abilities of the opponents (like seeing a Dread Mage, or hearing a dragon scream), as well as opponent behaviors (the opponent has always attacked my base from the right, or has always built a tower near his gold mines), the AI can build a model of his opponents. It is important to keep this model as up-to-date as possible, so the AI can use this model to make much more appropriate decisions in dealing with his opponents. By noting which players have specialty units in their army, the AI can build a fairly accurate tech tree for his opponents and know what other technologies or units each opponent has access to, and can plan for future attacks that might use these. Recording player behavioral tendencies (which types of units the player likes, the time between player attacks, the usual

TeamLRN

kinds of defenses the player uses, etc.), the AI can better assign defenses and build the correct units to answer upcoming challenges from his opponents. In essence, this is what human military generals do, as well as the meaning of the age-old saying, "know your enemy."

- *Resource Management.* Most RTS games (*Myth* is a notable exception) have an economy that must be tended as much, if not more, than the battles. Raw resources such as gold or wood and secondary resources such as units and structures must be considered when doing resource management. Most games' AI handle this complex task by starting the AI off with a build order (a string of things to build, one after another, that will jump-start a thriving economy), which is a technique that even human players use. This leads to very predictable AI behavior, however, because experienced human players are quick to discover this build order and, from it, learn the approximate times for attacks and when AI defenses will come online so they can exploit defensive holes. Better to involve resource allocation systems that recognize resource deficiencies and rectify them by using the planning system to organize goals necessary to fill these needs. By using a need-based system, AI opponents could be implemented that bias heavily toward certain units or resources and would rely much more on map type and personality, rather than blindly following a build order and then reacting to the outcome of the initial first large battle. Even humans who use a build order are quick to adapt the build order to specific things that they see (either in the form of map resources or enemy activity, through their scouts) so that they are not caught blind.
- *Reconnaissance.* Most of these games have some form of "fog of war," which is a mechanism for visually representing two things: unexplored terrain and line of sight. To combat these perception deficiencies, players must use units to explore the map, to uncover map features such as borders or resources, and to find the enemy and his forces. This is a very tough assignment indeed. Most AI opponents in RTS games do a good job of exploring the map, simply because they can micromanage a scout unit much more effectively than most humans, but the concept of keeping tabs on enemy movements and encampments through additional recon is not very common. Humans have to use this to see what kinds of threats the AI (or other human players) are building up against them, as well as noticing any changes to the area that have occurred since the last time a scout went through, such as the creation of guarding structures, or the depletion of resources by other players. One way that some games have tackled this problem is to have the AI-controlled player use a scattered methodology when building his structures. The AI player doesn't have to remember where anything is, so it can create very random and scattered towns that give the AI system the greatest amount of line of sight possible. Then, advancing

armies from other players are sure to enter the line of sight of one of these forward buildings, thus alerting the system to invasion early on. This does lead to somewhat greater building loss by the AI, though, because the human will make sure that these forward buildings are taken down as they are passed. A better system would be the much more complex wall building and guard-post placement that most humans use.

- *Diplomacy systems.* One of the underused places for AI in today's RTS games is that of diplomacy, which is defined as different players working together toward a victory. *Age of Empires* takes diplomacy to mean "we won't kill each other," and you also share map visibility information, but doesn't go into such areas as supporting your ally's troop movements, or specializing ("you crank out units, I'll mine gold and build towers"), or even simply timing attacks better to coincide more readily with your allies. Human players manage all these diplomatic tasks very well, and AI systems should step up. Of course, this involves additional AI work and additional user interface work because the human would need ways to communicate to the AI ally that he's planning an attack from the south in 15 minutes, or that he needs help in sector six.

USEFUL AI TECHNIQUES

Messaging

With such a huge number of potential units in the game, polling for game state changes or enemy events would be computationally wasteful. Instead, messaging systems can be used for broadcasting of events and game flags to a large number of registered units quickly and easily.

Finite-State Machines (FSMs)

Never to be left out, FSMs can be useful somewhere within the various AI tasks that are part of the RTS world. Individual-unit AI (most likely implemented as stack-based FSMs, so that they can be temporarily interrupted, then restored easily), systems within the strategy level (a city building might be a basic FSM with an offline constructed build order that has been proven to work), and many other game elements can take advantage of the loyal FSM.

Fuzzy-State Machines (FuSM)

RTS games' higher-level strategic requirements are some of the few game genre problems that don't lend themselves well to regular state-machine-based solutions. The preponderance of imperfect information about the opponents and the world,

TeamLRN

combined with the number of micro decisions that need to be made, make for a game in which the AI opponent usually has multiple directions to play toward, all of which are winning decisions. A better system is fuzzy-state machines (FuSM), which provide the structure and reproducibility of state machines, while accounting for the somewhat flying blind nature of RTS decision making. The AI might not know how many tanks the enemy has, or how much gold the opponent has in reserve to purchase additional reserve troops, but must still try to thrust forward toward victory. FuSMs allow this type of gameplay decision, without using the more straightforward way of just cheating and giving the AI knowledge of his opponent's positions and army makeup and making "intelligent" decisions based on some randomness and the difficulty level of the game. An AI system does this by using the parallel nature of FuSMs to determine, separately, how much effort to spend on each facet of command that might require attention at any given time. Thus, the complete *blend* of behavior that the AI is exhibiting is going to be much more varied and contextual, and will not use omniscient cheating to help the AI.

Hierarchical AI

RTS games have multiple, sometimes conflicting AI requirements, such as needing to move an army from point A to point B, but along the way, a small ambush happens and army members are being attacked. Do the units attacked break off and return fire, does the entire army stop and make sure the problem is fixed, or does everybody ignore the threat and march on? The answer is determined by the amount of individual versus commander (or strategic versus tactical) AI, but also the interface between these differing layers and how one can influence the other. Hierarchical systems provide a means for RTS games to form high-level goals but also appear smart at a unit level, without choking the primary AI system for resources.

Planning

Goal planning is a large part of the RTS AI world. To accomplish higher-level tasks (for example, to guard the left side of my camp against air attack) any prerequisite tasks must also be added to the AI's current plan. Thus, for the just-mentioned task, the AI would have to also (1) gain any foundation technologies in the tech tree (for example, you might need to be able to make guard towers before you can make antiaircraft towers, or you could require a communications building so that your weapons could use radar to detect incoming planes), and (2) determine the necessary resource units to spend (which, if deficient, might spawn a secondary goal to gain more of the needed resources). Tech tree navigation is only one area of planning, however. Specific offensive or defensive goals require planning to appear intelligent as well. It has even been researched that to look truly intelligent, even

simple tasks like running away from a threat need some level of forward thinking (beyond just pathfinding).

Scripting

Although usually not used to the same extent as other genres, scripting is used to extend the story elements to certain games, or to more rigidly describe the behavior of certain units under certain conditions. Certain games (especially RTS games that have recently made the jump to 3D) seem to be concentrating on fewer units and more scripted and rich interactions between these units (such as *Warcraft III*). This emphasis on superunits has led to more scripting being used in this style of game, in much the same way that *Half-Life* led to more scripting in FPS games.

Data-Driven AI

Many of the larger RTS games are putting large portions of the AI decision making into noncode form, be it simplistic parameter setting (like the early *Command and Conquer* games) to actual rule definitions (such as the *Age of Empires* scripts). This allows two things: designers working on the games gain easier access to the game so they can tune the AI, and people who buy the game can tweak the AI settings themselves. *Age of Empires* especially needed a system like this, with *Age of Empires* having 12 civilizations, and *Age of Empires* upping that to 13. See Listing 6.1 for an example of a user-defined *Age of Empires* script.

LISTING 6.1 A Sample *Age of Empires* AI User Defined Script Showing Simple Rule Definitions

```
; attack
(defrule
    (or (goal GOAL-PROTECT-KNIGHT 1)
        (goal GOAL-START-THE-IMPERIAL-ARMY 1))
    (or (unit-type-count-total knight-line >= 25)
        (soldier-count >= 30))
=>
    (set-goal GOAL-FAST-ATTACK 1)
    (set-strategic-number sn-minimum-attack-group-size 8)
    (set-strategic-number sn-maximum-attack-group-size 30)
    (set-strategic-number sn-percent-attack-soldiers 100)
    (attack-now)
    (disable-timer TIMER-ATTACK)
    (enable-timer TIMER-ATTACK 30)
    (set-strategic-number sn-number-defend-groups 0)
    (disable-self)
)
```

TeamLRN

```
(defrule
    (current-age == feudal-age)
    (soldier-count > 30 )
    (goal GOAL-FAST-ATTACK 1)
=>
    (set-strategic-number sn-number-explore-groups 1)
    (set-strategic-number sn-percent-attack-soldiers 100)
    (attack-now)
    (set-goal GOAL-FIRST-RUCH 0)
    (disable-timer TIMER-ATTACK)
    (enable-timer TIMER-ATTACK 30)
    (disable-self)
)
(defrule
    (current-age == feudal-age)
    (soldier-count > 20 )
    (or    (players-current-age any-enemy >= castle-age)
        (players-population  any-enemy >= 20))
=>
    (set-goal GOAL-FAST-ATTACK 0)
)

(defrule
    (current-age >= feudal-age)
    (soldier-count > 20 )
=>
    (set-goal GOAL-FAST-ATTACK 1)
)

(defrule
    (current-age == feudal-age)
    (goal GOAL-FAST-ATTACK 1)
    (timer-triggered TIMER-ATTACK)
    (soldier-count > 20 )
=>
    (set-strategic-number sn-percent-attack-soldiers 100)
    (attack-now)
    (set-strategic-number sn-number-defend-groups 0)
    (disable-timer TIMER-ATTACK)
    (enable-timer TIMER-ATTACK 30)
)
```

EXAMPLES

Herzog Zwei, the granddaddy of RTS games, was really more an action game with the added fact that you had to get money to get more equipment. There was no real pathfinding, enemies would constantly get stuck, and you could trick the AI builder unit so that it was impossible for it to fight back. For the most part, *Herzog* was probably coded using a very simple state machine, with the states being Get Money, Attack, and Defend.

Westwood Studio's® *Dune: The Building of a Dynasty,* which came out two years later, started the formula that mostly continues today, in which players build a town, mine resources, span a tech tree, and fight enemies. The game didn't have the best AI, but understandably so, given the minimal system requirements of the game. *Dune* used mostly just an initial defense build order, followed by a phase of finding your base, and then attacking you. It wouldn't really rebuild its defenses (because they were only built during the opening phase), it wouldn't attack anywhere but the side of your base facing his base (no real flanking or trying to find weaknesses), and it cheated fairly badly (it didn't seem that the AI ran out of money, and it could build it's structures unconnected, whereas the human could not).

The golden age of RTS games included the *Command and Conquer* series, *Warcraft, Starcraft,* and many spin-offs and imitations. During this time the AI continued to push forward, the biggest improvement being pathfinding, but the games were still plagued by AI exploits that human players would find very quickly. This was due mostly to not having the processing power or memory space necessary to use things like influence maps or better planning algorithms.

More modern games—such as the *Age of Empires* series, *Empire Earth, Cossacks,* and the like—have built on these modest foundations and created full-featured games with plenty of challenge and fairly good AI opponents. Although some problems are perennial (such as formations mucking with pathfinding, and diplomacy AI being all but absent), these games can, and will, give human players a run for their money, without cheating (for the most part) and without exploits. Most of these titles use some form of advanced terrain costing to further their pathfinding. Most do some planning to determine goals and subgoals. Starting build orders are still usually quite popular, simply because of their ease of implementation and the tunable way that they affect difficulty level.

Some modern RTS games have changed direction a bit, with *Warcraft III, Command and Conquer: Generals,* and *Age of Mythology* being notable examples. These games have started emphasizing the use of champions or superunits, instead of throngs of mindless units. These champion units are tougher, more capable, and more expensive to build and to lose. They also employ a much higher amount of mission scripting, so that the game has a much more crafted feel, instead of the

TeamLRN

missions of earlier RTS games where you were just pitted against larger and larger opposition forces.

AREAS THAT NEED IMPROVEMENT

Learning

RTS AI too often gets caught in the same trap repeatedly. A simple example is readily seen in the *Age of Empire* series (although it is fairly pervasive among all RTS titles), where the computer will march one or two units past a tower (which will kill them) over and over. The AI should definitely take into account successful travel information about map locations (using the influence mapping techniques described earlier) so that it can stop being kill-zoned by smart players who notice lines of migration. Other learning opportunities for RTS games could include opponent modeling opportunities such as keeping track of the direction of player attack, noting which types of units the player favors, or even keeping track of game strategies across multiple games against a particular player. Does the player use early rushes? Does the player rely on units that require a lot of a certain resource? Does he frequently build a number of critical structures in a poorly defended place? Are his attacks balanced, or does he build lots of rocks, lots of paper, but never any scissors? When you start attacking a remote base, how long does it take him to respond? The answers to these kinds of questions could be stored along with statistics that would allow a smart AI system to respond to these kinds of issues and more. Using this kind of information doesn't mean that the AI slowly becomes unbeatable; it just means that the human has to switch tactics to win, somewhat forcing him to investigate other areas of the game's complexity. An AI opponent that is shutting down specific player offensive maneuvers doesn't necessarily mean that the AI *itself* has to be aggressive, unless the player has set the difficulty very high.

Determining When an AI Element Is Stuck

At some point, in almost every game, an AI element (from the lowliest economic peon, to an entire group of tanks) might get into a situation where they don't know what to do at all. Maybe all the resource centers are gone, the peon's army has no more money to build another one, and the peon has an armload of coal but doesn't know what to do with it. Or a group of tanks is being hounded by an aerial unit (and cannot fight back), but is also trapped in a close quarters area, and stuck in a pathfinding/fleeing cycle that keeps the tanks going in circles as they try to get away, but trip each other up, over and over again. This type of nasty feedback loop can make an AI element look extremely stupid, but it is precisely the kind of behavior that almost every RTS game has some form of. Detecting this kind of

"stalling" and either having a contingency plan, or some kind of bailout behavior, is essential to help the intelligence of these games.

A secondary type of this is the classic problem where you have to kill all the units in the enemy's army to win, and the AI has one peon unit, hidden behind a tree, somewhere on the map. Which leads to the player searching, for an hour and a half, until he happens upon the peon, who was just sitting there frozen, with nothing to do. The AI in RTS games should be able to recognize when it's been beaten (most do, but even the best get confused sometimes) and offer up a surrender. If the player *wants* to hunt down the last peon, let him; but also give the red and screaming player the chance to see his hard won "Victory!" screen without spending all day hunting for some idiot unit.

Helper AI

For alleviating micromanagement tasks that a human player performs repeatedly during the game, helper AI is an area that screams for exploration. Also mentioned in Chapter 4 during the discussion of RPG party members, "automatic" behavior that units perform on their own could be improved. A flexible system that would add new behaviors (if the game recognizes that the player is always doing a specific small behavior), exhaust unwanted behaviors, and perform with mild intelligence would make actually playing RTS games much more flavorful than the current "Build up, Attack, Build up, Attack" click fest, where the person who knows the best build order and can get things done the fastest wins. Sometimes, yes, that is exactly the game some people want to play. But right now it seems to be the way most RTS games are set up.

In effect, this system would find small behavior macros and then either ask the player if he needs help in doing that or just take over the task (possibly with some sort of "It's taken care of" message communicated to the player). The player could select the level of macro help he'd like, with level 0 being no help, level 5 would find things repeated more than five times and would extinguish these behaviors if you cancelled out of them more than once, and at level 10 it would discern anything you repeated more than twice and never extinguish these rules. At any rate, you would probably also want little macro "flags" to appear somewhere onscreen (or in some quick menu), so that the player could cancel any that he wanted to at any time.

Opponent Personality

Herzog Zwei had two opposing AI personalities (offense based and defense based), and this was one of the earliest, if not the first, of RTS games. It was a very different experience playing against the differing AI personalities. Imagine getting variation not just in difficulty level of the AI, but in other attributes as well. We do this in sports games, or fighting games, why not in RTS games? By using resource allocation

TeamLRN

systems to describe bias toward specific units, or specialization in different branches of the tech tree, we could generate opponents with much more flavor. In the development phase, different stable personalities could be tuned and played against each other, to find the combinations that lead to victory. These personalities could even be replaced by a singular AI opponent over time, so he would start play with a very balanced game, but after a brutal combat loss, might get "mad" and use a much more aggressive resource allocation table to force out more units, for retribution. This would flavor the AI when battling against it, and might carry over into the diplomacy game, so that you might think twice about allying with an AI character that you know has a tendency to turn on his allies, or is a hothead and will become angered by the smallest incursion, turning him into a liability if he's off hunting somebody instead of sticking to a larger agreed-upon battle plan.

More Strategy, Less Tactics

AI micromanagement leads to better per-unit behavior. To be considered human-like, however, RTS games need better strategic team leadership, not individual-unit intelligence that outdoes the human in speed or tedium. Instead of better planning algorithms and squad (or commander) level AI, which is analogous to the way a human plays, most games rely on the computer's ability to quickly micromanage attacking units on an individual basis (or rather, to have unit AI that is not present when a human player is under control, which makes it *feel* like micromanagement). This leaves the AI able to do things that are near impossible for a human, which leads to frustration, and a feeling that the AI is cheating. Perhaps the AI could be given limits on the amount of micromanaging it can do in a given time frame, to simulate the time it takes a human to scroll around, clicking the mouse and hitting hotkeys. In any sense, better strategic systems in RTS games will go a long way toward making the AI in these games more human and, ultimately, more fun to play against. Some things that a superior strategic system should accomplish are these:

- Grouping units by type, and then using groups to back up other groups, or respond to specific threats with the correct counter type of unit grouping. Right now, most battles initiated by the AI opponent are started by the AI generating a mix of units based on a scripted combination that works well together, affected by the resources the AI has, and to some lesser degree by the types of units they expect to see from the human player. This is a good start, but that's where the strategic AI in most games ends. Once this war party actually reaches the human's forces, the AI could respond to what is actually there much more efficiently using a commander level of AI decisions that targets enemies with good counter units and makes adjustments as the battle ensues, just like a person would.

- Setting up attack lines, to take advantage of multiple fronts, and leave support lines open for additional forces to come in. Again, most RTS games suffer from using the individual-unit AI far too much once the battle has begun. They also don't use much in the way of attack scheduling. Splitting up an army, and coming from two sides, is a technique used when an advancing enemy places units where they are not protected very well. But it requires that these two fronts be timed so that they happen fairly concurrently, otherwise all you've done is split your army in two.
- Using terrain features to set up optimal wall structures. Wall construction separates good RTS AI from the truly great. Some games use a random map generator to keep multiplayer games fresh, so the need for a dedicated wall constructor is paramount to make quality, useful walls that still use terrain features to their advantage.
- Schedule retreats if they are foreseeable, or just initiate them if everything falls apart. Battles with numbers of units *going kamikaze* should only really happen if there are larger motives at play, that of using their sacrifice as a diversion (to attack another front, or make a run for a particular resource, or something), if that force is specifically designed to fight against some entrenched enemy defense, or if the AI has some sort of "micromanage" points (to prevent it from doing things faster than a human could do) and spends them all doing something else on the map so cannot pay any attention to the losing battle. Retreats should be a bit more elegant than just selecting every unit and giving them a destination of home base.
- Set up ambush situations, or cover lines of retreat for advancing armies. A common strategy that human players employ is to keep a large force back from the front lines, and then have a few fast units go forward and draw some enemy forces from their entrenchments and back to this waiting ambush. Or, the human will use these fast units to draw a considerable number of the defense forces away from one side of his enemy's main base, and then send in the larger force to this less-protected area. Either way, the essential strategy is to protect the line of retreat by any of your forces. If they have to fall back, the AI won't have to worry about fast enemy units following the retreat line and picking off slower units trying to flee.

SUMMARY

RTS games have given game players the amazing opportunity to be generals in charge of an entire army, complete with an economy to replenish that army. Because of the tremendous number of units and possible actions going on in real-time throughout the map, the AI challenges in RTS games are particularly large.

TeamLRN

- Individual-unit AI gives personality to units, without clogging the higher-level AI systems.
- Economic AI needs to be carefully tuned so that human players don't have to micromanage too much, or too little.
- Commander-level and team-level AI provide increasingly more strategic layers to the system, to keep each layer simple and easy to maintain.
- Town building AI is a unique challenge that must account for factors such as protection, visibility, and forward planning to look intelligent.
- Pathfinding takes up a large percentage of CPU cycles because of the numbers of units and the complex terrains. But, a good pathfinder implementation is also paramount to the success of the game.
- Support AI systems that are important to RTS games include terrain analysis, opponent modeling, resource management, reconnaissance, and diplomacy systems. Each delivers an important part of the RTS experience.
- Messaging is a very important AI technique for RTS games because of the high level of communication that needs to occur.
- FuSMs are a good way to model the huge amount of imperfect information that RTS AI systems have to deal with, along with the many directions that a team has to split its resources and attention.
- Hierarchical AI systems, as well as planning algorithms and scripting systems, are also key elements to many RTS AI engines.
- Learning, either directly, or through secondary means (like influence maps) can make the AI in RTS games far more adaptive.
- Determining when a unit (or entire game element) is stuck is a problem that many RTS games have not solved very well.
- Helper AI could be used as a layer that runs when a human is playing the game, to help alleviate micro tasks by giving the player the option of taking them over automatically.
- Opponents in RTS games rarely exhibit any personality, and as such, the human doesn't really connect at all with his opponent.
- RTS games need to concentrate on more strategic battle elements, and less individual-unit tactical AI.

7 First-Person Shooters/ Third-Person Shooters (FTPS)

FTPS games are the other major genre in which AI work has been embraced both from inside the industry and in classical academic research areas, mostly because of early efforts by Id Software. Most of Id's games have pushed the envelope for graphics and network programming has been groundbreaking in the area of user extensibility. Other leading games have followed suit. Many FTPSs include tools that people can use to add levels, change weapons, script new AI elements, and even perform what is called a "total conversion," meaning that the entire game has been changed in some way. An entire "mod" scene has sprung up, with many Web sites where people can get information about modifying their favorite game and download modifications (mods) created by other users.

One mod that makes specific use of AI techniques is called a "bot." This is what the FTPS world refers to as an autonomous agent that can navigate a map, find enemies, attack them intelligently, and respond to injury, powerups, and so on. See Listing 7.1 for a sample chunk of code from a *Quake* bot. Some bot writers have gone on to get legitimate jobs in game development because of their independent work in the mod world. A good example is Steve Polge, writer of the Reaper Bot (one of the earlier and more famous bots), going on to be the AI programmer for *Unreal.* Many level editors have gotten their start in the mod community as well. Interviews with companies doing FTPS games are often preceded by showing the interviewer levels or modifications that a candidate has done independently, often with good reviews from community sites.

LISTING 7.1 *QuakeC* Sample of User Defined Script for an AI-Controlled Bot

```
void (float dist) ai_run = {

   local vector delta;
   local float axis;
   local float direct;
   local float ang_rint;
   local float ang_floor;
   local float ang_ceil;
```


TeamLRN

```
movedist = dist;
if ( (self.enemy.health <= FALSE) ) {

   self.enemy = world;
   if ( (self.oldenemy.health > FALSE) ) {

      self.enemy = self.oldenemy;
      HuntTarget ();

   } else {

      if ( self.movetarget ) {

         self.th_walk ();

      } else {

         self.th_stand ();

      }
      return ;

   }

}
self.show_hostile = (time + TRUE);
enemy_vis = visible (self.enemy);
if ( enemy_vis ) {

   self.search_time = (time + MOVETYPE_FLY);

}
if ( ((coop || deathmatch) && (self.search_time < time)) ) {

   if ( FindTarget () ) {

      return ;

   }

}
enemy_infront = infront (self.enemy);
enemy_range = range (self.enemy);
enemy_yaw = vectoyaw ((self.enemy.origin - self.origin));
if ( (self.attack_state == AS_MISSILE) ) {
```

```
        ai_run_missile ();
        return ;

    }
    if ( (self.attack_state == AS_MELEE) ) {

        ai_run_melee ();
        return ;

    }
    if ( CheckAnyAttack () ) {

        return ;

    }
    if ( (self.attack_state == AS_SLIDING) ) {

        ai_run_slide ();
        return ;

    }
    movetogoal (dist);

};
```

Because of this extensibility (and the product's stability), some of Id's games have become test beds for AI research in academia. Many diverse research labs are using their games, with heavily modified code, to test AI techniques under conditions that are much closer to modeling real-world situations than used in the lab before, and with much more realistic time constraints. Everything from new ways to store environment information, to faster planning algorithms, to complete rule inferring systems have been tested using the game.

Another type of FTPS game that has become very popular lately is the squad combat game (SCG). This is a FTPS game in which the main "character" isn't just a single person but, rather, a squad (usually about 3–10 people) working toward a common goal. This type of game started out as a game mode in regular FTPS games, called Capture the Flag (both teams have a flag, and if you can get the other team's flag and return it to your base while you're still in possession of your own flag, your team gets a point). This concept was expanded into full-blown military squad simulations. The AI for these types of games gets pretty complex because squad group maneuvers and coordination is a much harder problem than the more straightforward FTPS games.

TeamLRN

COMMON AI ELEMENTS

Enemies

The main thrust of FTPSs is to have enemies, and lots of them. As such, the AI of these enemies is vital to the longevity of the product. Many games have touted "better enemy AI" for their game, only to have it shot down by exploits almost immediately.

Other FTPSs have used what some call "arcade AI," which is the simple pattern AI of old-style arcade games. *Doom* and the modern *Serious Sam* games use this technique very well. They gave the player a chance to simply run around with the biggest gun and destroy everything in his path, which is just what some people wanted. Still other games, such as *Half-Life,* tried for a much more scripted, intelligent, and rich gameplay experience, and were also successful.

How much work you put into your enemies (in the single-player portion of the game, anyhow) is directly related to the type of gameplay experience you are striving for. Strange, though, is the notion that both the arcade and scripted types of FTPS games are hard to do well. *Doom* hit a perfect balance with its mindless enemies, great level design, and weapon balance. It spawned countless copycats, almost all of which were not as good. *Half-Life* did the same with scripted content in an FTPS game, by having a great story, tons of hand-tuned situations and nonplayer character (NPC) behavior, and good atmosphere. These efforts were followed by a vast number of games seeking to do the same, but nobody really did it better.

Boss Enemies

Some of the more action-based FTPS games, such as *Serious Sam,* also contain Boss enemies as in a shooter or a role-playing game (RPG). At the end of the level, you come face to face with a (usually) larger and more powerful enemy, complete with special attacks and movement abilities. But even the more complex games like *Half-Life* had some really big creatures to tackle. These creatures are generally very tough but have some weakness that can be exploited if discovered. Some even required you to use elements of the environment to kill them.

Deathmatch Opponents

The AI opponents necessary for FTPS games fall into two basic categories: regular monster enemies and deathmatch bots. Monsters are creatures that are expected to act like beasts, or at best, evil humanoid killers. Bots, on the other hand, are trying to closely model human behavior and performance during deathmatch games. Some bots have been created to caricature certain behaviors (such as bots that only use a particular weapon and are always jumping, for instance), but they are mostly trying to model good, solid, human deathmatch execution.

If you plan to add a multiplayer portion to your product, you are going to need bot AI so that players can have a multiplayer experience if they don't have a means of connecting to someone else, or just want to practice. Unlike the regular enemies in a FTPS game, these guys are supposed to be as smart and as human as possible (with difficulty levels, of course) to provide the player with a fun, yet challenging run through the deathmatch environments.

The difficulty levels usually involve tweaking different aspects of the bot's behavior, such as aggressiveness, how often the bot will retreat and load up on health powerups, the appropriateness of weapon usage (or does the bot have a favorite weapon that it uses much better), as well as how good the bot's aim is.

Other behavior that is gradually finding its way into bot behaviors includes using chat messages to taunt players it has killed, or commending another player on a good shot. Although still very simplistic, the effect is becoming better as games continue to use it. Who knows, in the future, maybe we'll have the equivalent of full chat bots within our FTPS games, to make them seem even more human if nothing else.

Weapons

FTPS weapons have run the gamut from the seminal rocket launcher to the very odd "voodoo doll in *Blood* that had players stick pins in their enemies from afar. With weapons that bounce around corners, leave trails of deadly goo, or have to be steered like heat seeking missiles, sometimes it takes intelligence just to use some of the weapons that these games use. Other weapon intelligence issues involve not shooting splash damage weapons when the bot itself might be hurt by the effect, or even strange usages of the weapons, such as the electricity gun discharge in the first *Quake* game (if you shot the electricity gun into a pool of water, it would kill anybody in the pool, including the original gun owner if he was in the water). It could even be said that knowing which weapons work well in a duel against other weapons, as well as which weapon to pick, given player type, range, and amount of ammunition, is a definite intelligence test.

Cooperative Agents

An element that started showing up when more complex or story-driven FTPS games started to appear, cooperative agents are "helper" bots, or NPC types that would inhabit a level. When the player interacted with these special characters (instead of killing them), they might offer help or a new weapon, and so on. Some of these characters were quite complex, following you around a level, helping with enemies, and pointing out features of the map.

Games that have used this element successfully are *Half-Life, Medal of Honor: Underground,* and many others. Just as with RPGs, cooperative agents need to have

TeamLRN

enough "smarts" so that the player doesn't feel like he's babysitting; otherwise, he will quickly abandon the agent, or become frustrated with the game.

Squad Members

If you're constructing a game based on squad combat, then you're going to be spending an awfully large amount of your time in this category. Squad-based maneuvers range from the simple (leapfrogging forward movement while providing cover) to the very complex (part of a squad breaking off, to take out a guard post, while the main group continues forward, to remove a different guard, and then both groups meet at some point). The AI controlling these squad members needs to be reactive (don't just keep running to a spot because I told you to if you're being fired at; rather, get some cover, look for the source of fire, and then use some smart means of either communicating back to the commander, or using the terrain features to get to the target safely), proactive (if a grenade gets lobbed into our trench, someone should pick it up and lob it back, or jump on it), and communicative (about their success and failure, any slowdowns they are incurring, additional information they have uncovered, etc.). If you're making a game that is a bit less military (for example, a game where a player and his virtual family have to defend their home against alien attack), you would also need to account for some additional personality issues, including being calm under fire, dealing with being injured, panic, and seeing violence. These are all things that a professional soldier is trained to do well, but if you see your eight-year-old sister doing fine while under heavy laser fire with a serious leg wound, you might think it was pretty unrealistic.

On top of all this, the squad-level AI systems need to make the team competent, but not unstoppable. This is the fine line of game balance. If the squad is *too* capable, the player feels like a bystander, but if the squad is not capable enough, the player feels like he's surrounded by idiots. This is where gameplay testing comes into play—and plenty of it.

Pathfinding

Listed separately, even though it will be a part of both enemies and cooperative agents, pathfinding is one of the primary AI systems in an FTPS. Whereas in real-time strategy (RTS) games, pathfinding usually encompasses only terrain management, FTPS pathfinding usually involves using game elements (such as elevators, teleporters, levers, etc.) and specialized movement techniques (the "rocket jump," crossing underwater sequences that might hurt if not done correctly, etc.). As such, pathfinding games usually employ a combination of specialized level data, alongside custom pathfinding "costing," which can help account for special movement oddities. Local pathfinding for dynamic obstacles, or *obstacle avoidance*, is used to help with more immediate area problems. Avoidance can complement or com-

pletely override the normal pathfinding system, based on context. If a character has his back to a corner, and he's being pinned there by some other player or environmental element, the pathfinding system also needs to recognize this state as being "stuck" and have some sort of exit contingency for the character. Your autonomous AI-controlled characters can and will find every sticky spot on the map to get wedged into, and the look of your pathfinding system will suffer dramatically. By leaving nothing (or near nothing) to chance, you can allow the level designers free reign to create any environments they want to, and still give your creations a fighting chance to navigate them successfully.

Spatial Reasoning

In the same way that RTS AI systems need to use terrain analysis to find specific elements inherent in the game world that humans are good at locating (such as bottlenecks and crucial resource sites), FTPS games need to model the kinds of determinations that humans make about areas of the game world. Humans are very good at looking at an environment and finding sniper locations, choke points, good environmental cover, and such. However, this is a pretty difficult problem to tackle in a real-time environment (RTS games can use a simplified, overhead two-dimensional [2D] version of the map to make these kinds of determinations). As such, this problem is usually considered another step for level designers to help with, by tagging areas of the map with helper data that the AI opponents can discern and use to their advantage. However, systems that can perform this process automatically on a level have been developed, usually as a preprocessing stage that produces this spatial reasoning data in some usable form.

USEFUL AI TECHNIQUES

Finite-State Machines (FSMs)

The staple of the AI programming world makes its appearance again. FSMs can be exclusive (*Serious Sam),* or used only for fairly simple elements (as in *Half-Life*). Also, the life span of most enemies in these games can be very quick; no real forward planning is usually needed. Deathmatch AI for these games involves a minimum of states: usually along the lines of attack, retreat, explore, and get powerup. The rest of the intelligence comes from special navigation systems, the movement model for the bot, and other support routines. See Listing 7.2 for a snippet of the AI FSM code from *Quake 2.* This function is used to determine if certain AI states (namely `ai_run` and `ai_stand`) should transition to `ai_attack`. Note the comment line labeled JDC, the initials of John Carmack. Also notice the `//FIXME:` comment that is in the final released code. It's good to know that John is still human.

TeamLRN

LISTING 7.2 *Quake 2* AI Code Snippet. © Id Software, licensed under the GPL.

```
/*
=============
ai_checkattack

Decides if we're going to attack or do something else
used by ai_run and ai_stand
=============
*/
qboolean ai_checkattack (edict_t *self, float dist)
{
    vec3_t        temp;
    qboolean    hesDeadJim;

    // this causes monsters to run blindly to
        // the combat point w/o firing
    if (self->goalentity)
    {
        if (self->monsterinfo.aiflags & AI_COMBAT_POINT)
            return false;

        if (self->monsterinfo.aiflags & AI_SOUND_TARGET)
        {
            if ((level.time - self->enemy->teleport_time) > 5.0)
            {
                if (self->goalentity == self->enemy)
                    if (self->movetarget)
                        self->goalentity = self->movetarget;
                    else
                        self->goalentity = NULL;
                self->monsterinfo.aiflags &= ~AI_SOUND_TARGET;
                if (self->monsterinfo.aiflags &
                        AI_TEMP_STAND_GROUND)
                    self->monsterinfo.aiflags &=
                        ~(AI_STAND_GROUND | AI_TEMP_STAND_GROUND);
            }
            else
            {
                self->show_hostile = level.time + 1;
                return false;
            }
        }
    }
```

```
enemy_vis = false;

// see if the enemy is dead
hesDeadJim = false;
if ((!self->enemy) || (!self->enemy->inuse))
{
    hesDeadJim = true;
}
else if (self->monsterinfo.aiflags & AI_MEDIC)
{
    if (self->enemy->health > 0)
    {
        hesDeadJim = true;
        self->monsterinfo.aiflags &= ~AI_MEDIC;
    }
}
else
{
    if (self->monsterinfo.aiflags & AI_BRUTAL)
    {
        if (self->enemy->health <= -80)
            hesDeadJim = true;
    }
    else
    {
        if (self->enemy->health <= 0)
            hesDeadJim = true;
    }
}

if (hesDeadJim)
{
    self->enemy = NULL;
    // FIXME: look all around for other targets
    if (self->oldenemy && self->oldenemy->health > 0)
    {
        self->enemy = self->oldenemy;
        self->oldenemy = NULL;
        HuntTarget (self);
    }
    else
    {
        if (self->movetarget)
        {
```

TeamLRN

```
            self->goalentity = self->movetarget;
            self->monsterinfo.walk (self);
        }
        else
        {
            // we need the pausetime otherwise the stand code
            // will just revert to walking with no target and
            // the monsters will wonder around aimlessly trying
            // to hunt the world entity
            self->monsterinfo.pausetime = level.time +
                                                100000000;
            self->monsterinfo.stand (self);
        }
            }
            return true;
        }
    }

    self->show_hostile = level.time + 1;// wake up other monsters

    // check knowledge of enemy
    enemy_vis = visible(self, self->enemy);
    if (enemy_vis)
    {
        self->monsterinfo.search_time = level.time + 5;
        VectorCopy (self->enemy->s.origin, self->
                                            monsterinfo.last_sighting);
    }

// look for other coop players here
//    if (coop && self->monsterinfo.search_time < level.time)
//    {
//        if (FindTarget (self))
//            return true;
//    }

    enemy_infront = infront(self, self->enemy);
    enemy_range = range(self, self->enemy);
    VectorSubtract (self->enemy->s.origin, self->s.origin, temp);
    enemy_yaw = vectoyaw(temp);


    // JDC self->ideal_yaw = enemy_yaw;
```

```
if (self->monsterinfo.attack_state == AS_MISSILE)
{
    ai_run_missile (self);
    return true;
}
if (self->monsterinfo.attack_state == AS_MELEE)
{
    ai_run_melee (self);
    return true;
}

// if enemy is not currently visible, we will never attack
if (!enemy_vis)
    return false;

return self->monsterinfo.checkattack (self);
}
```

Fuzzy-State Machines (FuSMs)

Fuzzy-state machines have also been implemented in these games, especially because the number of fuzzy variables is usually low, so you don't run into the problems of combinatorial calculation growth that hurts fuzzy systems. Also, the states of inputs from which FTPS opponents must make their determinations are rarely as crisp as regular (i.e, nonfuzzy) states are considered to be. An AI-controlled opponent might be at 23% health, but have a really good weapon, and is also coming up behind the human player, unseen by the player. So, even though the AI opponent is very damaged, should he take the shot? The answer is probably yes, but only when you think of the system using a combination of the various fuzzy inputs to this agent. Again, this is only relevant when you consider the types of enemies you are programming for. Shooting the human in the back isn't very entertaining behavior, unless you are creating a deathmatch opponent.

This technique also works well because of the way many of these games portray their animation. The upper and lower bodies of the characters are usually almost completely decoupled from each other. The lower half tries to play some running animation that corresponds to the direction of travel, while the upper half aims, fires, and switches weapons. This leads nicely to a fuzzy solution where two states might be activated at lower levels, a character might be shooting at you, but also running for a health powerup, the result of a fuzzy-state system that treated "50% shoot, 50% get powerup" as a solution.

TeamLRN

Messaging Systems

In more deathmatch-style FTPSs, the thrust of the gameplay could somewhat be described as "a physics model with input handlers" (meaning that the gameplay is pretty much just taking input from the humans, using the physics code to move everybody around, and making the missile weapons collide with the players). Because of this, using a messaging system in the game proper is a good idea, in that you have a stable underlying system (the physics) running constantly, with events marking things happening of any interest (such as firing a rocket, or player X entering #23 teleporter). Plus, most of these games are multiplayer, and quite a few use the server-client model, so this makes using the message system to construct the AI a good idea. You could have a small state-type system, with changes in state being initiated by events, causing incoming messages, or "internal" messages (going to a getting hit state when you receive the message `IveBeenHitByRocket`, for instance).

Messaging also makes a lot of sense in SCGs because they need to be passing information back and forth among squad members quite regularly, including sharing a lot of information about visible threats, positions, status, and much more.

Scripting Systems

Some modern games have used a high level of scripting in their FTPS games. Everything including elements in the environment, enemies, conversations, player inter-

sides sequels to our perennial favorites, including *Unreal*) based in the realm of war themed games. *Battlefield: 1942, Call of Duty,* and *Battlefield: Vietnam* are very popular games that capture much of the grit of real war, while still looking very good and playing well. Purists of war gaming are not amused by some of the license that has been taken with historical details, or weapon details, but the medium-level shooter crowd is really enjoying the inclusion of a more realistic world (without wondering who's going to come around the corner with the BFG and blow a hole in the entire world), as well as the inclusion of all the vehicle types that many of the war FTPS games allow, including tanks, boats, and even planes.

AREAS THAT NEED IMPROVEMENT

Learning and Opponent Modeling

Holes in the AI's behavior are found and exploited in FTPS games, just like any other game genre. Because FTPSs are often played online in multiplayer situations,

Games such as *Hexen, Blood, Heretic,* and the like are all good examples of games that fell into this category. *Heretic* was also one of the early third-person shooter games to really give the new formula a great interface.

With the next level of FTPS games, we suddenly got a full taste of our real new addiction, online multiplayer deathmatch. Before this time, only those lucky enough to work at a computer company with a LAN, or with more than one in-home computer that you could string a null modem between had felt the terrible goodness of this mode. But finally, programmers were discovering ways of getting decent gameplay over the Internet, even with a dialup connection, and everybody wanted in on it. The games got better, *Quake* and *Unreal* being the top two. Also during this period, Id made *Quake* highly extensible for the end user (with *Unreal* following suit) and, thus, led to the development of the deathmatch bot, which forever changed the FTPS AI world. People started to see what an FTPS enemy could do, given a degree of intelligence, and started demanding more challenging enemies in the single-player portion as well. This led to a much higher level of AI complexity across the board.

Today, a new variant on these games is taking over people's free time. It's called squad combat, and some of the best are *Socom* and *Tom Clancy's Rainbow Six.* These games include all the regular FTPS AI and involve the coordination of multiple team members (if playing the single-player game and aren't being helped by other humans) in real-time combat missions against teams of enemies. There is a fine balance in these games between the high-level commands that you send to your team members and the realistic tactical AI that they need to perform to operate well in concert.

The last batch of FTPS games to come out have been almost completely (besides sequels to our perennial favorites, including *Unreal*) based in the realm of war themed games. *Battlefield: 1942, Call of Duty,* and *Battlefield: Vietnam* are very popular games that capture much of the grit of real war, while still looking very good and playing well. Purists of war gaming are not amused by some of the license that has been taken with historical details, or weapon details, but the medium-level shooter crowd is really enjoying the inclusion of a more realistic world (without wondering who's going to come around the corner with the BFG and blow a hole in the entire world), as well as the inclusion of all the vehicle types that many of the war FTPS games allow, including tanks, boats, and even planes.

AREAS THAT NEED IMPROVEMENT

Learning and Opponent Modeling

Holes in the AI's behavior are found and exploited in FTPS games, just like any other game genre. Because FTPSs are often played online in multiplayer situations,

TeamLRN

however, these holes are found even faster, and people will pass on this knowledge very quickly. FTPSs run the risk of becoming very repetitive, simply because even if you change the location, the enemy, and the weapon, you're still just hunting someone down and shooting him. AI enemies need to react much more to the personal playing style of their opponents from a game longevity standpoint. Enemies could keep track of various stats to affect their gameplay style, such as the following:

- *The weapons the human uses most.* Most people specialize, either because the damage of a certain weapon is high (such as the rocket launcher that everybody seems to love in the various *Quake* games), or because they have an affinity for a certain weapon and have practiced special techniques with it (such as the nail gun in the original *Quake,* which bounced around corners and could be really nasty if you took the time to find nice spots to fire at that would bounce to commonly tread areas of the map; or the devilish places people found to put *Duke Nukem 3D* trip mines).
- *The routes through the map the human uses.* One popular method of playing these games is to learn a good route through the map that takes you in contact with all the major powerups while keeping you moving so you don't get caught napping. The AI could discern these routes and either watch for the player along the route, or fire rockets and such down corridors that the human routinely uses, forcing the player to change his game.
- *The close-quarters combat style of the human.* If the human always circle strafes to left, for instance, the AI could use this to better dodge the oncoming fire.
- *The type of player the human is.* This mostly refers to the level of movement that the human employs while playing. It usually goes from high level of movement (or a Hunter type), to medium movement (or a Patroller type), to almost no movement (what is known as a Camper type). There are many others, but these are the primary types. By knowing this information about the player, the AI opponent can fine-tune how he looks for the player.

Personality

Even though the bots of today play well, and you don't feel like they're cheating you (at least not too much), they fall far short of having the kind of personality that you can sense when playing against another human. Especially when you play someone regularly, you can get a sense of his personality (his aggression level, how rattled he gets under fire, does he camp, etc.) and the range of personality (for example, the opponent is usually even headed, but in the final three minutes of a game, he goes berserk). Bot creation usually involves their weapons of choice, and their overall level. More personality would actually lead to a more immersive environment, as you learn the ins and outs of the bot. Plus, semirandom changes in this behavior would then actually be noticed and might throw the player off.

Creativity

Playing against humans, you can see the vast array of new and unique ways to use the weapons and environment that they have found. A FTPS with a very solid physics model (without many special cases, to allow for stable math) could either note human player trajectories and figure out how the human got there (by jumping and then firing a rocket sideways, to send the player flying high speed to another ledge), or could randomly try different ways of employing themselves and then tag their internal models of the level with these new ways of traversal. Many humans bounce around the map by jumping or using the backlash from weapons, and it makes them much harder to hit.

Although true creativity might be beyond the scope of an AI system, AI programmers could come up with a much richer degree of environment usage by the AI, and the overall effect would be that of a bot that "really knows the level well," an affectation usually given to players that can traverse the level in novel ways and attack their opponents by strange means.

Anticipation

One thing that good players employ all the time in FTPS games is the concept of anticipation. You might watch a player go into a room, and because there is only one door, time the firing of an area effect weapon so that it will hit the player as he comes back out the door.

This would require the AI to keep a mental model of the other player, and estimate how long it would take the player to enter the room, go to whatever powerup made the player enter the room in the first place, and then come back out. Then, the AI could set up the shot, or a more personal ambush, using this time estimate to determine how long is needed. This would be a fairly advanced move, but if a human player truly wants to practice what online play is like, this is the type of opponent he will need to acclimate to.

A lesser version of this would be to set up ambushes, either by reasoning that another player will use a certain doorway and lying in wait for him to come or by getting the attention of an enemy, running away, and waiting for him to follow from some safe spot that the AI has scouted out earlier.

Better Conversation Engines

Right now, the state of the art for FTPS AI talk back is along the lines of canned one-liners that the AI shouts when it's just killed you, or you killed it. These get repetitive quickly and are almost never contextual or interesting. With a small grammar and some semblance of a sentence engine, the AI could use more contextual shouts that actually work to drawing the player in by bringing a sense of realism.

TeamLRN

Motivations

AI FTPS bots right now have two primary motivations: stay alive and kill you. Some don't even care if they stay alive. But people don't fight like that. They get angry, sometimes with specific people. Or, they get rattled, and retreat for a while until they settle down. AI systems need to model this behavior, at some level, to mimic their human counterparts more truthfully. Imagine AI bots that call for a temporary truce with you, to team up on other human players, or that can't stand campers (people who sit in hidden spots and snipe players from afar) and hunt them down exclusively. These types of more emotional behavior, combined with a bit higher verbal output, might just make them seem much more human.

Better Squad AI

Most of the squad-based games have relied on very simple squad commands (cover me, follow, stay here, etc.). These types of commands are obviously easier to code, but were also used because the interface necessary to run a squad needed to be fairly simple, so that it can be used quickly and efficiently during battle.

Better would be a context-based menu of possible answers to the current situation, sort of like playbooks for football. The commander could choose which one he wanted to use, the squad would start it up, and from there, the commander could direct single soldiers to do something different, or change the play. With this system, the designers could implement a number of base strategies for any given incursion, custom tailoring squad formations, and the types of actions that each play entails. The human player could vary from this formula by directing certain soldiers to do other things, but these plays could be used to quickly set up each soldier with a workable plan. The different types of solutions presented to the player for each game situation might be attitude based (aggressive versus defensive), goal based (save ammo, spread out, etc.), or even time based (use extreme caution versus run now). Thus, the type of commands employed by the human player would create the overall battle flavor. He could experiment with the different solutions to find the one that he felt most comfortable with, as well as the types of formations that left him open for more victories, or even more interesting game situations.

SUMMARY

FTPS games involve some fairly disparate types of AI programming, from simple creatures to deathmatch bots with personality and style. The mindless enemies of the genre's roots have been replaced by intelligent systems that are capable of almost anything that a human player is.

- Early FTPS games set the stage for AI research to be done in their games by making most game code accessible and extensible; this led to user-made modifications, or mods.
- Deathmatch bots were one of the mods that brought another level of AI to the genre, by creating fully autonomous agents that explored the level, hunted players, used weapons and powerups intelligently, and generally acted like regular human players.
- Regular enemies in an FTPS game refer to those implemented in the single-player campaigns, either the mindless arcade-style enemies, or the more scripted story following the style of enemy.
- Deathmatch AI is also required, so that people who don't have access to an Internet connection, or just want to practice, can play in a deathmatch setting against somebody.
- Cooperative AI bots have given some games an infusion of story and broken up the action by providing the player with people help during parts of the game, or by interacting with them in some way other than combat.
- Squad AI refers to the systems that need to be in place for games where the player is controlling more than one character, and the others need to be CPU controlled. The intelligence of these bots needs to be high, but the competence needs to be closely tuned, so that the player feels important, but not alone.
- Pathfinding in FTPS games can be especially tricky because the environments are usually fully 3D and can have very complex constructions. They also include a number of additional gameplay elements, such as ladders, elevators, teleporters, and the like that require pathfinding attention.
- Spatial reasoning provides the AI controlled characters with ways in which to find level-specific areas of concern, such as sniper points or good places for cover and visibility.
- FSMs are put to work in FTPS games, but so are FuSMs because of the nature of inputs in FTPS games.
- Messaging makes a lot of sense in this genre. Regular FTPS games can benefit from it because of the inherent event-driven gameplay (move, shoot, get hit, etc.), and the nature of a server-based online model. SCGs can also use the messaging system to coordinate information back and forth between characters easily.
- Scripting is used in those FTPS games that are going for a more handcrafted feel, rather than the classic "we made the rules, and a bunch of levels" mentality.
- By endowing our creations with even modest learning and opponent modeling, we stop the stale breaking down of gameplay into finding the best weapon, and using it repeatedly by getting the player to mix up the action a bit.
- Creative solutions to movement and attack positions would give AI opponents a considerable advance toward true deathmatch intelligence.

TeamLRN

- Anticipation of impending events would allow AI characters to set up direct, as well as impromptu, ambushes by keeping a mental model of the possible future.
- Better conversation engines might change the canned shouts and taunts in today's games to more context-based, and thus more realistic, banter.
- Giving AI opponents the ability to change motivation might lead to advanced concepts such as temporary truces, or to showing some sort of emotional flare-up.
- The AI employed by most squad games is very simple, but could lend itself well to a contextual, quick command system that would lead to better-looking squad maneuvers and quicker control of the situation by the human.

8 Platform Games

Platform games are a staple of the console world—the classic *Donkey Kong* is a very early example—and have progressed yearly until such epic games as *Ratchet and Clank* have arrived. Platform games are one of the consummate gaming exercises and will most likely always be with us in some form or another.

Platform games of old were mostly 2D, single-screen, *Mario Bros*-style of gameplay. A character would start on the bottom of the screen and have to navigate enemies and the environment using mostly jumping (hence the name, "platform" game, stemming from the need to leap from platform to platform). These were very popular in the arcade world because they presented a new type of challenge: using timing, to the tried and true arcade concept of pattern recognition (before platformers, most arcade games were almost completely about patterns, either shooters with patterns of enemies coming at you like *Galaga,* or simple enemy patterns to be avoided like *Pac-Man* and *Frogger*). Platform games still had mostly patterned enemies (because the technical reasons for using patterns hadn't gone away), but now the player was expected to precision time jumps over enemies and from ledge to ledge to traverse the level and gain the summit.

Later, this concept was expanded into the scrolling platform game, which gained huge favor. This type of game was almost identical to the early platform games, but added the notion of a continuing world, which continually scrolled by as you continued forward. So, instead of a single screen that you had to ascend, you now had an entire world of challenges, which only revealed themselves to you as you progressed into the level. *Super Mario Bros., Sonic the Hedgehog,* and *Mega Man* (see screenshot in Figure 8.1) were influential games in this category, each spawning many sequels and hundreds of imitators.

In 1995, a PC game called *Abuse* was released by a company called crack.com, which later released the entire source code for the product. *Abuse* was an advanced 2D scroller, with fully networked multiplayer support, and an almost first-person shooter/third-person shooter (FTPS) game feel. As shown in Listing 8.1, which is a sample from the sources, the AI for the enemies in *Abuse* was written in LISP.

TeamLRN

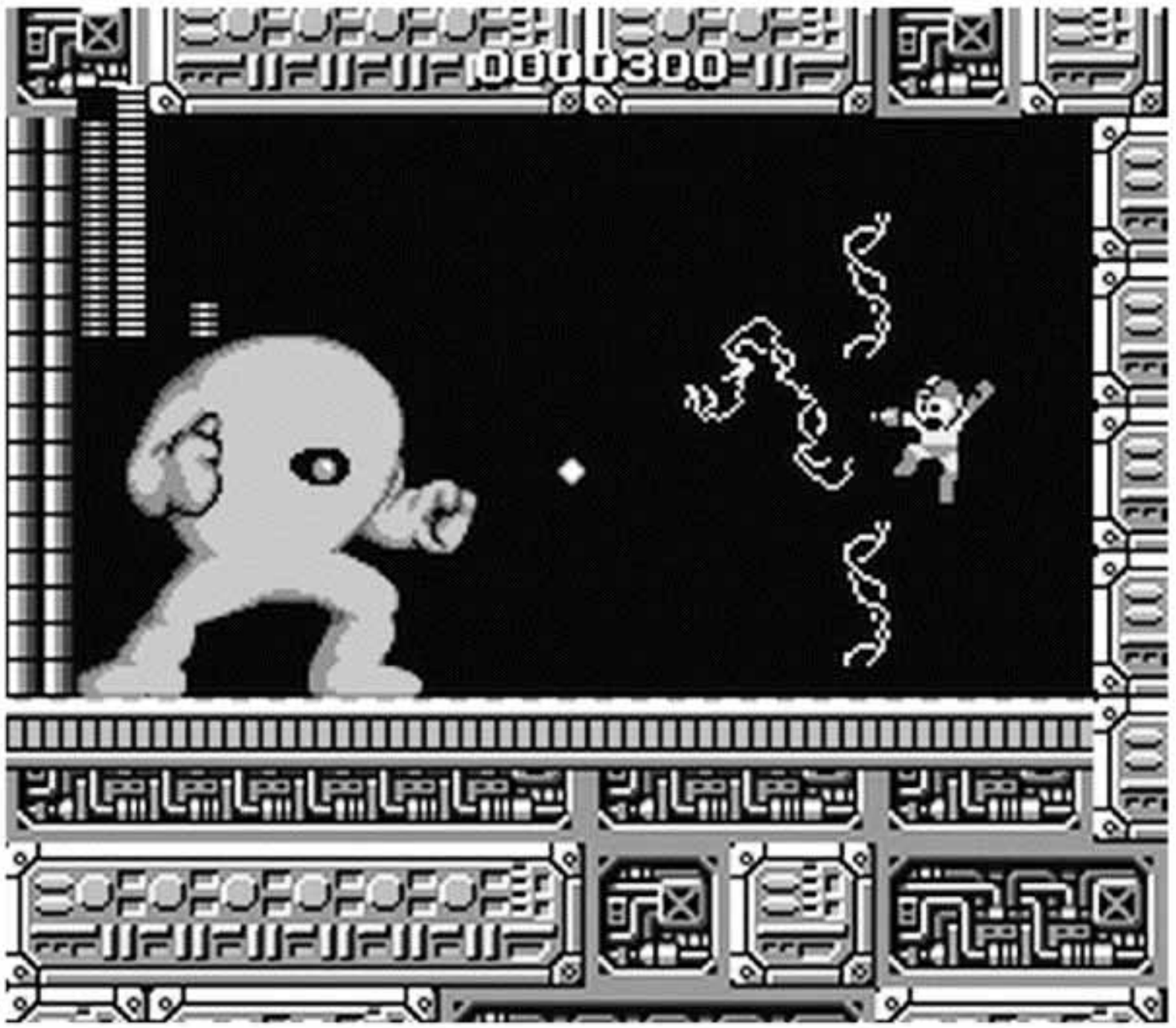


FIGURE 8.1 *Mega Man* screenshot. © Capcom Co., Ltd. Reprinted with permission.

You will note that the basic setup for the AI of this creature (in this case, an ant) is a finite-states machine (FSM) implemented as a `select` statement with various states.

LISTING 8.1 Sample LISP Source Code from an Enemy in the Side Scroller *Abuse*

```
(defun ant_ai ()
      (push_char 30 20)
      (if (or (eq (state) flinch_up) (eq (state) flinch_down))
      (progn (next_picture) T)
      (progn

      (select (aistate)
          (0   (set_state hanging)
               (if (eq hide_flag 0)
               (set_aistate 15)
```

```
      (set_aistate 16)))

(15 ;; hanging on the roof waiting for the main character
 (if (next_picture) T (set_state hanging))
 (if (if (eq (total_objects) 0);; no sensor, wait for guy
     (and (< (distx) 130) (< (y) (with_object (bg) (y))))
   (not (eq (with_object (get_object 0) (aistate)) 0)))
     (progn
        (set_state fall_start)
             (set_direction (toward))
        (set_aistate 1))))

(16 ;; hiding
 (set_state hiding)
 (if (if (eq (total_objects) 0);; no sensor, wait for guy
     (and (< (distx) 130) (< (y) (with_object (bg) (y))))
   (not (eq (with_object (get_object 0) (aistate)) 0)))
     (progn
        (set_state fall_start)
             (set_direction (toward))
        (set_aistate 1))))

(1 ;; falling down
 (set_state falling)
 (scream_check)
 (if (blocked_down (move 0 0 0))
     (progn
        (set_state landing)
        (play_sound ALAND_SND 127 (x) (y))
        (set_aistate 9))))

(9 ;; landing /turn around(gerneal finish animation state)
 (if (next_picture) T
   (if (try_move 0 2)
   (progn
     (set_gravity 1)
     (set_aistate 1))
     (progn (set_state stopped)
        (go_state 2)))))  ;; running

(2 ;; running
 (scream_check)
 (if (eq (random 20) 0) (setq need_to_dodge 1))
 (if (not (ant_dodge))
```

TeamLRN

```
(if (eq (facing) (toward))
(progn
  (next_picture)
  (if (and (eq (random 5) 0) (< (distx) 180)
                             (< (disty) 100)
       (can_hit_player))
      (progn
         (set_state weapon_fire)
         (set_aistate 8))  ;; fire at player
         (if (and (< (distx)100)(> (distx) 10)
              (eq (random 5) 0))
    (set_aistate 4)  ;; wait for pounce

        (if (and (> (distx) 140)
         (not_ant_congestion)
         (not (will_fall_if_jump)))
         (set_aistate 6)

       (if (> (direction) 0)
           (if (and (not_ant_congestion) (blocked_right
                                    (no_fall_move 1 0 0)))
           (set_direction -1))
         (if (and (not_ant_congestion) (blocked_left
                                   (no_fall_move -1 0 0)))
             (set_direction 1)))))))
     (progn
       (set_direction (toward))
       (set_state turn_around)
       (set_aistate 9)))))

(4 ;; wait for pounce
 (if (ant_dodge) T
   (progn
     (set_state pounce_wait)
     (move 0 0 0)
     (if (> (state_time) (alien_wait_time))
     (progn
        (play_sound ASLASH_SND 127 (x) (y))
        (set_state stopped)
        (go_state 6))))))

(6 ;; jump
 (setq need_to_dodge 0)
```

```
 (if (blocked_down (move (direction) -1 0))
     (progn
        (set_aistate 2))))

(8 ;; fire at player
 (if (ant_dodge) T
   (if (eq (state) fire_wait)
   (if (next_picture)
       T
     (progn
        (fire_at_player)
        (set_state stopped)
        (set_aistate 2)))
        (set_state fire_wait))))

(12 ;; jump to roof
 (setq need_to_dodge 0)
 (set_state jump_up)
 (set_yvel (+ (yvel) 1))
 (set_xacel 0)
 (let ((top (- (y) 31))
   (old_yvel (yvel))
   (new_top (+ (- (y) 31) (yvel))))
   (let ((y2 (car (cdr (see_dist (x) top (x) new_top)))))
     (try_move 0 (- y2 top) nil)
     (if (not (eq y2 new_top))
     (if (> old_yvel 0)
       (progn
          (set_state stopped)
          (set_aistate 2))
     (progn
        (set_state top_walk)
        (set_aistate 13)))))))

(13 ;; roof walking
 (scream_check)
 (if (or (and (< (y) (with_object (bg) (y)))
      (< (distx) 10) (eq (random 8) 0))
     (eq need_to_dodge 1))  ;; shooting at us, fall down
     (progn
        (set_gravity 1)
        (set_state run_jump)
        (go_state 6))
```

TeamLRN

```
(progn
  (if (not (eq (facing) (toward)))
       ;; run toward player
  (set_direction (- 0 (direction))))
  (if (and (< (distx) 120) (eq (random 5) 0))
  (progn
    (set_state ceil_fire)
    (go_state 14))
(let ((xspeed (if (> (direction) 0) (get_ability
                                        run_top_speed)
       (- 0 (get_ability run_top_speed)))))
  (if(and(can_see (x)(- (y) 31)(+(x) xspeed)(- (y) 31) nil)
      (not (can_see (+ (x) xspeed) (- (y) 31)
                 (+ (x) xspeed) (- (y) 32) nil)))
      (progn
         (set_x (+ (x) xspeed))
         (if (not (next_picture))
            (set_state top_walk)))
         (set_aistate 1)))))))


(14 ;; cieling shoot
 (if (next_picture)
     T
   (progn
      (fire_at_player)
      (set_state top_walk)
      (set_aistate 13))))

)))

T)
```

In 1996, *Mario64* came out, presenting us with the next chapter: the fully 3D platform game. *Mario64* took scrolling levels into the realm of a fully realized three-dimensional land, but somehow kept all the goodness of its earlier brothers. This game is still the blueprint by which modern platformers measure themselves and serves as a model of great gameplay, beautiful camera work, and a highly polished overall experience.

Today, platform games are mostly about three things: exploration (the need to figure out where things are hidden, and how to get there), puzzle solving (either through specific gameplay or through combining elements found in the world), and physical challenges (timed jumps, performing chains of specific moves, over-

coming a time limit, etc.). Game designers in this genre are continually pushing the envelope of new gameplay mechanics, new types of challenges, and new ways to make this genre fun and engaging.

COMMON AI ELEMENTS

Enemies

Most platform enemies are very simple, with basic behaviors, because enemies are usually considered little more than obstacles in the platform world. They complement the hardness of the exploration challenges (for example, by being placed in the exact location that an inexperienced player might jump to, or by forcing the player to then perform another immediate jump). In this way, placement of enemies becomes another level of tuning for designers because they find the setups that lead to the precise difficulty level they are striving for.

However, some enemies are more general, being either crafty or highly skilled (such as the little blue thieves in the *Golden Axe* games who are almost impossible to stop). In the *Oddworld* games, many of the enemies were actually invincible, at least to direct attack. You had to find the way to disable these enemies, by affecting the environment or another character, and thus indirectly removing the threat. *Oddworld* was almost an extended puzzle game, with each enemy being another puzzle that the player had to determine how to disarm.

But generally, platformers are more about physical challenges (jumping, climbing, etc), and so the enemies sometimes ride in the back seat. Many games have also used the concept of enemies that *are* platforms, where the player is walking on the backs of large enemies like stepping stones, but that doesn't mean the enemy has to like it. He can fight back, tip the player off, and so forth.

Boss Enemies

Modern platform games usually have large, scripted, end-of-level boss creatures. Most games use scripted patterns for the boss monsters (which the player will learn over time), and in addition, will usually force the player into performing some sort of advanced jumping challenge or other game mechanic exhibition (blasting away pieces of the floor, so that your available landing positions become less, or temporarily covering large portions of the floor with damaging fire, spikes, or explosions). Boss enemies are extremely important to the platform game experience, as in all games that use them. They provide a break from the regular gameplay mechanics and provide pacing, and their commonly large size and surprising abilities make for interesting game experiences.

TeamLRN

Cooperative Elements

Some games include a supportive character, such as the helper dog, Rush, that was added to the *Mega Man* games. This character is either under direct control of the user, or functions automatically, helping as needed. In the latter case, AI code must control this character, usually as secondary attacks, some form of powerup retrieval, or some combination move that augments the gameplay. Consequently, the AI is usually not overly complex for these game agents and is mostly reacting to what the player is doing. In some ways, you do not want an overly powerful helper because a helper that could do too much would eventually make the player feel less important. Most helpers are about 80% autonomous (meaning they are running a small script or what have you that is reacting to the player), and the rest of their use is in their response to some kind of "action" key initiated by the player. *Come here, pick me up,* or *go get that* are all examples of a controlled callable action that the player is allowed to use the helper for.

Camera

Once platform games made the switch to 3D, they faced the problem that has felled many games involving precise positioning and environmental challenges in 3D space: *where to place the camera to best see everything.* Nowadays, with more dynamic environments and faster gameplay, this problem is even more pronounced. Some games have used the higher graphical power of the more modern game consoles to try to remedy this by having environmental elements that are occluding visibility become transparent, so the player can see through them to the action. Although this does help to some degree, it distances the player from the game experience by making the player feel like an observer to the action, rather than the main character himself. Clever camera code, and a tight integration with the level itself, can be used to create a camera system that can give you good visibility, while maintaining connection with the character. Camera AI is usually created with a few different methods:

- *Algorithmically placing the camera behind the main character toward his direction of travel (or some other vector).* This leads to, at the very least, dependable camera movement, and with camera relative controls, allows the least amount of surprise movement by the human player (meaning, that the camera will not suddenly cut to a dramatically different angle to the player, and hence affect the direction the controls will move the player in). The problem with this simple system is that it is very hard to use it to account for things like special terrain features, dynamic enemy placement, special moves that might move the character very rapidly or in some strange direction, and so forth. In effect, an algorithmic solution helps with only one-half the problem. You need a good

general solution, but also a means of approaching all the special cases that a game might confront because of gameplay mechanics or level design.

- *Laying down tracks of level data for placement and orientation.* This method, usually used in combination with the first technique, involves the level designers placing a number of camera paths in the map. At a specific location within the map, the camera knows where to position itself and orient toward by pointing to this map data. This leads to a much greater use of environmentally affected camera angles, and can create dramatic camera shots that give the player a sense of being there. It can also help the user to determine the direction of play within a particularly large or open world. For instance, in your game, you might have a very deep pit with many platforms that a player would have to drop down onto. Using a camera system like this one, the camera could help the player to know the general direction of the next platform, by biasing the position of the camera as the player approached the edge of each stage.
- *A free camera mode.* Meaning a "first person" mode, in which the player has direct control of the orientation of the camera, looking out from the eyes of the main character. Most games include this mode because of the frustration of getting the other two modes to be all-inclusive. Even those games in which almost nowhere does the automatic camera break down, some developers give the player this option anyway, so that the player can pause occasionally and appreciate the game environment (or just to feel more in control).

USEFUL AI TECHNIQUES

Finite-State Machines (FSMs)

State machines are useful in platform games as well. These games have very straightforward enemies, with usually only a few behaviors exhibited by any one enemy (except bosses, perhaps, although boss enemies in platformers are usually *very* state- or script-based). Also, these behaviors are usually very crisp, meaning there is little gray area between them. The ghouls in *Maximo,* for example, are either walking very slowly in some random direction, or they see you, and charge directly toward you very quickly.

Messaging Systems

The puzzle-style nature of most platform games lends itself well to using event messages to notify enemies and environment elements about game-state change because the game would have to poll for an undisclosed period as the human figures things out, which is a wasteful way to do things. Instead, only after the hero has

TeamLRN

found the magic green button on top of the roof of the house and pressed it, is an event triggered so that the gate blocking the green cave will retract.

Scripted Systems

Because of the pattern nature of the boss enemies, not to mention some of the normal game enemies, scripting is a natural way to craft the AI for these elements. Scripting in platform games allows a very fine control to be exerted over the flow of a particular part of the game, say that of a boss encounter, or an in-game cinematic sequence that gives the player information. Some of the more complex platformers have an in-game help character that follows you around for the first level and shows you how to perform all the moves and special powers that the main character has at his disposal. Scripting would allow you to add all of this helper character's actions, as well as dialogue, and tie it into the control scheme of the game so that the helper will wait for you to practice the moves, explore on your own, or even ask questions and have the helper repeat part of the script.

Data-Driven Systems

The camera for 3D platformers sometimes becomes very complex, so camera paths must be constructed within the level editor for these games if a suitable algorithmic solution cannot be found. Designers can also do a lot of level tuning when they populate the levels with enemies, by knowing the patterns of movement for different types of creatures, as well as the effect these placements will have on the human traversing that section of the level. These games can become very data driven if enough forethought is put into the types of challenges the designer wants to incorporate, as well as the limits of the level editor and the control needed by the designers for level tuning.

EXAMPLES

Classic platform games like *Donkey Kong, Castlevania, Sonic the Hedgehog, Mario Bros.*, and *Metroid* are some of the big names in the platform game hall of fame. *Castlevania* was almost too hard. *Sonic* was almost too fast. Samus the main character from *Metroid,* was definitely too cool. All these games used state-based enemies, often singular-state enemies. Usually, these enemies would employ simple movement patterns (such as moving back and forth between two objects), or they would be "hiding" until you get near, and then they'd jump out at you. Many of these games used the concept that enemy contact hurts you, so enemies rarely had more to their attack strategy than ramming into you, although some did have simple projectiles.

The next generation of platform games brought us titles like *Mario64* (*the* 3D platformer, in which many of the techniques used by other companies were all but

invented by Nintendo's prime game designer Shigeru Miyamoto), *Spyro the Dragon,* and *Crash Bandicoot.* The jump to 3D provided new challenges because of the added complexity of moving within 3D worlds, but also brought a new evil: the bad camera system. The games continued to use most of the earlier styles of AI implementation, with patterned or scripted enemies, and slightly more complex level bosses. Sadly, during both the 2D and 3D eras of platform games, many platformers became showcases for cutesy new characters instead of gameplay. We were inundated with edgy, *slightly* bad attitude and somewhat cute animals off all kinds, trying to hawk games that were derivative at best. Lucky for us, the industry got over that little hurdle.

Today, platform games are as strong as ever, with increasingly devious puzzles, enemy AI, and level design. Games like *Ratchet and Clank, Jak and Daxter,* and *Super Mario Sunshine* continue to push the envelope. Some of these games still use simple FSM and scripted AI, but augment it when necessary with smarter opponents and clever sidekicks. The camera systems of these modern games, although still having problems, continue to get better, with heavily layered camera systems getting closer and closer to always pointing in the right direction, while maintaining and enhancing the overall feel of the game.

AREAS THAT NEED IMPROVEMENT

Camera Work

As good as some games' cameras are, very few games indeed have had total success in this area, partly because people have different tastes for the camera and partly because it is a very hard problem, especially if you intend to have an algorithmic camera. In some ways, the camera needs to somehow *anticipate* the movements of the player (or even the intent to move, which is even more impossible) and move the camera to show the player what is in that direction. This problem is also very game specific. Characters that can jump a long way need to see farther out; characters engaged in heavy combat need to have bearings so that they can land hits on a nearby enemy, who may be attacking back with much better accuracy. In the future, we may even get a specialized peripheral, such as the microphone headset being used in some games today with voice recognition, except that it would track certain movements to help with the camera.

Help Systems

Some platformers are simply too hard for some people, or a given location puzzle can stump a player for far too long. This kind of slowdown in the flow of the game can ruin the experience very quickly. If the game could discern that the human is stuck, and needs help, it could possibly offer hints to get the player moving again.

TeamLRN

This could be an option that the player could turn on or off, so that diehard players who want to find everything themselves wouldn't have the surprise ruined for them. But casual gamers might appreciate the helping hand after spending four hours trying futilely to make an impossible jump because they don't realize that they need to walk around the corner and use the invisible catapult to get across the chasm. This is a specialized system, but because of the goal-oriented nature of these games, it would be possible to have a help manager that could be goal based. Thus, each small section of gameplay could keep track of the attempts being made by the human to solve that atomic portion of the game, and note failures. In addition, puzzles of the same type later in the game could respond more quickly because the game passes on the information that the player had difficulty with similar earlier challenges. Again, this kind of system would have to be a difficulty setting (which could be set on or off, or some level of help), but could be turned on by default in the first "training" level, or whatever system your game will use.

SUMMARY

Platform games have gone from very simple affairs, to grandiose living worlds within 10 years. Even with this vast change in the landscape, many companies have managed to keep the fun formula intact, with careful adherence to the genre's strengths and by minimizing the effect on all the additional technology to the gameplay mechanics with clever controls and good AI systems.

- Most enemies in platform games are very simple, with patterned or simple movements, to facilitate the fact that killing enemies is secondary to the physical challenges one of the game.
- Boss enemies are generally much larger, and more powerful, but are generally still scripted. The trick is to discover the pattern, then use it against the creature to beat him.
- Cooperative elements in platform games are more like semi-intelligent powerups, in that they usually just augment the main character.
- The camera system, if the game is 3D, is vital to the overall quality level of the game because seeing the right thing at the right time is complicated heavily by the much bigger and more open worlds. Techniques involving algorithmic solutions, camera tracks laid down in an editor, and a free look camera are typical methods of approaching the problem.
- FSMs are used heavily in these games because of the simple nature of the AI enemies and such.
- Messaging systems make sense in this genre because of the event-driven nature of the puzzles and interactions.

- Scripting will aid in the creation of the patterned movements of enemies, and give in-game cinematic events a means by which to tailor custom animation and audio sequences.
- Camera work needs to continue to strive toward giving the player a system by which to get the best angle on things, without sacrificing control.
- Help systems could be implemented, to give hints (or outright aid) to players who are stuck on a puzzle or physical challenge, if they so desire it. This will help frustrated players, but does require a significant amount of AI to achieve.

TeamLRN

9 Shooter Games

The term *shooter games* refers to the fairly open genre encompassing classic shooters (static and horizontal or vertical scrolling) and the modern variation, which is played using a light gun. Most of these types of games use very simple AI or patterns for their enemies, with the trick to any given game level being finding this pattern (or AI weak point) and exploiting it to get easily to the next level or enemy. Some shooters throw enough enemies at you that even if you know the pattern, survival is still questionable. Simple control schemes are generally the law of the land; people usually can't look down to find a button in the middle of a sea of enemy bullets. A notable exception was *Defender II: Stargate*, a truly classic horizontal shooter, that had no less than seven controls: the up/down joystick, thrust, reverse (to turn around), a hyperspace button (randomly teleported you), a shoot button, an "inviso" button (which was an invincible shield of sorts), and a smartbomb button (which killed all on screen enemies). The game was devilishly hard and made even more so by the nature of the control scheme. But it was a gigantic hit and continues to be a classic favorite. Again, the rule seems to be, if the game is good enough, people will take the time to learn how to play it well.

Shooters originated in the arcades, but didn't translate well to the personal computer world, although they have made a decent showing on the various home consoles. Shooters usually involve a ship our some kind of character who is facing monstrous waves of enemies that come at him in patterns. The player kills as many enemies as possible while avoiding (or in some light games, ducking behind cover) the enemy's incoming shots. Along the way, you pick up powerups and fight bosses, which are usually massive affairs in these games.

Interestingly, numerous independently made shooters can be found for download on the Web. Many people get their start game programming by home brewing a 2D shooter of some sort. This is the kind of game that one person can still make by himself (possibly with some help on the art) and do a good job. Listing 9.1 shows some of the enemy AI code from the open-source game *Wing*, which the author (Adam Hiatt) jokingly mentions is a recursive acronym that stands for "Wing Is

TeamLRN

Not Galaga." Notice that Adam's game uses a simple implementation of a finite-state–based AI system, where he has various behaviors written (`Attack_1` through `Attack_5`), and the enemies cycle between them in patterns.

LISTING 9.1 Sample AI Code from *Wing,* by Adam Hiatt. Licensed under the GNU.

```
//=============================================================
==
void EnemyTYPE :: UpdateAI ( int plane_x, int plane_y )
{
    EnemyNodeTYPE * scan = enemy_list;
   for (; scan != NULL; scan = scan -> next)
   {
      if ( scan -> health <= 0 && scan->explode_stage ==
                                        ENEMY_EXPLODE_STAGES - 1 )
        DeleteNode ( scan );
      else
      {
         if ( scan -> attacking )
         {
            if ( (scan -> xpos >= plane_x && scan -> xpos < plane_x
                  + PLANE_WIDTH) ||
(scan -> xpos + EnemyWidths [scan->TypeOfEnemy] >=
                  plane_x && scan -> xpos + EnemyWidths [scan->
                  TypeOfEnemy] < plane_x + PLANE_WIDTH))
            {
                if(timer - scan -> TimeOfLastFired > BULLET_PAUSE &&
                    (plane_y > scan -> ypos + EnemyHeights [scan->
                    TypeOfEnemy] && timer- scan->TimeOfLastFired >=
                    BULLET_PAUSE))
                {
                    scan -> TimeOfLastFired = timer;
                    enemy_bullets.Fire (scan -> xpos, scan->ypos,
                                XBulletVelocities [scan->weapon],
                                      -(YBulletVelocities [scan->
                                      weapon]), scan->weapon );
                }
          }

       switch ( scan->state )
      {
            case ATTACKING_1 : Attack_1 ( scan );
                               break;
            case ATTACKING_2 : Attack_2 ( scan );
                               break;
```

```
        case ATTACKING_3 : Attack_3 ( scan,plane_x );
                                   break;
        case ATTACKING_4 : Attack_4 ( scan );
                                   break;
        case ATTACKING_5 : Attack_5 ( scan );
                                   break;
        case ATTACKING_6 : Attack_5 ( scan );
                                   break;
        default          :         break;
}
scan -> state_stage ++;
if ( (scan -> ypos < -80 || scan -> ypos>SCREEN_HEIGHT) ||
     (scan -> xpos + EnemyWidths[scan->TypeOfEnemy] < 0 ||
      scan -> xpos > SCREEN_WIDTH ) )
        {
            scan -> attacking = false;
            num_enemies_attacking –;
        }
      }
    }
  }
}
//=============================================================================
void EnemyTYPE :: Attack_1 ( EnemyNodeTYPE * enemy )
{
    if ((enemy->xpos >= SCREEN_WIDTH - 75 && enemy->dx > 0 )||
       (enemy->xpos <= 5 && enemy->dx < 0))
        enemy->dx = -(enemy->dx) ;
    else if ( enemy -> state_stage % 20 == 0 )
    {
       if( enemy->xpos < SCREEN_WIDTH / 2 )
{
   if ( enemy -> xpos <= 160 && enemy -> dx < 0 )
       enemy->dx /= 2;
   else if ( enemy ->dx < 8 && enemy ->dx > -8 )
       enemy->dx *= 2;
   if ( enemy->dx == 0 )
       enemy-> dx = 1;
}
else
{
    if ( enemy-> xpos >= SCREEN_WIDTH-160 && enemy-> dx > 0 )
        enemy->dx /= 2;
    else if ( enemy ->dx < 8 && enemy ->dx > -8 )
        enemy->dx *= 2;
```

TeamLRN

```
                if ( enemy->dx == 0 )
                    enemy-> dx = 1;
            }
        }
        enemy->ypos += enemy->dy;
        enemy->xpos += enemy->dx;
}
//=============================================================
void EnemyTYPE :: Attack_2 ( EnemyNodeTYPE * enemy )
{
        if ( enemy -> ypos == INIT_ENEMY_Y )
       {
        enemy -> dy = 4;
            if ( enemy -> xpos < SCREEN_WIDTH / 2 )
            enemy -> dx = 3;
            else
                enemy -> dx = -3;
       }

       if ( (enemy -> ypos) % 160 == 0)
            enemy->dx = -(enemy->dx);

        enemy->ypos += enemy->dy;
        enemy->xpos += enemy->dx;
}
//=============================================================
void EnemyTYPE :: Attack_3 ( EnemyNodeTYPE * enemy, int plane_x )
{
        if ( enemy -> ypos == INIT_ENEMY_Y )
        {
            enemy -> dy = 6;
            if ( enemy -> xpos < SCREEN_WIDTH / 2 )
                enemy -> dx = 3;
            else
                enemy -> dx = -3;
        }
        else if ( enemy -> ypos > 175 )
        {
            if ( enemy -> dy == 6)
            {
                enemy -> dy = 4;
                if ( enemy -> xpos > plane_x  )
                    enemy -> dx = -10;
                else
                    enemy -> dx = 10;
```

```
        }
        if ( enemy -> state_stage % 20 == 0 )
            enemy -> dx /= 2;
    }
    enemy->ypos += enemy->dy;
    enemy->xpos += enemy->dx;
}
//=============================================================
void EnemyTYPE :: Attack_4  ( EnemyNodeTYPE * enemy )
{
    if ( enemy -> ypos == INIT_ENEMY_Y )
    {
        enemy -> dy = 4;
        if ( enemy -> xpos < SCREEN_WIDTH / 2 )
            enemy -> dx = 3;
        else
            enemy -> dx = -3;
    }

    if ( (enemy -> ypos) % 160 == 0)
        enemy->dx = -(enemy->dx);

    if ( enemy-> ypos > 0 )
    {
        if ( enemy -> state_stage % 40 == 0 )
        {
            enemy-> dx = rand() % 13;
            enemy-> dy = rand () %13;
        }

        if ( enemy->dx > 7 )
            enemy->dx = -rand ()%7;
        if ( enemy->dy > 7 )
            enemy->dy = -rand ()%7;
    }
    else
        enemy-> dy = 4 ;

    enemy->ypos += enemy->dy;
    enemy->xpos += enemy->dx;

}
//=============================================================
void EnemyTYPE :: Attack_5  ( EnemyNodeTYPE * enemy )
{
```

TeamLRN

```
if ( enemy -> ypos == INIT_ENEMY_Y )
{
    enemy -> dy = 4;
    if ( enemy -> xpos < SCREEN_WIDTH / 2 )
        enemy -> dx = 3;
    else
        enemy -> dx = -3;
}

if ( (enemy -> ypos) % 160 == 0)
    enemy->dx = -(enemy->dx);

if ( enemy-> ypos > 0 )
{
    if ( enemy -> state_stage % 30 == 0 )
    {
        enemy-> dx = rand() % 13;
        enemy-> dy = rand () %13;
    }

    if ( enemy->dx > 6 )
        enemy->dx = -rand ()%6;
    if ( enemy->dy > 6 )
        enemy->dy = -rand ()%6;
}
else
    enemy-> dy = 3 ;

if ( enemy->xpos + enemy->dx < 0 || enemy->xpos + enemy->dx +
    EnemyWidths [enemy->TypeOfEnemy] > SCREEN_WIDTH )
    enemy->dx = -(enemy->dx);

    enemy->xpos += enemy->dx;
    enemy->ypos += enemy->dy;
}
```

COMMON AI ELEMENTS

Enemies

Shooters enemies are usually distinctly patterned, so that you can successively learn more and more of the pattern and get farther into the game. As such, the AI for these games is not usually intelligent at all. The light gun games are the same basic

mechanic: a pattern of guys will pop out from behind things, and you have to shoot them before they shoot you.

However, some games do stray from this basic formula and make AI enemies that readily seek the player or use almost first-person shooter/third-person shooter (FTPS) "bot-like" behavior, using fairly decent "smarts" to counter the human. However, even games with advanced enemies generally keep the player on some kind of *rails* (a set path through the map, so-named because it feels like you're in a slow train car riding along on set rails), which keep you constrained and allow the "smart" opponent to duck off screen to escape your attacks. The concept of rails is used in both conventional shooters and light gun games, mainly to control pacing of the game (these rails were originally created in arcade games to limit players' progress to a certain rate during gameplay, while giving the player a slowly changing view of the game world).

Other games use large, moving creatures (such as the dinosaurs in *Jurassic Park: The Lost World*) that occasionally display vulnerable spots that you shoot out. This behavior is basically the same—targets jumping out at you—but the increased on screen movement of this system does add a lot to the look and feel of the game.

Boss Enemies

Just like in role-playing games (RPGs), bosses in shooter games are frequently considered a treat that you find at the end of each level. Shooters usually go overboard on the boss enemies because of the fairly repetitive gameplay inherent in the genre. Good boss creations can sometimes save, or even make, the experience of an average shooter much better and more memorable. As such, the AI system for the bosses is very important and should be flexible enough to encompass any sort of specialized needs that each boss in the game will require. One thing to note is that many games fully script the movement and firing times of their boss enemies. The bosses of scrolling shooters are usually huge, horribly beweaponed monoliths, spewing bullets of every shape and size in all directions. They generally attack in waves (which translate to *states* as far as implementation is concerned), with phases of heavy attack, followed by a brief respite, followed by some blindingly large gun blast, and then it all repeats again. Bosses are typically impervious to all damage, except for key locations (typically colored red, or glowing in some way), that may or may not *also* be state based (in that they are sometimes covered by a protective shell of some sort).

During hectic boss battles, many scrolling shooters had what hardcore players referred to as *safe zones,* which were very specific locations on the screen where you could sit and never be hit by an enemy bullet, but still get an occasional shot at the boss. Some games embraced this, making the boss very difficult and counting on the human to find the safe spot, whereas other games went the other way, adding an occasional "homing" shot to ferret out nonmoving players.

TeamLRN

Cooperative Elements

Some shooter games include an AI controlled drone or some sort of helper object that is either an integral part of the gameplay mechanics (like the TOZ in *Gaires*), or becomes a powerup or weapon that helps the player once found (the "Option" powerup in the *Gradius* games). These elements are usually pretty simple, but this determination is completely up to the game designer. You don't want a drone doing too much of the work, however.

USEFUL AI TECHNIQUES

Finite-State Machines (FSMs)

State machines continue their usefulness in this genre, mostly because of the simple, straightforward nature of the AI in most of these games. The organization of the games themselves (level based), with an easy start period, followed by a buildup, and then a boss, also lends itself to a state-based architecture. Many of the enemies in this genre only had one state, such as the main creature in *Centipede,* which used a simple rule for its AI. It would move forward, until it hits a mushroom. It would then move down and reverse direction. The only other behavior it had was that it sped up a lot if there was only one segment of the creature left. A very simple rule, and the layout of the level provided the variance in the gameplay. In modern AI programming, this is called *emergent* behavior. The elements of *Centipede* combine, and the final behavior emerges from the interactions. Back then it was just called good game design.

Scripted Systems

The boss enemies in shooters are usually of the immobile-behemoth-with-one-or-two-well-guarded-vulnerable-spots type, but even if they are more mobile, they are most likely just scripted events. Boss monsters rarely react to the human's actions, although they might slowly head in your direction, or jump on top of you, or something. Rather, they tend to move in patterns while spitting out waves of bullets and other things to harm the player. These simple chains of behavior are textbook uses for a simple scripting system.

Adding the ability to randomly branch within a script will give a degree of variety to your pattern scripts (because each chunk will be executed in some random order). Scripts also make it very easy to tag specific enemy spawns with difficulty level information (so that more enemies will attack the player in harder games, or from different angles and locations), so that the same script can be used for easy, normal, and hard levels of difficulty.

Data-Driven Systems

The general enemy AI for shooters (if following the patterned waves paradigm) is very open for a full data-driven structure. The basic types of enemy movement and firing patterns could be defined using code, and then a designer (or whomever) could quite easily set up a database table of when and where these patterns would appear in the levels, or they could actually be placed into some form of level editor that would then generate these appearance tables. In this way, the designer could tweak and tune the enemy content of his levels quickly and easily, without programmer help. Of course, new patterns required programmer intervention. But even this could be set up in an editor if need be, by providing the designer even more basic building blocks to construct behavior patterns out of.

EXCEPTIONS

Zanac, an 8-bit Nintendo Entertainment System (NES) game from 1986, claimed to have "automatic level of difficulty AI code," which would take into account the human's attack patterns and skill level. In reality, however, it was just checking a few stats (like your rate of fire, your hit percentage, and how long you'd been alive) and was then adjusting the number, speed, and aggression of enemies. If you went too long, killed every ship, and used a turbo button enhanced controller, it would take this system about ten minutes of game playing to be at the point of filling almost the entire screen with bullets. This was a great concept that made the game's difficulty scale adjust with the ability of the player, right? Nope. The human could dupe it by not killing everybody, missing shots, and occasionally dying on purpose. All of which brings up a big failing of games that try this method of difficulty scaling: you must consider the performance of the human, and you have to filter malicious or odd behavior, so that the system can't be fooled into helping the AI defeat itself.

EXAMPLES

Shooters were some of the very first true video games. Sure, the *Pong* types were running the roost for a few years. But then came 1978 and *Space Invaders,* what some consider to the be first true video game—complete with a score field, lives, and enemies that crept ever closer, while firing. Over the years, the controls have grown steadily more involved, the enemy patterns have grown more complex, and the powerups have grown steadily more elaborate and powerful.

Other early games like *Gradius, 1943, Raiden,* and *R-Type* further defined the genre: you, versus an appalling number of enemies, and they only stop coming so that the truly huge end boss can slip in and throw some death your way.

TeamLRN

Along the way, you can pick up numerous powerups, which will turn your simple ship into a bullet-producing factory. These games used patterned movement for their enemies. The advancing waves of enemy craft would move in back and forth patterns, various serpentine or circular shapes, or combination lines like a football play: move straight across to the left until you're lined up with the player, then double your speed and charge him.

In later years, the popularity of shooters started to wane, and along came the light gun game. Games like *Duck Hunt, Wild Gunman* (which even made its way into the second *Back to the Future* movie), *House of the Dead, Time Crisis,* and *Point Blank* (see the screenshot in Figure 9.1) are all great examples of this variant. These games were functionally just like their predecessors, but with a different input medium. Most still require you to dodge enemy fire in some fashion, by requiring your on-screen persona to duck behind cover, or to have the player shoot and move a character around (like *Cabal*). Most just required you to shoot first. Almost all of them include powerups that will give you more powerful weapons or more health and the like.

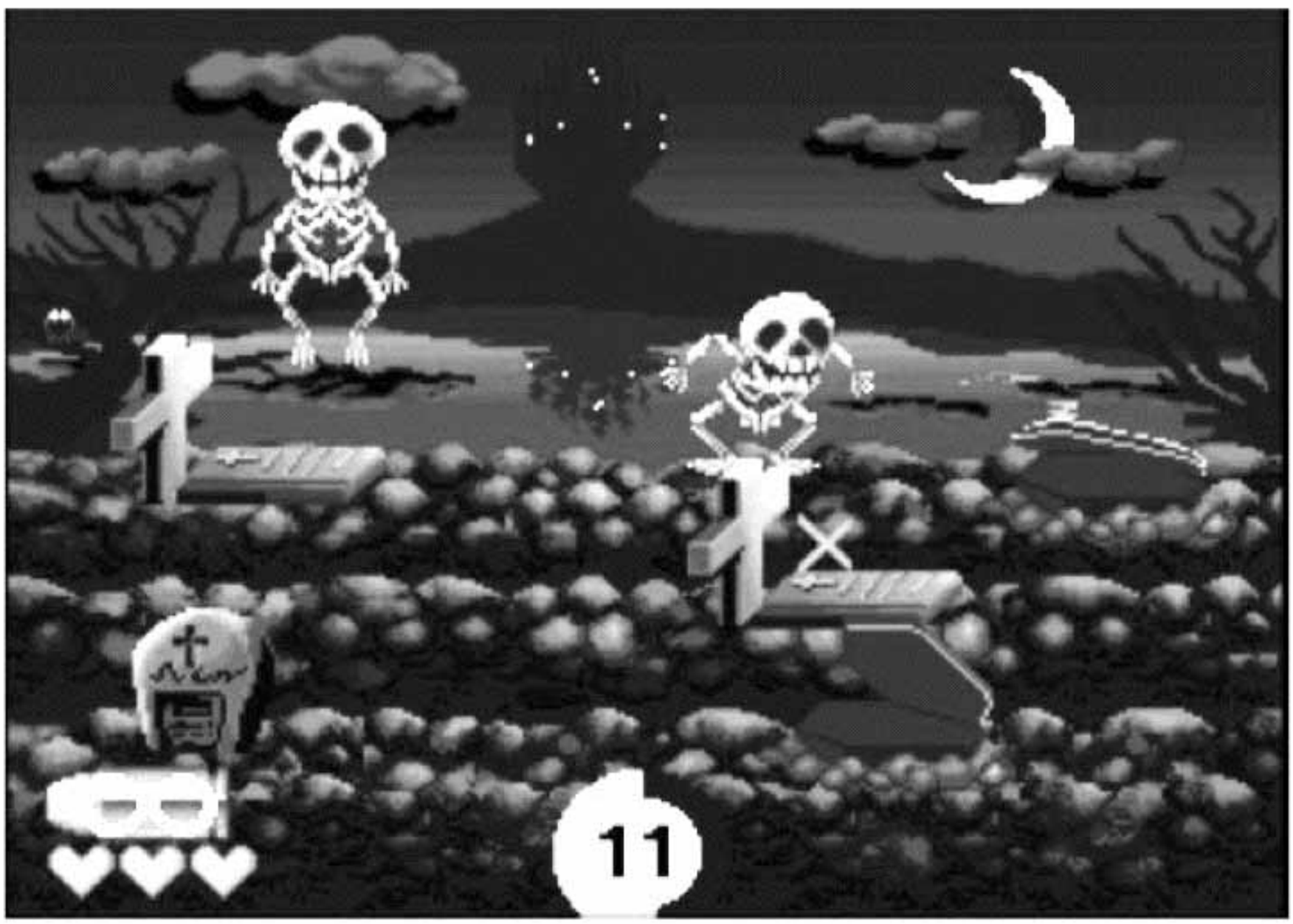


FIGURE 9.1 *Point Blank* screenshot. POINT BLANK® © 1994 Namco Ltd., All rights reserved. Courtesy of Namco Holding Corp.

Some shooter games in the arcade arena have tried to get some additional gameplay out of the genre by using strange control methods. *Robotron* and *Smash TV* used two joysticks, so you could move in one direction and shoot in another. *Cabal* and *Blood Bros* used a trackball that controlled your weapon's aim and that of a third-person character at the bottom of the screen. You had to aim while dodging the enemy fire directed at this character. Light gun games follow this same trend, with games that use different guns (such as automatic weapons, large rifles, pistols, etc.), or specialty guns (such as *Silent Scope,* which used a small LCD screen to simulate a sniper scope; or even *Brave Firefighters,* which puts you in control of a fire hose that you use to put out fires as they appear in the game).

AREAS THAT NEED IMPROVEMENT

Infusion of Actual AI

These games have fallen from grace as of late, probably because the old methods of pattern recognition and finding boss vulnerabilities have been done so many times that the concept is wearing thin. The light gun variant brought about a temporary return to these kinds of games, but eventually this small gameplay addition will be tired as well. Possibly, however, the gameplay could remain, but enemies with actual AI routines could be written. Scrolling shooters with this type of AI would almost be more like FTPS deathmatches, with the essential shooter gameplay mechanic and the bot opponents of the FTPS games. This might be the way to continue the dynasty of shooter-style games on the PC, by making a shooter deathmatch game, with online play and (because of the simplified 2D playing field) possibly many more simultaneous players.

SUMMARY

Shooter games are an old genre and are starting to seem stale because of the lack of innovation in gameplay and content. The light gun variation gave the genre additional fuel for a while, but the shooter game needs something new to continue to be a viable genre.

- Enemies in shooter games are patterned; the object is to figure out the pattern to get further into the game.
- Boss enemies are considered a treat and are very important elements of the shooter genre.

TeamLRN

- Cooperative elements are usually advanced powerups that involve additional gameplay techniques.
- FSMs or data-driven AI are usually the methods used in shooters. The simple nature of the AI controlled enemies, coupled with the fact that each level of a shooter is usually one long scripted pattern of appearing enemies, lends well to these two approaches.
- Either more complex FSMs, or a scripting system, might be useful for the larger boss enemies.

An infusion of actual AI techniques could possibly liven up this genre; a possible direction might be creating AI-controlled bots capable of fighting the player in a deathmatch-style mode of play, except within a shooter gameplay world.

10 Sports Games

Sports games have been with us from the very beginning: technically, *Pong* was a tennis game. The combination of instantly recognizable gameplay (everybody knows how to play your game!) combined with head-to-head action gives sports games a mass appeal that many other genres can only dream of. Coupled with the sea of rabid fans that buy perennial titles in multiple sports, the genre has become *the* moneymaking enterprise for companies that can capture the minds of sports gamers.

AI has become increasingly important in sports games. Early sports games were almost like action games, in that you learned the patterns exhibited by the other team and exploited them to win the game. Remember back to the LED football games, where you could score a touchdown easily by merely steering your red dot around the "defenders" very quickly and without stopping.

Today's sports gamers want the computer opponents to play like they do in real life, with intelligence, quickness, and a modicum of style. Games where the AI opponents merely shoot more accurately, or are "cheating" the stats of the opponents, are quickly called out for their unfair number juggling and are just as quickly taken back to the store.

Most competitive sports games fall into two basic categories:

- *Fluid gameplay sports.* These are sports like soccer, hockey, or basketball, where the game is quick, very dynamic, and continues for long periods with few or no stops. The nature of these games' constantly changing playfield conditions mean that even the simplest strategies need to be watched closely, to determine when a given *play* (a series of coordinated movements designed to score on the other team) isn't working, and recover gracefully by responding to the next set of game conditions. State-based AI tends to break down in these types of games because so many "states" are connected to other states that you end up with a spider web instead of a nice flow diagram. State hierarchies help with this

TeamLRN

problem, but the structure of working hierarchies tends to be anything but balanced.

- *Resetting gameplay sports.* These are games that stop and reset after a set event or time, such as football and baseball. The AI team in this style of game gains the benefit of being able to start from scratch fairly frequently, so the organization of the AI system can be designed with this in mind. This type of game lends itself much better to a state-based system because the sport itself is divided nicely into distinct game flow states.

One benefit of working on the AI engine for a sports title is that the game is usually fully designed before production starts. At least, the basic game you are trying to model is. If you're making a basketball derivative that uses robots and weapons, you're somewhat on your own. But a straight sports simulation has the advantage of a vast amount of information about how to play a successful game, with years of research and player statistics to back it up.

This strength is also a weakness. Everywhere you look, there are sports people. People who eat, drink, and breathe these games. People who know all the stats, follow their teams, and are very passionate about the game and the players. These are the kinds of people that buy sports games in the first place. The primary audience of your game is armed with this vast array of intimate knowledge of the sport, so it really puts the pressure on the developer. If you are making a pure simulation, you had better do it well. Someone who plays your game is going to know if the behavior he sees come out of a player would never happen in real life. Some of the players that your game might be trying to model are celebrities, and their actions and performance level is a signature that people either recognize being correctly represented by your system, or not.

COMMON AI ELEMENTS

Coach- or Team-Level AI

Consider coach- or team-level AI the *strategic* AI found in real-time strategy (RTS) or chess games. High-level AI makes decisions such as which play to call, or to substitute a player because he's in foul trouble and you want to save him for the last quarter. Without this level to a sports game AI system, the gameplay of the team can seem random, and without a purpose. Which of course, is exactly the case.

The team-level layer encompasses whole team-level decisions, but might also handle slightly smaller tasks that still involve more than one player (in a coordi-

nating fashion), such as a handoff in football or a player setting a pick for the ball handler in basketball. Usually, this level in the system uses some kind of shared data area (such as a blackboard system, or a team singleton class) that encapsulates the workings of this level, as well as provides a central place for the various other game elements to look when they need access to the team decisions.

A common mistake when coding this section of a sports game AI system is to not break down the tasks or use any kind of attribute data at this level. Most sports games make almost constant use of attributes when working at the player level (so that some hit the ball better than others, or are much faster), but this same type of thinking should be used when coding the team level. Using *team*-level attributes and overall goals, the same system can also simulate the various ways that particular teams play the game, which is particularly important in games where the coach is one of the more important elements, for example, college basketball. The players are good but inexperienced, so the coaches are calling almost all the plays and strategies, and two college teams might play wildly different types of games, even though the players on each team have similar skill levels.

Player-Level AI

At the player level, AI decisions concern the more personal, tactical behaviors that involve *just* the player: making a quick move to try and get open, leading off from first base, or performing a juke move against his defender to perform the larger goal given to him by the team AI level. The decisions and behaviors coming out of this layer would also heavily take into account the personal attributes for the player, as a reflection of his real-life counterpart (if any). By perturbing the behavior of the AI with real statistics, the human player will feel like he's playing with a character commensurate with the skill level of the real sports player. In this way, the AI of sports games also includes a large simulation element, in that you don't want everybody to be superheroes. Instead, guys who are bad passers should actually miss more, and poor defenders should break down and allow more offensive players to perform well.

The player level of the AI is actually more like two systems: the tactical decision-making part and animation selection, once the specific behavior has been assigned (see Chapter 23, "Distributed AI Design," for more on this). As an example, let's look the thought process behind trying to get open for a pass in football.

- The decision-making system decides that it wants to get open. So, the type of juke move to play (based on attributes, personal preference, and defensive match up) and the direction of movement (calculated because of proximity

TeamLRN

to other players and court boundaries, as well as court position in general) are determined.

- The animation selection process would then take this behavior data (direction and type information) and use it to determine the exact animation that the player will use to juke. Other factors that the animation layer will account for: the *type* of player (big, small, fast, showy, or some signature move), the speed of the player, the direction change (small changes might just rotate the player, bigger changes necessitate turnaround-type transition moves), some random factor so that the same animation doesn't play all the time, and many other factors, depending on the behavior.

Animation selection in sports games can sometimes become a secondary step of almost every action the player does. Most sports titles today use motion-captured animation for most moves in the game. Motion capture provides the signature moves of the stars, and shows the richness of secondary body movement that is usually only caught with motion-capture techniques because it is very hard to hand animate well. For some moves (such as football end zone dances or basketball dunks), players demand a huge variety of animations because they become the in-game taunts that allow you to rub in a particularly good play against your opponent. With this flood of available animations for a given behavior, systems must be put in place that can accurately pick the most contextually correct animation from large numbers of available animations using all manner of conditions. Generalized animation selection techniques (such as table-based systems) can be used to describe the links between the attribute data (and any other determinants) and the various animations for each action, which vastly improves the overall organization of your game and limits duplicate code by data driving this process.

Animation selection is usually not considered purely part of the AI system because the human player requires this same functionality to perform game behaviors. Even so, the process does need some level of intelligence because it is generally a highly context-sensitive determination (meaning, a unique process on a behavior-by-behavior basis); general approaches can quickly make your game look bland or just plain inappropriate.

Pathfinding

Finding good movement paths during the frenzy of a sports game can be truly frightening. Sure, the number of characters visibly on screen is limited, and the environment is usually free of static obstacles (although not always, you do have a

large net in hockey and soccer), but the dynamic obstacles (the other players and possibly a referee) are in almost-constant motion, making traditional path planning too slow and cumbersome. Lightweight, CPU-optimized methods must be used to make players move around each other as they do in the game.

Navigation in most sports titles also requires game-specific information to be considered when choosing paths. For example, in basketball, if your team is on offense, don't run right in front of the ball holder if you can help it. Even though you have technically avoided him, you have also cut off his movement and probably even caused a bit of a traffic jam right in front of him. This is not desirable. In football, which has even more rules along this line, finding good paths (or closing them) is actually a major part of the game.

Camera

The camera system for a modern sports game usually has two very conflicting goals: to show the action in the best possible way to facilitate good gameplay, and to look like TV broadcast sports games. These two goals focus the kinds of camera angles, cuts, and movement styles that can be used with the game, while still being playable. The balance of these two goals can only be determined by the design of the specific game. Are you shooting for the experience of "being the player"? Then you could probably experiment with different camera angles that are almost first person or heavily skewed toward a certain player's perspective. Or are you trying to get the human to feel like he's "at the game"? Then you'll want to expand your camera focus, giving the human a wider, whole court viewpoint on the action. Other camera styles that might be analogous to game design types include "be the coach," "watch the game on TV" (a very popular choice), "old school" (overhead, almost two-dimensional [2D] view used by many older games), and so on. See Figure 10.1 for two examples of these styles in use.

Miscellaneous Elements

Miscellaneous elements include things like cheerleaders, mascots, sideline coaches, the crowd, and everything else that makes up the side characters during sports games. Although they usually use very simple AI, these elements can really add up to making your game look much more real by supplying the player with elements that are alive in the world, regardless of his direct interaction.

TeamLRN

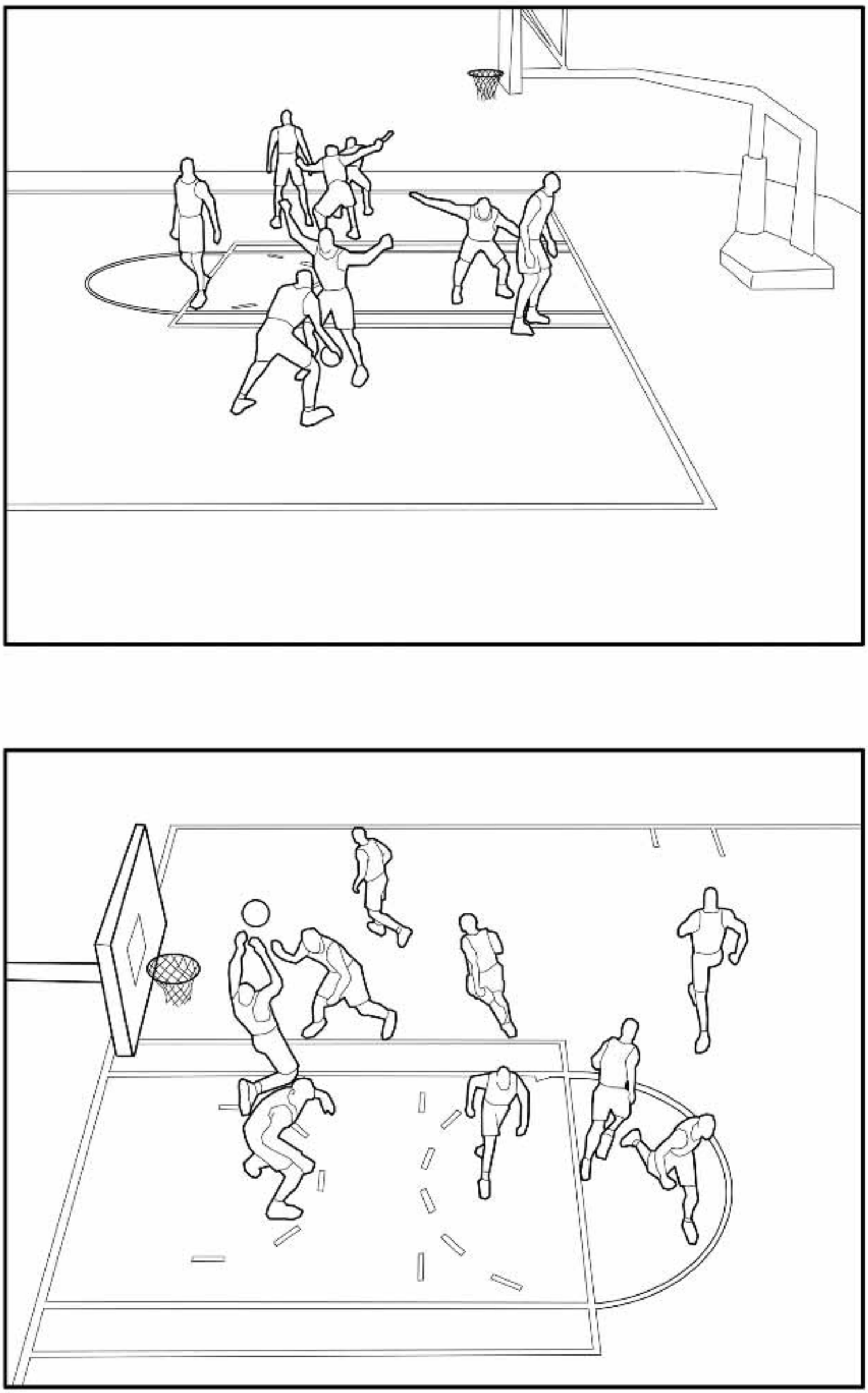


FIGURE 10.1 Different camera styles used in sports games can affect gameplay.

USEFUL AI TECHNIQUES

Finite-State Machines (FSMs) and Fuzzy-State Machines (FuSMs)

Games that fall into the "resetting gameplay" types are much easier to fit into a purely state-based AI model than are their more dynamic brothers. However, all games follow a set game flow (even basketball has tip-off, inbound, gameplay, and freethrow states), and the structure of AI within this game flow level is state based. But inside certain states within this game flow, the decisions the coaches and players must make is anything but clear-cut. Indeed, fuzzy decisions must be made at almost every level of sports games, and FuSMs can be used to provide this type of cloudy decision making.

Another way is to use a level of fuzziness at the perception level in your sports game. The states themselves would remain somewhat crisp, but the activations for each state would get a little blurry. So, a state variable that referred to whether or not you have an open look to take a slap shot would require a bit of fuzziness in its calculation (using a reaction time, a value hysteresis, and taking into account some player-level attributes, instead of just shooting a ray from the puck to the net and declaring it clear of obstacles), so that the crisp "Shoot the puck" state would therefore only be activated under this more fuzzy determination.

Listing 10.1 includes some example code from Sony's basketball game *NBA Shootout 2004 (PS2)*. This code shows some (roughly 10%) of the high-level behavior states that the AI player holding the ball could perform. The system was implemented using a hierarchical FSM.

LISTING 10.1 Example FSM Behaviors from *NBA Shootout 2004*. Code © Sony Computer Entertainment America. Reprinted with permission.

```
//--------------------
//--------------------
//AlleyOop
//--------------------
//--------------------
void gAlleyOop::Update(AIJob* playerjob)
{
    playerjob->ShowGoalLabel("Alley Oop");
}
bool gAlleyOop::GetPriority(AIJob* playerjob)
{
    bool doTheOop = false;
    int shotDistanceType = playerjob->m_pPhysic->
                                        GetShotDistanceType();
    t_Player* oopPlayer = NULL;
```

TeamLRN

```
if( (fmodf(GameTime::GetElapsedTime(),BP_ALLEY_OOP_INTERVAL) <
  GameTime::GetDeltaTime())&& Random.Get(BP_ALLEY_OOP_CHANCE) &&
  ( ( shotDistanceType == t_BallAI::distance_outside ) ||
    (shotDistanceType == t_BallAI::distance_three_point ) ) )
{
    AlleyOopCoach.SetPasser( playerjob->m_Player );

    if((oopPlayer = AlleyOopCoach.FindAlleyOopReceiver())!= NULL)
    {
        if( oopPlayer->GetBallHandlerJob()->
              GetNumberOpponentsLineOfSightColumn( Basket.
              GetPosition(), BP_LINE_OF_SIGHT_WIDTH ) <= 1 )
            doTheOop = playerjob->m_Player->
                  GetBallPlayerSkill()->AlleyOop(oopPlayer);
        }
    }
    return doTheOop;
}
//--------------------
//--------------------
//LastDitchShot
//--------------------
//--------------------

void gLastDitchShot::Update(AIJob* playerjob)
{
    playerjob->ShowGoalLabel("Last Ditch Shot");
    Team[playerjob->m_Player->team].ClearMiniPlay();
    ((BallHandlerJob*)(playerjob))->DoShootBall();
    return;
}

//--------------------

bool gLastDitchShot::GetPriority(AIJob* playerjob)
{
    if( Court.IsBehindBackboard(playerjob->m_Player) )
        return false;
    if(Team[playerjob->m_Player->team].m_humanOnMyTeam &&
    playerjob->m_Player->GetBallHandlerJob()->m_justReceivedBall)
        return false;

    // last ditch effect
    return( GameState.GameClock.GetTime() <= 2.0f ||
                GameState.ShotClock.GetTime() < 2.0f );
```

```
}
//--------------------
//--------------------
//FastBreak
//--------------------
//--------------------

void gFastBreak::Update(AIJob* playerjob)
{
    playerjob->ShowGoalLabel("Fast Break");

    //try passing, it won't do it if it cannot
    ((BallHandlerJob*)(playerjob))->DoFastBreakPass();

    Vec3 basket = Basket.GetPosition();
    Vec3 target;
    target.x    = (playerjob->m_pPhysic->position.x+basket.x)/2.0f;
    target.y    = 0.0f;
    target.z    = (playerjob->m_pPhysic->position.z+basket.z)/2.0f;

    playerjob->m_pPhysic->SetDestDirection
                ( Basket.GetPlayerDirection(playerjob->m_Player) );
    playerjob->m_pPhysic->SetTargetPositionBallHandler( target );
    playerjob->m_pPhysic->SetCPUGotoAction( PHYS_TURBO );
}

//--------------------

bool gFastBreak::GetPriority(AIJob* playerjob)
{
    if( !GameState.isFastBreak )
        return false;

    if(playerjob->m_Player->GetPlayerSkill()->m_inCollision )
        return false;

    return true;
}
//--------------------
//--------------------
//LongHold
//--------------------
//--------------------
```

TeamLRN

```
void gLongHold::Update(AIJob* playerjob)
{
    playerjob->ShowGoalLabel("Long Hold");

    t_Player* passTo = playerjob->m_Player->m_pBestPassTo;
    int chance = (Basket.GetPlayerDistance(playerjob->m_Player) >
              FEET(15.0f) && playerjob->m_Player->
              m_pHasDefenderInPlace)? 90 :playerjob->m_Player->
              Personality->passes ;
    bool wouldPass = Random.Percent( chance );

    if( passTo != NULL && wouldPass)
    {
        GoalOffPass.Update(playerjob);
    }
    else
    {
        ((BallHandlerJob*)(playerjob))->DoJumpShot();
    }
}

//--------------------

bool gLongHold::GetPriority(AIJob* playerjob)
{
    //the point guard on the initial bring up
         //shouldn't be limited as much
    if(Rules.shotClock == LowmemGameRules::ON && playerjob->
            m_Player->position == POINT_GUARD &&
            GameState.ShotClock.GetTime() > 9.0f)
          return false;

Time stillTime = 0.0f;
stillTime = playerjob->m_Player->GetBallPlayerSkill()->
            m_ballHoldTimer.Get();

Time decisionTime = lerp(playerjob->m_Player->Personality->
                         dribbles/100,3.0f,5.0f);
if(playerjob->m_Player->m_isOut)
{
    if ( GameState.period >= 3 &&
        GameState.GameClock.GetTime() < 60.0f)
        decisionTime = 60.0f;
```

```
        else
            decisionTime = lerp(playerjob->m_Player->Personality->
                                playsPerimeter/100,3.0f,6.0f);
    }

    if( Court.IsInKey( playerjob->m_Player ) )
        decisionTime = 1.5f;

    bool result = false;

    if ( stillTime > decisionTime )
    {
        dbgprintf( "Long hold timeout: decision - %f still - %f\n",
                   decisionTime, stillTime );

        result = true;
    }

    return result;
}

//--------------------
//--------------------
//OffPass
//--------------------
//--------------------

void gOffPass::Update(AIJob* playerjob)
{
    char msg[80];
    sprintf(msg,"Offense pass, chance:%d",chance);
    playerjob->ShowGoalLabel(msg);

    t_Player* m_passTo  =  playerjob->m_Player->m_pBestPassTo;
    //if invalid, try the team stuff
    if((!m_passTo || m_passTo == playerjob->m_Player))
        m_passTo = Team[playerjob->m_Player->
                                team].m_bestPlayerToShoot;
    if(!m_passTo || m_passTo == playerjob->m_Player)//failsafe
        m_passTo = playerjob->m_Player->
              GetClosestPlayerToPlayer(playerjob->m_Player->team);
```

TeamLRN

```
        if(m_passTo && (((m_passTo==GameRules.LastPossession.player) &&
        (playerjob->m_Player->GetBallPlayerSkill()->
         m_ballHoldTimer.Get()>1.0f)) ||
        ( m_passTo != GameRules.LastPossession.player ) ) )
        {
          playerjob->m_Player->GetBallPlayerSkill()->PassBall(m_passTo);
             playerjob->m_Player->GetOffenseSkill()->
                        m_targetTimer.Clear();//go back to where ya from
        }
    }

    //--------------------

    bool gOffPass::GetPriority(AIJob* playerjob)
    {
        //if nobody to pass to...
        if(!playerjob->m_Player->m_pBestPassTo)
            return false;

        if(playerjob->m_Player == playerjob->m_Player->m_pBestPassTo)
            return false;

        chance =0;
        if(playerjob->m_Player->IsInsidePlayer())
        {   //inside players
            if(Basket.GetPlayerDistance(playerjob->m_Player) <=
                    FEET(2.0f))
                chance      = 10;//basket is close
            else if(playerjob->m_Player == t_Team::m_pDoubledOffPlayer)
                chance      = (playerjob->m_Player->position==CENTER)?
                                70:80;
        //double team
        else if(playerjob->m_Player->m_pHasDefenderInPlace)
        {
            if(playerjob->m_Player->GetPlayerSkill()->m_canDribble)
            {
                if(playerjob->m_Player->Ratings->insideShooting<75)
                    chance  = (playerjob->m_Player->
                              position==CENTER)? 60:50;
                    //covered, can dribble, low inside shot
                else
                    chance  = (playerjob->m_Player->
                              position==CENTER)? 20:40;
```

```
                    //covered, can dribble, high inside shot
            }
            else
                chance  = (playerjob->m_Player->
                           position==CENTER)? 50:70;
                //covered, can't dribble
        }
        else
            chance = 10;//not covered (or dteamed, or really close)
    }
    else // outside players
    {
        if(!playerjob->m_Player->GetPlayerSkill()->m_canDribble)
            chance      = 100;//can't dribble
        else if(playerjob->m_Player->m_pHasDefenderInPlace)
            chance      = 30;//covered
        else
        {
            if(!playerjob->m_Player->m_pHasDefenderInPlace)
                chance      = 10;//wide open
            else
                chance      = 30;//not covered, no lane
        }
    }

    //offset for longer holds, greater increase if
    //you're inside or can't dribble
    float modVal;
    if(playerjob->m_Player->IsInsidePlayer() ||
       !playerjob->m_Player->GetPlayerSkill()->m_canDribble)
    {
        modVal = GameTime::GetGoalDeltaTime();
    }
    else
    {
        modVal = 0.1f;
    }

    float rem = fmodf(playerjob->m_Player->GetBallPlayerSkill()->
                m_ballHoldTimer.Get(), modVal);
    int holdAdj = int(rem/GameTime::GetGoalDeltaTime());
    chance += holdAdj;
```

TeamLRN

```
    //now check for tendencies
    bool wouldI = Random.Percent( playerjob->m_Player->
                                        Personality->passes);

    return (wouldI && Random.Percent(chance));
}

//--------------------
//--------------------
//Dunk
//--------------------
//--------------------

void gDunk::Update(AIJob* playerjob)
{
    playerjob->ShowGoalLabel("Dunk");
    if(playerjob->m_Player->GetBallPlayerSkill()->DunkBall())
        playerjob->m_Player->Task.SetCPUSequence(TaskDoChargeMove);

}

//--------------------

bool gDunk::GetPriority(AIJob* playerjob)
{
    //don't try if you can't
    if(!playerjob->m_Player->m_canDunk)
        return false;
    //always dunk if you're wide open
    else if(playerjob->m_Player->m_laneCoverage <= 0.1f)
        return true;

    //otherwise, use personality
    return(Random.Percent(playerjob->m_Player->Personality->dunks));
}
```

Data-Driven Systems

With huge numbers of players and callable plays, vast statistical data, and tons of animations, sports games rely on at least some data-driven AI. Plus, with a push toward ever more realistic sports AI as well as online play, data-driven systems will make it much easier to tune the AI, and to update it with online changes that reflect

either real-life player statistic changes or further game balancing polish. Some things that are commonly performed with data driven techniques are the following:

- *Playbooks.* Instead of creating plays for the AI system, a better system is to create atomic behaviors that the AI-controlled players can perform, and then have an editor that designers can use to chain these behaviors into full plays to create the playbook for the teams in your game. In this way, the designers can experiment with new plays and handpick the best ones (or the ones that each team likes to use most in real life), and the AI programmer can now concentrate on additional behaviors, instead of trying to tune hardcoded plays.
- *Animation picking.* By being able to specify (through a visual editor or some kind of scripting tool) the types of conditions that specify the best animation for a given behavior, designers can quickly spell out the kinds of animations that make sense for each in-game action and can change or expand these animation lists as needed, without any code changing.
- *Player statistics.* At this level, the players need statistical data that approaches the levels represented by their real life counterparts, and additional in-game statistics must be created so that the myriad attributes can be related in some way to the game simulation.

Messaging Systems

With many players having to communicate to each other, and such a dynamic environment, it makes very good sense to include a messaging system into the AI framework for your sports game. Everything from coordinating plays between two players (or even collision events), to noting actions by the human, could be sent through the messaging system, with the AI responding to only those messages that he is interested in, instead of having to monitor the entire playing field continuously. Different levels of the AI system can use the same system as well so the physics layer will respond to the collision event, where the team level will respond to a coordination event between two players.

EXAMPLES

Early sports games, such as *Football* and *Basketball* on the Intellivision and Atari, couldn't even support the full number of players on each team. They also used simplified AI that more resembled pillars that you had to negotiate around.

Sports games really began to come into the spotlight with the NES game system, as programmers finally had the horsepower necessary to do a much better job

TeamLRN

of approximating the game, although still at a somewhat primitive level. Games like *RBI Baseball, Tecmo Super Bowl, Ice Hockey,* and *Double Dribble* are still loved by sports games fans. The gameplay employed by these titles was simplified, but was approaching a simulation of actual play, and we finally started to see a greater use of statistics (instead of two equal teams playing against each other).

Many of today's games, even with their greater graphical look, still employ most gameplay institutions that were created during this period. This has in some ways made sports games a bit stalled in gameplay evolution, but it has the advantage of making most games instantly playable by longtime fans because the control scheme, overall game mechanics, and general game strategies are still somewhat familiar. A similar situation occurred with fighting games borrowing *Street Fighter*'s six-button control layout and special joystick moves.

The 1990s brought us the continuing seasonal versions of all the popular games, now in 16-bit versions and beyond. As the games incrementally increased in quality and scope, and as the consoles began to use more sophisticated controllers, the games gave the player more controls to do things with. This means the AI has to follow suit, so the complexity increases.

Today's sports games are marvels of AI, with perennial games like *Madden,* Sega's *NBA* and *NFL 2K* series, and *World Soccer* playing sophisticated simulations of their sports, while showing the personalities of the players and giving the game player a great sports experience. These games use a variety of AI systems, including complex FSMs to make play calling and tactical decisions against you, data-driven systems to choose the correct animations based on several factors, sophisticated simulation calculations to make in game characters perform like they do in real life, and even more in an increasing attempt to make the games more realistic and fun.

AREAS THAT NEED IMPROVEMENT

Learning

Sports game AI continues to fall the victim of exploits, with even the best AI-controlled team losing because the human did something repeatedly that the AI is poor at stopping. If the AI could compensate for this by specifically targeting this repetitive behavior, it would force the human player to either change his game tactic, or stop scoring so easily. Teammate AI could also learn from this, by discerning favorite plays that the human employs and better support that play if it were to happen again. This type of sports learning has been implemented using influence maps (by incrementally changing positioning data to reflect more winning positions) and by statistical learning (by keeping track of behaviors that work, or don't work, and adjusting appropriately). This system doesn't have to increase difficulty of the

game; it will just stop exploits from ruining the overall performance of the AI system. In the end, this system will merely cause the player to change his game plan a bit more often, and the overall experience will just be that much closer to a real game. Of course, this same system can be used to increase difficulty, if you want, because the system can learn quite quickly the kinds of things that the human is poor at stopping, and have bias toward those kinds of behaviors (in effect, the system is finding exploits against the human's intelligence).

Game Balance

Sports games quite frequently suffer from the problem of game balance. Certain sports tasks, like defense in basketball, are much harder to do than others (the reason for this is that basketball is a very fast sport, and the actions of the defense are reactive, thus always slightly behind the offense). How do we support this for the human (to make this task fun), without killing the balance of the game by making it too easy to defend, and therefore shutting down the offense? As this issue continues to evolve, on a case-by-case basis, it will continue to consume the AI programmer's time as he comes across problems that require decisions based on the game at hand and the fun factor of the game.

Online game play further complicates the task of game balancing. So far, there has been an inherent lag associated with all but the fastest connections in online games because of bandwidth limitations and other issues. The kinds of highly reactive behaviors in sports games end up suffering visually because of it, more so than in more physics-based games like FTPS, which have very simple animations and can use physics to predict character and projectile movement to fill in the gaps caused by lag.

So, in online sports games, game balance issues relate to the problem of dealing with this lag, especially in games that can get out of synch with each other, or are running an event-driven networked game method. If one player sees that he caught the pass, but the server machine says that he did not, then that player is going to be pretty confused when he suddenly doesn't have the ball anymore. If this happens once, it might be overlooked. But if it is a systemic problem, where the clients in your game are continually catching up to the server's reality, by popping animations, behaviors, and positions, the game becomes unplayable in a hurry.

Gameplay Innovation

Sports games have become increasingly similar, and somewhat stagnant, in how they play. Marketing has driven innovation almost out of this highly profitable sector of the game industry. Even *Madden,* arguably one of the best and most successful franchises in all sports gaming, hasn't done anything really innovative in many

TeamLRN

years. The *Madden* team has incrementally improved graphical quality, presentation, and animations and have also made some small changes to the interface. But, the game is almost identical, gameplay wise, to the some of the earliest *Madden* football games. It's just a lot prettier. Is this really what the consumer wants? Or is this what the consumer has been given? The motivation, of course, is to not lose any market share by scaring people off because of gameplay mechanics or AI behaviors that people either don't enjoy immediately or can't learn quickly enough. No matter what marketing thinks, people will buy a game and actually spend the time to learn a new interface or game mechanic if the experience is good enough. Nobody knew how to control a basketball game when the first one came out, yet, customers still bought it.

There is plenty of room for innovation in the sports game world, both in gameplay and in competitive and cooperative AI. We must strive to offer something new to the consumers, lest this genre begin to grow stale. Imagine an AI system in football that discusses things with you during a huddle and helps to develop a plan against the other team. Imagine a commentator AI system that does television-style slow motion while remarking about the play and drawing things on the screen for emphasis. Imagine more intuitive voice controls for these games, where you could shout "toward" a certain player (with head movement tracking or some other means) and get an appropriate response. These are the kinds of things that will keep the genre fresh and growing.

SUMMARY

Sports games have come a long way from the incredibly simplistic versions that were first created for home consoles in the 1970s. With ever more realistic visuals and gameplay, the need for high-quality AI controlled athletes is ever higher. Sports games are some of the highest moneymaking games in the business right now, and the players that shell out that money demand quality in every element.

- The two main categories of sports gameplay are fluid and resetting games. Fluid refers to games that have mostly nonstop gameplay, with very dynamic situations. Resetting games are those that have periodic resets or stops in the action, so are more linear.
- The high level of sports knowledge of the common purchaser of sports games means that a much higher level of detail needs to be taken into account for simulation style sports titles.
- A coach- or team-level AI provides the system with more far-reaching decision-making and provides a means for coordinating actions among multiple players.

- Player-level AI systems are usually more tactical than the coach level and usually include both decision-making and animation-selecting elements.
- Pathfinding in sports games usually involves much higher numbers of dynamic obstacles and needs to take into account special means of travel with the rules of a specific game.
- Animation-picking systems are very important to sports games that include numerous animations for any possible action because the system needs a fast way to query a database of animations and make intelligent decisions.
- Miscellaneous elements make a world bigger than the game court and give the player a greater sense of immersion.
- FSMs and FuSMs are used widely in these games. The type of game (fluid or resetting) can sometimes be a factor when using these techniques, but because of the inherent nature of any sports game, some degree of state machine will be used in the construction of the game.
- Data-driven systems help offload some of the tremendous amount of detail that needs to be addressed, on a player level, playbook level, and animation level.
- Messaging will help the various layers of the AI system communicate and provides a quick means of cutting through the very dynamic environment.
- Learning will help to solve the problem of AI exploits and could aid the player in learning the system.
- AI systems need to extend their abilities in those areas where game balance and fair gaming need to be addressed because the addition of too much intelligence in the system will give more aid to the player, but wreck game balance.
- The genre must continue to innovate in gameplay and opponent and cooperative AI systems, so it doesn't go stale. This is especially true now that the visuals of these games have achieved almost television-quality levels because they will no longer be such a point of concentration.

TeamLRN

11 Racing Games

The racing genre is a very interesting one, both from a gameplay standpoint and from an AI standpoint. The genre is mostly divided into two main groups, vehicular and specialty. The two groups do have a common thread, which is having at least some semblance of a physics-based simulation of racing.

Early games like *Pole Position* (or even its granddad, the 1974 Atari game, *Gran Trak*) are much more along the lines of action games, in that the processing power of the hardware at that time really didn't allow for much simulation—they were really just fun gameplay systems.

Yes, some racing games (even modern ones) do take liberties with their physics, but that's what videogames are about, holding onto the areas of reality that we don't mind being limited by, and stripping out the parts of reality we do. So, we want mostly realistic cornering and handling, but we also want to be able to jump a car over ten semi trucks and still be able to drive away after landing. This is like gamers who don't mind having to reload a rocket launcher between shots, but they would mind if they could only carry three rockets at a time; they want a hundred shots in the backpack, never mind that a load like that would probably weigh far more than the character could carry for any distance, much less jump with.

Two variants of vehicular racing games appeared early, and the split stuck. They are differentiated by their camera perspective: the first-third person racing game (such as *OutRun,* or *Stun Runner*) and an overhead view (*RC Pro-AM,* or *Mickey Thompson's Offroad Challenge*). The overhead games tended to be skewed toward the more arcade feel of game, with very unrealistic physics; the other group stayed more true to its roots.

The specialty racing games are mostly fad driven—they involve the hot racing style sport at the time. Past examples that received some degree of success include snowboarding, skiing, boats, wave runners, hovercraft, dirt bikes, and the like. These games had to augment traditional racing AI with sport-specific behaviors, such as performing tricks or dealing with futuristic or nontraditional physics systems.

TeamLRN

One last subtype is the cart racing game (made popular by *Mario Kart,* but since has seen decent success with quite a few different characters), which simplifies the driving portion of the game and adds obstacles, strange tracks, and other action elements.

Pure vehicular simulation can become a fairly technology intensive undertaking. You need fairly complex mathematical solutions to deal with the different suspension systems, good multibody collision handlers, AI opponents that can adjust to differing road conditions (especially for off-road racing or in games that include rain, oil, or ice hazards), as well as any special concerns your game might bring.

Some of the best racing games have become showcases for the computational and graphical power of new game systems as they first are released. This is because the physics models and control schemes that these games are using nowadays have been so highly polished that they need almost no tweaking at all; you just work on a nice graphics engine, pump out some higher-quality car models, and there you have it: a finished, high-quality launch title.

Overall, the AI of pure racing games has gotten very advanced over the years, with many great examples of track AI that does a very competitive job without cheating. In fact, the racing genre was starting to loose popularity because of a lack of freshness. Too many games came out in which the primary driving simulation was so good, and so close to reality, that almost nothing could be done any better. The genre needed a shot in the arm to revive it.

In 1995, *Twisted Metal* was released, and the first true vehicular combat game was born (although other games released earlier had cars and weapons, they were usually more cartoony, like *Mario Kart,* or just plain action games, like *SpyHunter,* so they weren't really driving simulations, but they were definitely an influence on the genre). *Twisted Metal* was a moderately realistic driving simulation (for its time), coupled with arena-style levels and a bunch of weapons thrown in. People forgave the subpar graphical quality and the very strange control setup because the additional gameplay elements were truly original, and very fun to play. It wasn't enough, however, mostly because the single-player experience suffered from bad AI (both the performance as well as the difficulty level), and the gameplay was pretty repetitive when the player was not playing against another human (trash talking at your buddy sitting next to you, and hearing him scream as he's killed, seems to add replay value to almost any game). Other games came out, including the stylish *Interstate '76,* which added the concept of a linear story and overall attitude that worked well. But it also suffered from the replay and single-player problems of *Twisted Metal.* Again, the genre needed more.

Recently, that something more seems to have come. By going one step further, and adding complex adventure and story elements to the racing genre in addition

to weapons, games have truly opened the possibilities for racing games. *Grand Theft Auto* started out in 1997 as a somewhat primitive overhead 2D game with a very simple concept: to provide a living city in which the player can perform many different activities, including driving, to eek out a life as a thuggish criminal. It has kept the concept over the years, but has since moved to the full splendor of a completely realized 3D world, with a realistic driving simulation (well, maybe a bit over-the-top driving simulation), and all the sex, violence, and rock and roll you could imagine. It has also become one of the best selling games of all time, with the four games in the series selling more than 25 million copies so far. The combination of providing open-ended gameplay and adult content has proved hugely popular. Many other games have since capitalized on this formula, so the full-blown vehicular action genre has picked up where the racing simulation and the combat games have left off. The action elements of these games venture quite far into the adventure or first-person shooters/third-person shooters (FTPS) game's territory, but the primary gameplay system is vehicular, or at least it has been until now.

COMMON AI ELEMENTS

Track AI

Where the road meets the rubber, track AI is the system needed to keep a CPU-controlled car on a racetrack (or city street) at high speed and within the rules of the game. Usually, this is a state-based system, with the different vehicle states detailing the main ways that a racer can be on the track (most likely `OnTrack`, `OffTrack`, `WrongWay`, and `Recovering`, or something similar). Each vehicle state would have ways of steering and applying the throttle and brake to best serve the particular state the vehicle is in, combined with the vehicle's position on the track and relation to the other drivers. As guidelines, most games use a combination of physics and "optimal lines of travel" (which are either data paths laid down in the track editor, or calculated automatically by a technique known as "finding the path of minimum curvature," as shown in Figure 11.1) that mimic the invisible lines of travel that humans use when they race on tracks and roads. In addition, there are also optimal offset positions, if the true optimal position is already occupied.

Note too that some racing games are not occurring on roads, but water (with boats or jet-skis), snowy mountains (with snowboarding), or even more exotic terrains (like the tubes and chutes of *Stun Runner*). Thus, they might not use a pure version of the minimum curvature technique because the dynamics of the surface might entail other types of optimal maneuvers.

TeamLRN

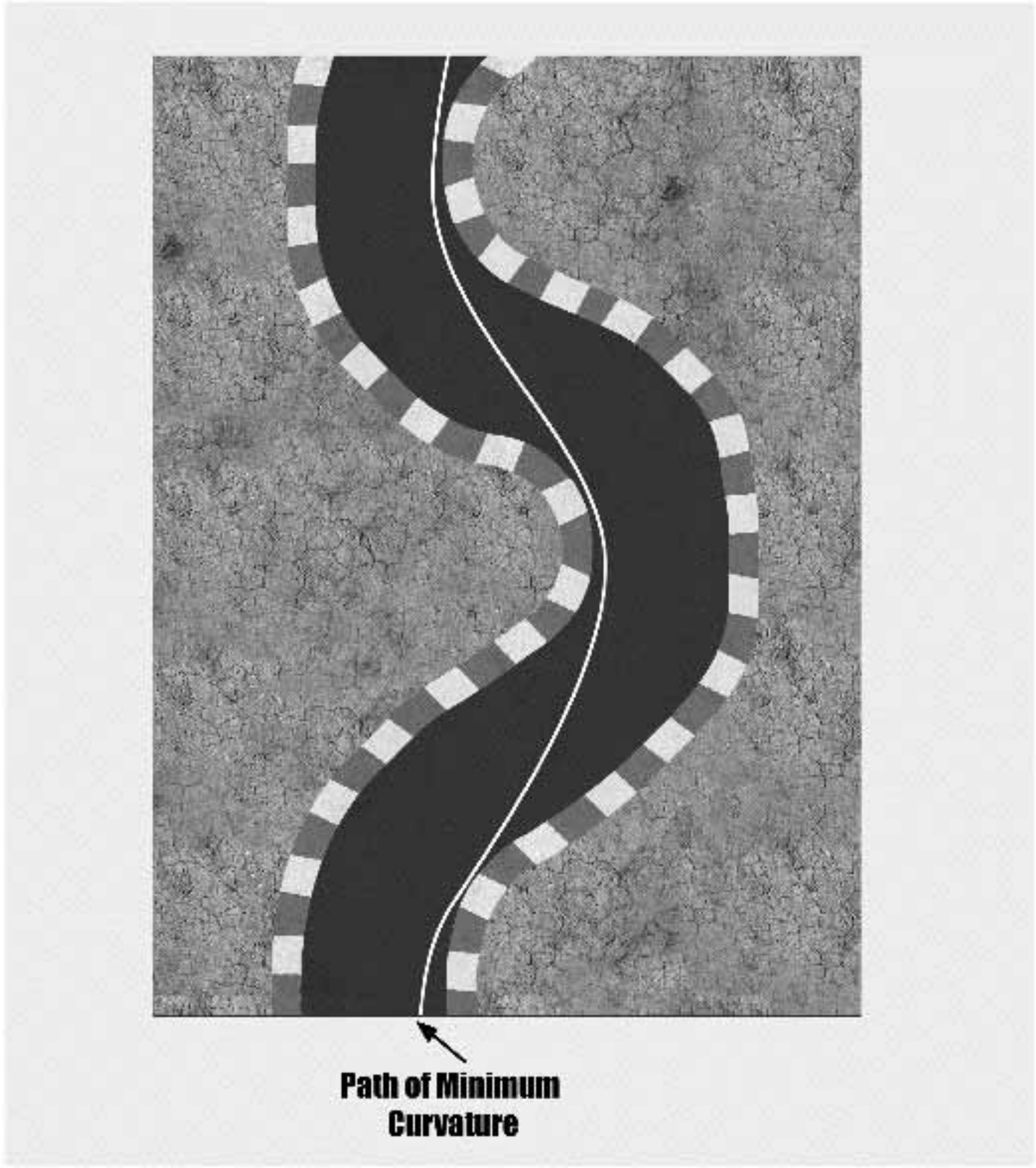


FIGURE 11.1 Track with path of minimum curvature shown.

Traffic

A number of these games are built around racing in functional cities, so they have working traffic systems, complete with stoplights, highway systems, and plenty of cars using them. The traffic in these games is usually just good enough to be good looking, but rarely reacts much to the player's movements (in fact, that might be on purpose; you don't want everyone getting out of your way, now, do you).

Some games, however, use complex enough traffic systems that they are very realistic, with lane changes, cars getting over for police vehicles, proper use of traffic lights and intersections, and so on. This is mostly FSM behavior, with a lot of message passing to ensure that accidents don't happen (unless some rowdy human happens along at 130 mph or something), and some randomness to ensure that things don't look repetitive.

Pedestrians

Ever since race games started appearing with cities for backdrops, there have been pedestrians to deal with. Different games have done different things with them; the *Midtown Madness* games, being a bit "nicer," have simply had the pedestrians walking around on paths somewhat randomly, and if a car gets too close, they dive out of the way. But some games, like *Grand Theft Auto* or *Carmageddon,* let the user pretty much run over anybody he wants. The pedestrians will try to get out of the way, but clever violence hounds can find a way, and the people will fall. In fact, *Grand Theft Auto* has quite a range of pedestrian types, all of which are running different AI, based on function. In most games, this type of behavior is state based, probably with some messaging.

Other systems use very simple flocking-type behaviors, with areas in the level being assigned particular values of attract and repel (thus, certain storefronts might attract people, who would look in the window for a while and then walk toward the next attractor, whereas a dead body might be a powerful repelling force, so that people look like they're avoiding the accident). *State of Emergency* made good use of a system similar to this. The crowds were very fluid and reacted well to most of the action.

Enemy and Combat

This is the car equivalent of deathmatch bot code because some games allow full combat either car-on-car, or pedestrian-on-car, or any other combination. This code needs to combine the race AI mentioned earlier with the bot AI from FTPS games, including the human-level performance checking that would do things like making the AI misfire and drive into walls occasionally, to ensure that the player doesn't feel cheated (or merely that he's being pursued by a relentless evil robot, unless that's your intention). It might also include multiple cars working together, as in police cars taking different streets to cut off multiple ways of escape, or two cars boxing you in so you can't turn.

Nonplayer Characters

NPCs are the noncombative people in the game that you might have to deal with, such as people who are going to give you information, or sell you a better car. As in role-playing games, these characters usually have scripted behaviors and dialogue to facilitate these encounters. They generally aren't very reactive because most of these games don't have sophisticated conversation engines (it's really not the point; if people want that, they'll play an RPG), so most NPCs are usually handled in a noninteractive cut scene.

TeamLRN

Other Competitive Behavior

Some racing games also require other behavior from their AI opponents, such as performing tricks in snowboarding or motocross games. These systems need to have either scripted chains of moves that look well together or a decent enough understanding of physics and timing so that they pick moves that they can pull off successfully and stylishly. This kind of decision structure actually is more like a fighting game, where each move would have some kind of time associated with it (that is, how long it takes to perform the move), and the AI would make move determinations based on how long it has (from simple physics calculations that take into account speed and height achieved), as well as skill level and personality.

USEFUL AI TECHNIQUES

Finite-State Machines (FSMs)

Race games have a fairly straightforward AI layout, mostly defined by the laws of physics, and the (usually) simple objectives of your current "race" (be it to get to the finish line first, or to pick up a package and bring it back while surviving the attacks of the other players). Also, the state layout for the game flow of most classical racing games is very straightforward (start, racing, off the track, overtake, pacing, pit). FSMs make themselves useful again.

Scripted Systems

The vehicular action genre usually follows a story of some sort (although some are extremely open ended) and work well with the scripting paradigm. Also, some of the ambient pedestrian and traffic systems can lend themselves well to a scripted system, where various patterns of movement are scripted and interact with the street layout of the city. Sometimes this is just a first layer, with overriding reactive systems in place to affect this scripted behavior in specific ways. So, if you have a crowd milling about in a mall, checking out the merchandise, using the escalators, and such, this could be a series of small scripts that each AI-controlled person would use to look like they have intimate knowledge of the environment. But if a car suddenly comes crashing through the window, the pedestrians' flee behaviors would kick in, overriding the normal script, in a mad dash to escape being crushed.

Messaging Systems

The ambient traffic and pedestrian systems most commonly use messaging systems to talk to one another and coordinate movement in the complex ways that these things happen in real life. Of course it is also possible to code these types of behav-

ior using FSMs (even if you use a messaging system, you'll still probably want to control overall behavior of traffic and pedestrians with scripting or state machines), but if you're going to have a large number of ambient vehicles and walkers (and want them to respond to periodic or situational events either singly or in coordination), this is probably the way to go.

Genetic Algorithms

Some of these games have an enormous number of cars (*Gran Turismo 2* has more than 500), each of which require tuning of their handling and performance abilities, so some companies have used techniques to automate this tuning task with a simple offline genetic algorithm application used to modify the car's performance parameters until optimal results are achieved. These results are then stored and used directly during actual gameplay. This is a very straightforward use of this kind of technique (as a preprocessor that optimally tunes a system of parameters), and the amount of time the genetic algorithm will take to perform these calculations is dramatically less than the time it would take a programmer or designer working within the game using trial and error.

EXAMPLES

Driving games have been with us almost from the beginning, with the earliest ones coming out in the early 1970s. These early driving titles were little more than a scrolling field of two small lines that you had to stay between. But this simple representation is all the mind needs, given a steering wheel and a gas pedal, that is.

The driving game has come a long way, with the older *Pole Position* and *SpyHunter* looking dated next to the almost movie-quality visuals of today's *Gran Turismo.* Also, the arcade style and fast and loose gameplay of the past has been all but lost to the almost perfect rendition of the handling and performance modeling in today's better racing games. Not that we missed realism in games like *Crazy Taxi,* however. *Midtown Madness* gave us great city traffic, *The Simpsons: Hit and Run* successfully extended the game model to a comic license and managed to keep the comedy, *Interstate '76* infused a degree of style and a good story into the mix, and *Carmageddon* actually had us using the windshield wiper to clean off the blood.

AREAS THAT NEED IMPROVEMENT

Classical racing simulation games have been all but mastered. If your racing simulation doesn't include a well-built, solid physics model combined with a polished,

TeamLRN

intuitive control scheme, ultrarealistic visuals, and some way to differentiate yourself from the games that already have accomplished all these things, don't even bother putting it on a store shelf. However, the new variations of incorporating vehicular racing with other elements of gameplay still have many directions in which to improve.

Other Areas of Interest Than Crime

To possibly push these games more mainstream (which is hard to imagine considering the many millions of units these types of games have already sold), more parent palatable game types could be found—not everybody's mother wants to see her kid running over a prostitute for her wallet. Violence in videogames does sell, but it doesn't have to be as extreme as in *Grand Theft Auto.*

More Intelligent AI Enemies

Imagine you are being chased by teams of AI cars, but they rarely work together to set up roadblocks and head you off; instead, it becomes a Blues Brothers–style chase with one lead vehicle being trailed by 40 cop cars. This is pretty much the norm for the genre, but more complex maneuvers could be used for the opponents. Just give the human "criminal" player a police scanner, so he can hear about the roadblocks slightly ahead of time and circumvent capture. Some games are making headway in this area, but they are rare.

Other problems can be seen in simple overtake maneuvers on racetracks in some games. AI-controlled cars in some games pay very little attention to other AI-controlled cars; they do adjust their speed and turning to some degree, but the collision between AI vehicles is tuned to minimize the effect they have on each other to simplify the overall race simulation. Thus, AI cars in some games don't use real overtake moves to get by each other—one car will bump the other out of the way, in a subtle way that looks OK from afar, but not up close. Instead, why not give each vehicle a more realistic AI race model, so that the human doesn't notice this AI cheat.

Persistent Worlds

A vehicular action game has not yet been adapted to the multiplayer online model, but this would be a big boon to the genre. Imagine a game based on the *Autoduel* world (the 1985 game from Origin™ based on the Steve Jackson *Car Wars* pen-and-paper RPG—it's sort of a Mad Max after the collapse of civilization type of place), or *Grand Theft Auto,* for that matter. The dynamics of these kinds of story worlds lend themselves very well to the gameplay mechanics of racing with the large, open worlds that online games require.

The problems lie in simple computing power; driving the complex mathematics of the vehicle simulations and running traffic AI for an entire city (rather than a sphere of traffic centered on the player, as is used in *Midtown Madness*) do not work well with the limited bandwidth capabilities of the Internet. Such online game choppiness (which is somewhat tolerated and can be compensated for in some game types) might make the game unplayable. We shall see whether or not these limitations can be breached and bring racing-style gameplay to the online community.

SUMMARY

Racing games went from very simplistic games in the 1970s arcades, to some of the most graphically and technologically sound games of all time. This quick rise in quality came at the price of gameplay innovation, however, and the genre almost stalled out. The modern infusion of additional gameplay elements into racing games has truly invigorated the genre and gave it a new life.

- The racing genre is globally defined as a game using a somewhat physics-based model of racing.
- Vehicular racing games involve the more common types of vehicles: cars, motorcycles, F1 racers, and so on. The vehicles can be on or off road, and involve an actual racetrack, or take place in a city or other locale.
- Specialty racing games involve racing of some other type, like jet skis, snowboarding, or the like.
- The creation of vehicular combat games increased the gameplay potential of the genre, and adventure and action elements were eventually added, extending to the vehicular action game.
- Track AI is the system by which CPU-controlled racers maintain control while racing over the terrain within the confines of the physics system and rules of the game.
- For games that take place within urban areas, traffic and pedestrian systems greatly add to the visual and situational realism of the city.
- Combat AI is required in games that use these additional gameplay elements beyond the racing competitions.
- NPC AI would be required if your game uses additional character interaction other than combat or specialized areas of economy or information.
- Other competitive elements would require AI work, if your game was such that it involved doing tricks or other actions while racing.
- FSMs make themselves useful in this genre because of the linear nature of most race scenarios.

TeamLRN

- Scripting lends itself well to the story of a vehicular action game, as well as to the nature of traffic and pedestrian systems.
- Messaging will ease the need for communication between game elements in complex race and traffic AI systems.
- Genetic algorithms can help automate the process of tuning the handling and performance parameters of the hundreds of cars that are sometimes represented in a large racing game.
- Other areas of interest other than crime need to be explored for vehicular action games. This will continue the push toward mass appeal, and keep our mothers off our backs.
- The opponent AI needs additional intelligence because the level of pathing through cities and overtaking on racetracks is still inferior to human level.
- A persistent world game in this genre could do a lot toward extending the genre.

12 Classic Strategy Games

Game Theory is roughly defined as *the study of human behavior when dealing with interactions in which the outcomes depend on the strategies of two or more persons who have opposing or, at best, mixed motives.* John von Neumann virtually founded the field in 1928 by studying the concept of bluffing in poker and discovering that the analysis had significant ramifications for economics.

In Game Theory, the concept of a *game* takes on special meaning. Instead of the more common entertainment-oriented definition, Game Theory defines the *game* as *an undertaking where several agents strive to maximize their payoff by taking actions, but the result relies on the actions of all the players.* By discovering that this generalization exists across different types of "games," Game Theory hopes to explain these kinds of human interactions across many varying playfields, from business to war, and from the checkerboard to overpopulation.

Some of the classic "games" that have been studied under Game Theory include Barbarians at the Gate, Mutually Assured Destruction, the Prisoner's Dilemma, and Caveat Emptor. These are all mathematical constructs that attempt to define what are called dominant strategies of the various human behaviors that each detail.

In some of his earliest work, von Neumann made a very important discovery, with one very large requirement. The discovery was that for some games, *rationality* (meaning the best action to take) could be mathematically calculated, given the strategies and payoffs inherent in the game. The requirement was that the game be what is called a *zero-sum* game, which is defined as being *a game in which one player's winning actions directly result in another's equivalent loss.* In other words, these are games in which a number of players engage in a system of pure competition, where there is only one winner. This is not a trivial requirement. Many of the more socially important problems that Game Theory had hoped to tackle (such as economics, dealing with use of natural resources, and political systems) are not zero-sum games. Although Game Theory can still give insights into these other kinds of games, it cannot help define game-specific rationality like it can in the limited world of zero-sum games.

TeamLRN

Von Neumann's work became a foundation for early AI researchers' work, as they set out to create programs that could accomplish complex tasks requiring rationality. How best to test their creations than by finding some abstract version of worldly problems, that also manages to fit neatly into a clean mathematical model, so that rationality can be assured? Zero-sum games answered the call and are still some of the most studied of all AI problems.

Classical strategy games such as chess, checkers, tic-tac-toe, and even poker are all examples of zero-sum games. It also turns out that nonzero-sum games like *Monopoly* (where it might be possible that two people could form and alliance, and both "win" money from the bank) can be converted to a zero-sum game by considering one of the players to be the board itself (or the Bank, in *Monopoly*). This ghost player is in essence losing the sum of the amount won by the players, and thus all the formal assumptions and proofs concerning zero-sum gaming can be employed.

Researchers began using computers to build an "intelligent program" capable of playing these games almost as soon as computers made their appearance. Alan Turing (of the Turing test fame) and Claude Shannon wrote some of the first chess programs in 1950, barely five years after ENIAC came online. Both men put forth that a program that could competently play these games epitomized the definition of something requiring (and exhibiting) intelligence.

This brings up an interesting parable about AI problems in general. In the past, if a task was too difficult for a computer to accomplish, it was said that if someone could devise a program to do that task, then that program would be intelligent. But, after years of work, when someone finally does release a program that performs the task, the detractors declare it to be simple brute force search (or whatever computer technique the program uses), and not real intelligence. Thus, AI never gets to actually solve any problems.

Researchers turned to games for a number of reasons. They are more complex and lend themselves more to real-world situations than so-called *toy* problems do and represent a more uncertain and (somewhat) exciting world than massive search determinations like The Traveling Salesman Problem, or integrated circuit design. Classic strategy games also personify the optimal conditions for classic AI search techniques. They are games of perfect information (both players know everything about the game world), the moves are mostly global in effect (rather than within some small sphere of influence), and the games are turn based, which gives the computer time to think. Strategy games are also very complex (state space wise), thus requiring intelligent methods for finding rational solutions. This is precisely the list of attributes that currently make a good computer AI simulation. However, because these games also add the element of an opponent, they provide the problem with elements of uncertainty and, more specifically, directed uncertainty. Undirected uncertainty would be randomness introduced by dice or some

similar means, and is thus unbiased and is merely part of the cost of playing. But directed uncertainty deals with things like bluffing, mixing strategies to appear random, and using irrational moves for whatever reason.

If you consider the previously mentioned optimal conditions for AI problem solving, it is easy to determine the parts of strategy games that will be weak for an AI system. Closed chess endgames (the term "closed" refers to a state with a number of interlocked pawns across the middle of the board; see Figure 12.1) are notoriously hard for traditional AI systems. The reason? The moves are no longer global in effect. Suddenly, we can cut up the chessboard into separate chunks and throw off the computer by making diversionary moves on the other side, to make the AI system think something is going on. Tactics like this are one way that Gary Kasparov beats many of the computer chess programs (and because he's one of the best chess players in the history of the game, of course).

FIGURE 12.1 A closed chess game position.

What separates most academic studies from more traditional entertainment versions of classical gameplaying programs is the notion of a time limit. Given the unreasonable request of an infinite amount of time, the best solution will always be found. But given the limits of the real world, gameplaying programs always have

TeamLRN

some form of time limitation, and we must make do with the amount of time that we have allotted to us. Of course, as computation speeds go up, we are getting closer and closer to the point when brute force methods will be possible, given even modest time constraints. But there will always be another, more complex game that will force AI researchers to use alternate methods to find better solutions fast, without relying on total brute search.

AI researchers have "solved" several of these games, meaning that the entire state space has been mapped out and can be easily searched by today's computers to result in optimal performance (that being a win for the first player to move, or a draw). Games that have been solved are tic-tac-toe, checkers, *Connect Four*, *Go-Moku,* and *Othello.* Several others are in various states of being solved. Chess is getting close. The highest-classed chess programs use a stored "opening book" (chains of moves that have been researched over the centuries by chess masters to give good play), use a smart search technique of some kind for the transitory middle game phase, and then have another stored database of good moves for the endgame phase. See Figure 12.2 for a listing of solved and partially solved games.

Solved

- ☑ Awari
- ☑ Connect Four
- ☑ GoMoku
- ☑ Hex (up to 9 x 9)
- ☑ Nim
- ☑ Nine Men's Morris
- ☑ Three Men's Morris
- ☑ Qubic
- ☑ Tic Tac Toe

Partially Solved

- ☐ Checkers
- ☐ Chess
- ☐ Go (up to 4 x 4—game is usually 19 x 19)
- ☐ Reversi

FIGURE 12.2 Classical games that have been solved, in whole or partially.

Some games can have such huge state spaces (the game of *Go* has a game tree size of around 10,400, which is a number larger than the amount of atoms in the universe, give or take) that they are all but immune to brute force search methods

and, thus, require either very clever directed search routines within recognized portions of the state space or intelligent algorithms to develop novel solutions given the game rules. Either way, these are some of the most classically defined AI problems there are. Listing 12.1 shows the `search()` and `think()` functions from the open source chess program, *Faile*, written by Adrien M. Regimbald. The entire source is on the CD-ROM, along with its corresponding Web links for more information. *Faile* is a very compact (the entire source zip file is 42 K), yet full-featured, alpha-beta search system, which gives this tiny little program expert-level AI play capability. Notice that the search function uses bounded optimality, in that it has a time limit, and will make decisions based on the best move it has seen given the time it has left, and will even make decisions on whether to continue search or not based on time. More detail will be given on this later in the chapter when alpha-beta search is discussed.

LISTING 12.1 `search()` and `think()` from *Faile*. Distributed under the MIT license.

```
long int search (int alpha, int beta, int depth, bool do_null) {

  /* search the current node using alpha-beta with negamax search */

  move_s moves[MOVE_BUFF], h_move;
  int num_moves, i, j, ep_temp, extensions = 0, h_type;
  long int score = -INF, move_ordering[MOVE_BUFF],
           null_score = -INF, i_alpha,h_score;
  bool no_moves, legal_move;
  d_long temp_hash;

  /* before we do anything, see if we're out of time
         or we have input: */
  if (i_depth > mindepth && !(nodes & 4095)) {
    if (rdifftime (rtime (), start_time) >= time_for_move) {
      /* see if our score has suddenly dropped, and if so,
             try to allocate some extra time: */
      if (allow_more_time && bad_root_score) {
    allow_more_time = FALSE;
    if (time_left > (5*time_for_move)) {
      time_for_move *= 2;
    }
    else {
      time_exit = TRUE;
      return 0;
    }
      }
```

TeamLRN

```
    else {
  time_exit = TRUE;
  return 0;
    }
  }
  #ifndef ANSI
  if (xb_mode && bioskey ()) {
    time_exit = TRUE;
    return 0;
  }
  #endif
}

/* check for a draw by repetition before continuing: */
if (is_draw ()) {
  return 0;
}

pv_length[ply] = ply;

/* see what info we can get from our hash table: */
h_score = chk_hash (alpha, beta, depth, &h_type, &h_move);
if (h_type != no_info) {
  switch (h_type) {
    case exact:
  return (h_score);
    case u_bound:
  return (h_score);
    case l_bound:
  return (h_score);
    case avoid_null:
  do_null = FALSE;
  break;
    default:
  break;
  }
}

temp_hash = cur_pos;
ep_temp = ep_square;
i_alpha = alpha;
```

```
/* perform check extensions if we haven't gone past maxdepth: */
if (in_check ()) {
  if (ply < maxdepth+1) extensions++;
}
/* if not in check, look into null moves: */
else {
  /* conditions for null move:
     - not in check
     - we didn't just make a null move
     - we don't have a risk of zugzwang by being in the endgame
     - depth is >= R + 1
     what we do after null move:
     - if score is close to
          -mated, we're in danger, increase depth
     - if score is >= beta, we can get an early cutoff and exit */
  if (do_null && null_red && piece_count >= 5 &&
        depth >= null_red+1) {
    /* update the rep_history just so things don't get funky: */
    rep_history[game_ply++] = cur_pos;
    fifty++;

    xor (&cur_pos, color_h_values[0]);
    xor (&cur_pos, color_h_values[1]);
    xor (&cur_pos, ep_h_values[ep_square]);
    xor (&cur_pos, ep_h_values[0]);

    white_to_move ^= 1;
    ply++;
    ep_square = 0;
    null_score = -search (-beta, -beta+1,
                          depth-null_red-1, FALSE);
    ep_square = ep_temp;
    ply--;
    white_to_move ^= 1;

    game_ply--;
    fifty--;

    xor (&cur_pos, color_h_values[0]);
    xor (&cur_pos, color_h_values[1]);
    xor (&cur_pos, ep_h_values[ep_square]);
    xor (&cur_pos, ep_h_values[0]);
    assert (cur_pos.x1 == compute_hash ().x1 &&
        cur_pos.x2 == compute_hash ().x2);
```

TeamLRN

```
    /* check to see if we ran out of time: */
    if (time_exit)
  return 0;

    /* check to see if we can get a quick
            cutoff from our null move: */
    if (null_score >= beta)
  return beta;

    if (null_score < -INF+10*maxdepth)
  extensions++;
  }
}

/* try to find a stable position before passing
        the position to eval (): */
if (!(depth+extensions)) {
  captures = TRUE;
  score = qsearch (alpha, beta, maxdepth);
  captures = FALSE;
  return score;
}

num_moves = 0;
no_moves = TRUE;

/* generate and order moves: */
gen (&moves[0], &num_moves);
order_moves (&moves[0], &move_ordering[0], num_moves, &h_move);

/* loop through the moves at the current node: */
while (remove_one (&i, &move_ordering[0], num_moves)) {

  make (&moves[0], i);
  assert (cur_pos.x1 == compute_hash ().x1 &&
      cur_pos.x2 == compute_hash ().x2);
  ply++;
  legal_move = FALSE;

  /* go deeper if it's a legal move: */
  if (check_legal (&moves[0], i)) {
    nodes++;
    score = -search (-beta, -alpha, depth-1+extensions, TRUE);
```

```
  no_moves = FALSE;
  legal_move = TRUE;
}

ply--;
unmake (&moves[0], i);
ep_square = ep_temp;
cur_pos = temp_hash;

/* return if we've run out of time: */
if (time_exit) return 0;

/* check our current score vs. alpha: */
if (score > alpha && legal_move) {

  /* update the history heuristic since we have a cutoff: */
  history_h[moves[i].from][moves[i].target] += depth;

  /* try for an early cutoff: */
  if (score >= beta) {
u_killers (moves[i], score);
store_hash (i_alpha, depth, score, l_bound, moves[i]);
return beta;
  }
  alpha = score;

  /* update the pv: */
  pv[ply][ply] = moves[i];
  for (j = ply+1; j < pv_length[ply+1]; j++)
pv[ply][j] = pv[ply+1][j];
  pv_length[ply] = pv_length[ply+1];
  }

}

/* check for mate / stalemate: */
if (no_moves) {
  if (in_check ()) {
    alpha = -INF+ply;
  }
  else {
    alpha = 0;
  }
}
```

TeamLRN

```
else {
  /* check the 50 move rule if no mate situation
          is on the board: */
  if (fifty > 100) {
    return 0;
  }
}

/* store our hash info: */
if (alpha > i_alpha)
  store_hash (i_alpha, depth, alpha, exact, pv[ply][ply]);
else
  store_hash (i_alpha, depth, alpha, u_bound, dummy);

  return alpha;

}
//--------------------
move_s think (void) {

  /* Perform iterative deepening to go further in the search */

  move_s comp_move, temp_move;
  int ep_temp, i, j;
  long int elapsed;

  /* see if we can get a book move: */
  comp_move = book_move ();
  if (is_valid_comp (comp_move)) {
    /* print out a pv line indicating a book move: */
    printf ("0 0 0 0 (Book move)\n");
    return (comp_move);
  }

nodes = 0;
qnodes = 0;
allow_more_time = TRUE;

/* allocate our time for this move: */
time_for_move = allocate_time ();

/* clear the pv before a new search: */
for (i = 0; i < PV_BUFF; i++)
  for (j = 0; j < PV_BUFF; j++)
    pv[i][j] = dummy;
```

```
/* clear the history heuristic: */
memset (history_h, 0, sizeof (history_h));

/* clear the killer moves: */
for (i = 0; i < PV_BUFF; i++) {
  killer_scores[i] = -INF;
  killer_scores2[i] = -INF;
  killer1[i] = dummy;
  killer2[i] = dummy;
  killer3[i] = dummy;
}

for (i_depth = 1; i_depth <= maxdepth; i_depth++) {
  /* don't bother going deeper if we've
        already used 2/3 of our time, and we
     have finished our mindepth search, since
        we likely won't finish */
  elapsed = rdifftime (rtime (), start_time);
  if (elapsed > time_for_move*2.0/3.0 && i_depth > mindepth)
    break;

  ep_temp = ep_square;
  temp_move = search_root (-INF, INF, i_depth);
  ep_square = ep_temp;

  /* if we haven't aborted our search on time,
        set the computer's move
     and post our thinking: */
  if (!time_failure) {
    /* if our search score suddenly drops, and
          we ran out of time on the
       search, just use previous results */
    comp_move = temp_move;
    last_root_score = cur_score;
    /* if our PV is really short, try to get some
          of it from hash info
       (don't modify this if it is a mate / draw though): */
    if (pv_length[1] <= 2 && i_depth > 1 &&
          abs (cur_score) < (INF-100) &&
        result != stalemate && result != draw_by_fifty &&
        result != draw_by_rep)
        hash_to_pv (i_depth);
    if (post && i_depth >= mindepth)
       post_thinking (cur_score);
}
```

TeamLRN

```
    /* reset the killer scores (we can keep the
          moves for move ordering for now, but the
          scores may not be accurate at higher depths, so we need
       to reset them): */
    for (j = 0; j < PV_BUFF; j++) {
      killer_scores[j] = -INF;
      killer_scores2[j] = -INF;
    }

  }

  /* update our elapsed time_cushion: */
  if (moves_to_tc) {
    elapsed = rdifftime (rtime (), start_time);
    time_cushion += time_for_move-elapsed+inc;
  }

  return comp_move;

}
```

COMMON AI ELEMENTS

Opponent AI

By definition, a zero-sum game must have an opponent to win or lose to. In an entertainment sense, this opponent must become another person and play the rules with some semblance of personality. For most games, this personality is simply represented by a difficulty rating and by playing against the program enough times at each rating, a human being will eventually determine the kinds of moves that the computer-controlled player will make and not make.

Helper AI

Consumer games like chess usually include a tutor mode, where the computer will run you through a number of drills and lessons to improve your game. Although some games only provide minimal tutoring content in the form of scripted lessons, others actually include intelligent help systems that see flaws in your game and can steer you to scripted lessons, or give advice about a board setup in real time. Many people buy chess products for this feature alone, because they want to learn or improve their games by getting instruction and practice from the AI system. Other games like *Bridge* that have somewhat large or confusing rule sets also use helper AI systems to teach the basic strategies of the game.

USEFUL AI TECHNIQUES

Finite-State Machine

Most of these games are fairly linear games (although some only have one basic state change, that of ending the game). The gameplay can be broken down into smaller parts (as in the opening, midgame, and endgame phases of chess), which are easily identifiable and can therefore allow the system to switch between different AI methods based on these sub states.

Alpha-Beta Search

This is pretty much the de facto standard for search in classical games that need minimax trees searched. Minimax trees are specially set up game state trees, with the layers of the tree comprising nodes representing the choices each player can make, and where the values associated with each node of the tree depict its closeness to a winning value (see Figure 12.3 for a simplified example of a minimax tree). The algorithm then follows that at each choice, the first opponent moves with the max score at his level of the tree, and the other player plays the minimum scored at his. This is because the first player is trying to maximize his score, and the second player is trying to minimize the first player's score. This technique leads to an optimized move direction for these types of games, but has the problem of assuming a completely defensive second player.

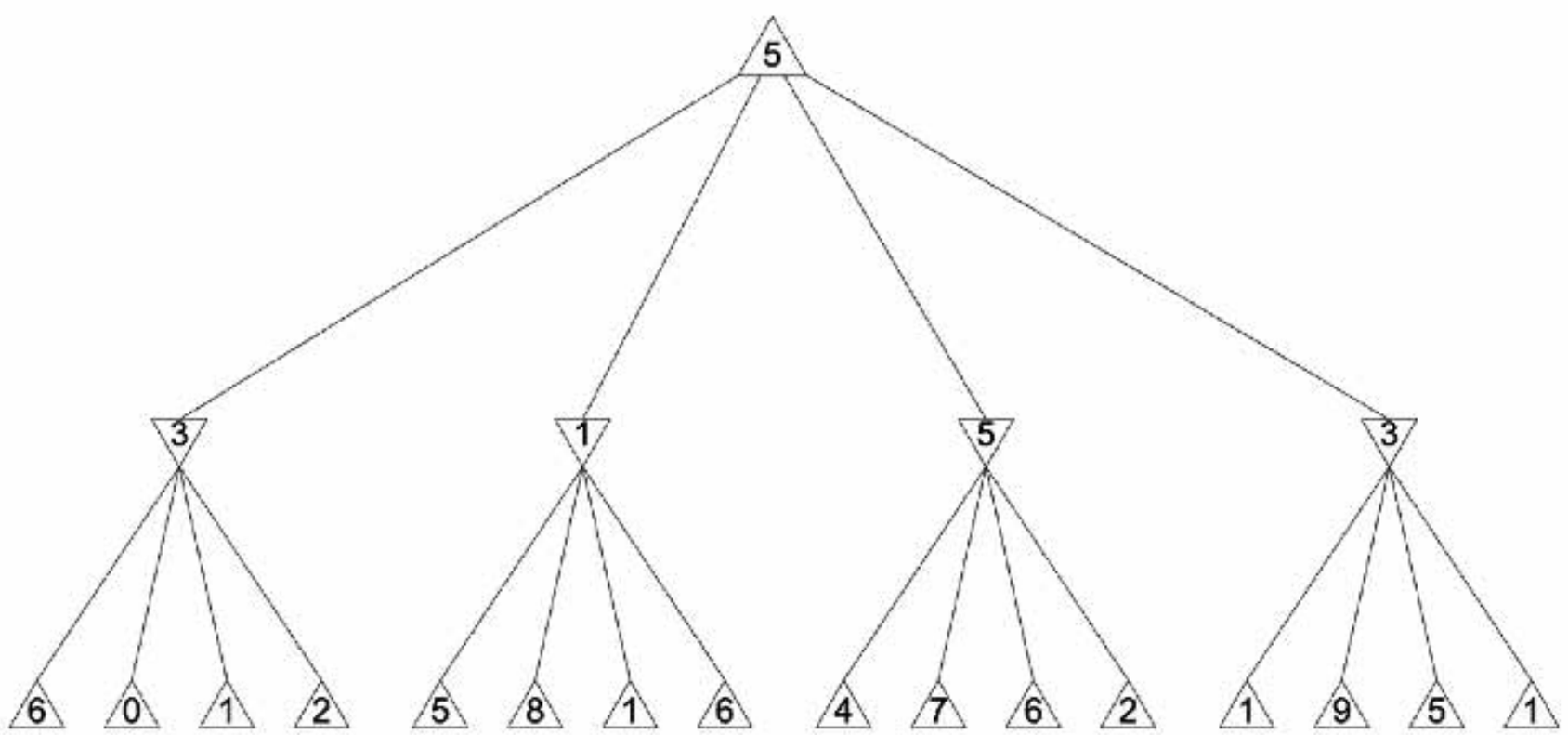


FIGURE 12.3 Simplified example of a minimax search tree showing one turn for each player, or one "ply."

TeamLRN

Minimax methods can be extended to games that also contain an element of pure chance, such as backgammon. This extension is called an expectimax tree and merely adds the element that a pure minimum and maximum value cannot be calculated at each tree node, thus introducing *chance* nodes that use an estimate of the random values that are being introduced into the game.

The problem with full minimax search is that it takes into account the whole tree. Consider chess, for which, at any given board position, there are usually about 35 legal moves. This means that a 1-level search is 35 entries, 2 levels is 352, 6 levels is almost two billion entries, and a 10-level search (which is in reality only 5 moves per player) is more than two quadrillion tree nodes. It is important to search as deeply as possible (average human players can usually make decisions based on looking 6 to 8 moves ahead, and grandmaster players sometimes make decisions 10 to 20 moves ahead) and alpha-beta search allows us to prune whole tree branches with total safety, so this vastly reduces the number of comparisons to perform, unless you get unlucky enough to have your game state tree set up in the worst-case scenario, which would mean that the optimization would be completely nulled out, and you would end up performing a regular minimax search.

Neural Nets (NNs)

Strategy games with larger state spaces or somewhat strange evaluations of board positions (such as *Go,* in which most of the position scoring involves very esoteric things like "influence" and "territory") have lent themselves well to the kinds of esoteric knowledge that can be stored in NNs. However, this kind of data structure is fiendishly hard to train, and even harder to debug. It is used in these sorts of situations because nothing else will really do the job.

Genetic Algorithms (GAs)

GAs can be considered another type of search, the so-called *random walk* search. This means searching the state space for solutions using some form of guided randomness. In this case, we use natural selection as our guide, and random mutation as our random element. We will discuss more of the specifics of this family of algorithms in Part IV of this book.

EXCEPTIONS

Most strategy games play straightforwardly, without malice or bias. However, some people have crafted their games to have some semblance of personality, such as *Checkers with an Attitude,* from Digenetics™, a game using various neural nets to play a very good, and distinctly personable, game of checkers.

EXAMPLES

Chess computer programs have been with us since the creation of the computer, with the earliest being in the early 1950s, and are still among the more popular classic games played as an entertainment. Early commercial games, like *Sargon,* weren't terribly intelligent and ran quite slowly. Nowadays, chess games have actually improved to the point where you can go to the store and buy a grandmaster level, very fast chess program to play against for less than $30. Over the years, some companies have tried to mix up the formula, while still keeping the same game, such as *Battle Chess* from 1988, which showed animated death sequences whenever you took an opponent's piece.

AREAS THAT NEED IMPROVEMENT

Creativity

Extended use of GAs might lead these types of AI opponents to find increasingly nonintuitive solutions, which GAs are strong for. Also, different heuristic-based searches could be implemented with NNs or GAs determining the heuristic, again so that creative local solutions could be found.

Speed

Speed is always an overriding factor in game AI programming, especially so in strategy games, which may entail tremendous amounts of searching. By improving our brute force methods, we may eventually find clever ways of arriving at decisions, without taking the time necessary to search massive trees to find the best solution. Or, the computers will just get so fast that the optimal search can be done trivially, and we'll take our AI somewhere else to play.

SUMMARY

Classic strategy games were some of the first to use academic AI techniques to build opponents because they represent the ideal candidate for AI directed-search methods. Strategy games have shown the entertainment industry the benefit of using real AI solutions for these types of problems (and for far less ideal situations like video games) and have even provided us with most of our data structures and methodologies.

- Classic strategy games are defined as being zero-sum games of perfect information, with mostly global moves, and are turn based.

TeamLRN

- The type of opponent AI you are coding is based on the type of game you are doing: a competition opponent requires optimal performance, but an entertainment opponent must use difficulty settings and such.
- Helper AI in entertainment strategy games is sometimes included for teaching and giving advice during practice games.
- FSMs can still be used in these games to break the state space into smaller chunks.
- Alpha-beta search is the primary opponent modeling means by which most classical strategy games consider opponent moves during planning.
- Genetic algorithms and neural nets can help facilitate directed search in new ways, or find unintuitive solutions.
- Creativity is a common lacking element in these games; they usually use more brute force in their search for the correct answer.
- Speed of the AI system is always a concern for these kinds of games because it represents the lion's share of the CPU time that the game is using.

13 Fighting Games

A strange mix of action and opponent puzzle genres, fighting games used to be *the* genre, especially in arcades. Early fighters, simple side-scrolling games with tough sounding names and main characters (sometimes referred to as "brawlers" or "beat-em-ups," like *Double Dragon, Bad Dudes,* and *Final Fight*) were more like horizontal scrolling shooter games where you used martial arts instead of projectiles. Other types of early brawlers included boxing games (like Nintendo's *Punch Out*) and wrestling competitions (*Pro Wrestling,* for the NES). All these games were popular, but fighting games were still just another genre.

However, the fighting genre reached the height of popularity in the early 1990s with the *Street Fighter 2: The World Warrior (SF2)* from Capcom (screenshot in Figure 13.1). *SF2* leapt onto the scene by taking the simple brawler formula and making the combat the entire experience, going over the top with concepts like combos, blocks, super moves, and in your face man-against-man action (although an earlier game, *Karate Champ,* did some of these things first, *SF2* did them all so much better that it stole the title of first real head-to-head fighter). Arcades moved all their other machines out, and lined up *SF2* machines. People everywhere got in line, "put their quarter up" on the ledge of the machine, and waited their turns. One important thing that *SF2* did was to reintroduce the concept of complex game controls to the game world. The special moves that *SF2* required of its advanced players were unlike anything the game world had seen before, and people loved being able to pull off monster combinations using complex hand movements that took days and weeks of practice.

The game proved to be so popular that it is usually credited with being a major reason that the Super Nintendo console finally caught up in sales to the Sega Genesis because the Super Nintendo version of *SF2* was a better version game, and fans couldn't get enough.

Fighters, like the other genres, gradually made the switch to 3D, but not all the way. While games like *Virtua Fighter* and *Tekken Tag Tournament* (screenshot in

TeamLRN

FIGURE 13.1 *Street Fighter 2: The World Warrier* screenshot. © Capcom Co., Ltd. Reprinted with permission.

Figure 13.2) carved niches for themselves using 3D combat methods, the *Street Fighter* series stayed in the 2D realm and, instead, created deeper systems of gameplay that couldn't be replicated in 3D because of the problems with cameras and the quick gameplay necessary to pull them off. These two disparate fighting game worlds still exist.

Wrestling games didn't really suffer from this transition to 3D, however. Most of wrestling involves grappling (by definition), so the types of character interactions becomes much larger, and you can set up very deep chains of moves as the characters move into and out of various locks and takedowns by initiating moves and counters. The characters are grappled together, so they don't have the problems of lining up with their opponent, and the camera can be more tightly positioned because of the proximity of the wrestlers.

In recent years, even though fighting games have fallen from their number one spot, they are still around and are invading other genres with games like *Buffy the Vampire Slayer,* which could be described as half fighter and half adventure game.

Such is the trend of all genres: start with a bang, develop until a level of maturity (and complexity) is reached where any additional improvement is incremental at best, languish in unpopularity for a while, and then in semidesperation merge with other genres to add content and flavor to the experience.

FIGURE 13.2 *Tekken Tag Tournament* screenshot. TEKKEN TAG TOURNAMENT® © 1994 1995 1996 1999 Namco Ltd., All rights reserved. Courtesy of Namco Holding Corp.

COMMON AI ELEMENTS

Enemies

The enemies in fighting games use some of the most heavily tuned and balanced opponent AI code ever written. One of the biggest selling points of most successful fighters was that the game was balanced, that is, that no one character was intrinsically easier to win with than any other. Some might be harder or easier to control, but with practice, you could be equally deadly with any of them. Because of this, precise control over individual characters' moves, down to the single frame of animation level was exercised. As such, most of these games use a form of scripting language that can describe events on a frame-by-frame basis, including sounds, turning on/off defensive and offensive collision spheres, marking points in the animation where branches are possible (for combos), and anything else the move might need to trigger. Character scripters would spend months working balance issues out of the game.

TeamLRN

In some fighting games, the background is more than just a backdrop and might contain elements that can be used in battle, or hidden behind, or simply smashed to receive some kind of powerup. Enemies in these games need to be able to react intelligently with these elements as well. For instance, an enemy approaching the main character for a fight has a big wooden crate sitting in between himself and his opponent. Does he use an avoidance system to move around it? Does he pick it up and throw it either out of the way or at his opponent? Does he jump over it? Or does he just smash through it with a huge punch? These are the kinds of advanced decisions that your enemies might have to make if you're working on a fighting game with background interactions.

Collision Systems

Collision systems at the character level are also supremely important to the fighting genre. Each character typically had a number of collision areas, each of which might change size for any given animation frame, or even be disabled for certain periods of time. To facilitate gameplay, the collisions were never really physics based but, rather, relied on tuned data that detailed such things as the amount of knock back felt by each player, the animation to play upon collision, a sound or effect to spawn, any "recover" time associated with the move (meaning, the amount of time after this move that I can't throw another move at all), and a host of other data values that the games needed.

Boss Enemies

Like RPGs and some other genres, fighting games use boss enemies to treat the player to a bigger, nastier enemy at the end of each level. In the 2D brawlers, these were sometimes the only memorable enemies in the whole game, another similarity to horizontal scrolling shooter games. Head-to-head fighting games traditionally only had one boss, the guy you had to fight after you defeated everybody else. This character was traditionally very tough to beat, the difficulty being much harder than whatever the rest of the game was set at.

Camera

In the 3D fighting games, you run into the problem of camera positioning, just as in 3D platformers. However, because of the fast-paced nature and camera-relative controls that the fighting character is using, the camera for 3D fighters needs special attention; otherwise, it will ruin the fighting game by messing up combos, causing moves to miss the target because of orientation problems, and generally make the game a mess to play.

Another difference from platform games is that the player really doesn't have the time to use a free-look camera because he's engaged in close-quarters combat. Also, because there are potentially two (or more) human players, a free-look camera wouldn't be viable from a control or visibility standpoint. Therefore, a good algorithmic or tracked camera system is essential.

Action and Adventure Elements

Some of the genre-crossing variants to the fighter are using more and more action or adventure game ingredients. Some involve heavy amounts of exploration and puzzle solving like adventure games do. But some also involve the jumping and climbing challenges of the platform game world. By blending in these additional game elements, developers are keeping the fighting game alive, while inventing new combinations of gameplay experiences that keep games fresh.

USEFUL AI TECHNIQUES

Finite-State Machines (FSMs)

Fighting games are usually state based, with the AI-controlled opponent doing a move, sitting there, or responding to a collision. A simple FSM can keep most fighter games in line and provide the developer with more than enough structure to add complexity without maintenance headaches. Usually the structure of the FSM is stored in a database of some kind, to facilitate the fact that during tuning and play testing, the state diagram of any given character might change dramatically.

Data-Driven Systems

Because of the huge number of characters, moves, blocks, throws, and combos, especially given the level of tuning and balance that needs to be given to these games, driving the primary fighting engine with designer-accessible scripting is really the only way to go. Usually, each move is scripted to allow very precise determination of attack, defense, combo branching, sound effects, collision times and size of collision area, as well as damage inflicted. The collision system is usually quite complex (even the first *SF2* game had many collision areas per enemy sprite, with separate head, arms, body, legs, etc.), with data tables detailing the animations to play if areas on the enemy are hit, or blocked, or whatever. Additional tables would describe the "personality" of each fighter, by listing out bias values on moves and combos, how aggressive the player was defensively, and just about everything else about the fighter.

TeamLRN

Scripting Systems

In addition to the notion that the designers need strict control over fighting animations (thus, they usually require a script to detail everything that needs to happen during each move), story elements and the like are still very prevalent. This occurs in some of the adventure-style fighting game variants especially, so scripting systems are used in fighters frequently.

Scripting systems are also useful for in-game cinematic moments, for example, when the fight starts and the characters enter the arena, or after someone wins and the winner plays some kind of victory dance. Hugely complex moves (sometimes called "super combos" or the like) might also require a level of scripting because super combos are usually constructed from other moves, all strung together in a specific fashion.

EXAMPLES

Early fighters were very simple affairs. You usually had a punch button, or maybe a punch and kick. Games in this realm were the side scrollers (or brawlers), such as *Bad Dudes, Kung-Fu Master, Golden Axe,* and *Ninja-Gaiden.* The enemies had very simple AI—usually they would just try to surround the player and throw whatever simple move or combination of moves they had in their arsenal. The side scrolling fighters had boss characters, but the bosses were usually just very fast, or had a lot of hit points, or some huge weapon; they were almost never smarter.

Then the head-to-head fighters started appearing, and were so popular that many different game franchises were started: *Samurai Showdown, King of Fighters, Mortal Kombat,* and of course, the *Street Fighter* series. As the years progressed, sequels continued to be better, more complex, and more technically enhanced games.

The AI-controlled enemies in head-to-head fighting games were usually completely fleshed out fighting opponents, with the full abilities of almost any human. The difficulty of the AI could be set by the operator (in the arcades) or by the player (on the home consoles) to fit any user skill level—everything from totally inept to almost invincible. This was only possible because in the course of constructing these games, with their finely tuned input windows, animation frame counting, and rigorously adjusted collision systems, the game developers allowed the entire system to be scaled up or down by raw difficulty as well as time scaling (for various turbo speed modes of play). The scripts and data associated with each move could handle sliding skill levels internally.

The 3D brawlers have also come a long way, with initial games like *Battle Area Toshinden* (the game that came with a lot of people's first Playstation console), all the way to the current brands: *Soul Calibur, Dead or Alive,* and the *Virtua Fighter*

games. These games use all the data-driven AI systems of their 2D brothers, but also use extensive camera work, and some even use a degree of pathfinding because of the advanced terrain usage.

Games like *Buffy the Vampire Slayer* (which used a popular license and lots of exploration challenges), *The Mark of Kri* (with its great integration of cinematics), and *Viewtiful Joe* (a throwback game that took today's advanced technology and married to it to a hardcore old-style brawler) are all examples of the use of heavy fighting systems in various other game types. All these games use some of the techniques used in regular fighters, along with a good amount of the AI challenges preset in mainstream action and adventure games.

AREAS THAT NEED IMPROVEMENT

Learning

Fighting games are like most video games; eventually, the human will find a weak point and exploit it repeatedly to make the game easier for himself. This was evident even in *SF2*, where continually jumping and doing a fierce punch over and over again could almost always defeat the usually very difficult character Zangief. If poor Zangief had even a smidgen of learning AI, he could have eventually seen the pattern of the human's attacks, and taken precautions. A learning system could also help with general case exploits and actually help keep the gameplay even (against the computer at least) by having the AI notice if the human is repeating a single, very powerful attack and circumventing it.

An AI set to lower difficulty could even help out the player by adjusting its attack patterns if some of its attacks were always hitting. In this way, the fight would be a bit more interesting, even if the human kept making the same mistakes.

SUMMARY

Fighting games, both 2D and 3D, give the player a level of character control that most other games do not. They appeal to both twitch gamers, as well as to tacticians who study the various blocking and attack systems looking for advantages.

- Fighting games started out as 2D side scrolling brawlers, with simple controls and very little strategy.
- Head-to-head fighters infused the genre with the depth of gameplay it needed to survive, and also made it the most popular genre for almost a decade.

TeamLRN

- Fighting game enemies and boss enemies require heavy tuning to preserve game balance, so this needs to be taken into account when coding them.
- The collision systems used in fighters are also very complex, requiring much higher resolution of targets then most games.
- The camera system (for 3D fighters), and any additional action or adventure elements may also require AI code.
- FSMs and scripting (or some other form of data-driven AI) constitute the most common means by which fighting game AI is created. Data driving a fighter is important due to the high amount of tuning and designer input that needs to occur at many levels of gameplay.
- Learning in fighting games could help against AI exploits and keep gameplay from becoming repetitive.

14 Miscellaneous Genres of Note

Although most games fall into the categories in the previous chapters, there are still many games that are either hard to categorize, or in a class all by themselves. This chapter will highlight some of the most notable of these games and will briskly discuss the artificial intelligence (AI) methodologies used in their creation.

CIVILIZATION GAMES

Civilization (or civ) games are turn-based strategy games. *Big* turn-based strategy games; sometimes with monstrous amounts of units to control, and hundreds of things for the player to manage and tweak on any given turn. Almost exclusively a PC genre (mostly because of interface concerns), there are a few console games of this type—*Final Fantasy Tactics* and even the handheld game *Advance Wars* are good examples. The genre is almost owned by one man, Sid Meier. He was designing a spin-off of the 1989 hit game *SimCity*™ (which will be discussed later, with God games) and two years later, managed to create an entirely new genre. The game was called *Civilization*, and has since spawned an entire series, as well as dozens of other civ games. The *Civilization* series (Figures 14.1 and 14.2 show the evolution from *Civilization* to *Civilization 3*), as well as the recent *Alpha Centauri* and many others, are all great civ games, with incredibly deep strategy, challenging AI systems, good interfaces, and almost infinite replay value. Some other great examples of civ games are *X-Com,* the *Heroes of Might and Magic* games, and the *Master of Orion* series.

The interface is called turn based, which means that players (a mix of humans and AI opponents) take turns issuing orders to their armies, cities, or whatever, and then watch the turn's total activities unfold. This process continues, back and forth, until the game is over. The player can control everything: which battles are instigated, what cities and towns are producing, what types of research are being studied,

TeamLRN

FIGURE 14.1 *Civilization* screenshot. Sid Meier's Civilization® and Sid Meier's Civilization® III courtesy of Atari Interactive, Inc. © 2004 Atari Interactive, Inc. All rights reserved. Used with permission.

what new inventions are having resources allocated to them, and so forth. These games can last a long time—many hours or even days. But, because of this turn-based mechanic, both sides have a much larger amount of time in which to make decisions, and so deep gameplay strategies can emerge. The concept of bounded optimality discussed in Chapter 1, "Basic Definitions and Concepts," really takes effect here; with the time restriction felt by more real-time AI systems all but lifted for the AI-controlled opponents of these games. Humans don't really enjoy waiting for the computer to make moves and decisions, so the AI engines for most civ style games do a lot of calculations while the human is performing his turn and, thus, can limit the amount of time taken for the computer opponent's turn.

Unlike real-time strategy (RTS) games, there is very little unit-based intelligence. Almost all decisions are strategic, with the conflicts between individual combat units (or even between units and defended cities) being reduced to random rolls based on the unit's strength and defense numbers. This leads to more of a simulation feel, rather than the action element that individual combat adds to the RTS genre.

Typical AI systems used in civ games are the following:

- Most of the same types of AI methods required by RTS games, including finite-state machines (FSM), fuzzy-state machines (FuSMs), hierarchical AI systems, good pathfinding, and messaging systems.

FIGURE 14.2 *Civilization 3* screenshot. Sid Meier's Civilization® and Sid Meier's Civilization® III courtesy of Atari Interactive, Inc. © 2004 Atari Interactive, Inc. All rights reserved. Used with permission.

- Civ games also borrow most of the support systems also used by RTS games, including terrain analysis, resource management, city planning techniques, and opponent modeling.
- A heavy data-driven element because of the number of civilization types (as well as the many types of units, technologies, resources, etc.) usually represented in these games.
- Robust planning algorithms because these games usually have expansive technology trees and huge game worlds. See Listing 14.1 for a very small sample of AI code from *FreeCiv*, an open source recreation of *Civilization*. *FreeCiv* has a huge following and has been ported to many platforms.
- Advanced AI systems for counselors and diplomacy. Many of these games have such a large amount of "work" to be done that some people would find it boring or tiresome to do everything, so the concept of counselors was introduced. These AI characters can offer to help the player with parts of the game that he finds tedious or confusing by offering advice when asked. This system would,

TeamLRN

in effect, use the AI decision-making engine to pass over the game world while the human is in control, and then inform the human what the computer would do right now, as a suggestion that can be taken or discarded. Typically, these counselors were specialized into the various parts of the game, such as trade, or research, or government. In that way, the player only needs to consult those counselors that he wants to and can ignore everybody else. Diplomacy systems are also much more complex. Different groups will make alliances, and leaders might manipulate, outright lie, or hold grudges. The states of mind of these diplomatic types varies greatly during a game, and satisfying everybody is really not possible, just like in real life. In fact, in the original *Civilization,* it is all but impossible to run an entirely bloodless game, where the civs all live in peace and prosperity until someone wins through technical superiority. Does Sid know something about human nature?

LISTING 14.1 Sample AI Code from *FreeCiv*

```
/**************************************************************************
   Buy and upgrade stuff!
**************************************************************************/
static void ai_spend_gold(struct player *pplayer)
{
  struct ai_choice bestchoice;
  int cached_limit = ai_gold_reserve(pplayer);

  /* Disband troops that are at home but don't serve a purpose. */
  city_list_iterate(pplayer->cities, pcity) {
    struct tile *ptile = map_get_tile(pcity->x, pcity->y);
    unit_list_iterate(ptile->units, punit) {
      if (((unit_types[punit->type].shield_cost > 0
            && pcity->shield_prod == 0)
           || unit_has_role(punit->type, L_EXPLORER))
          && pcity->id == punit->homecity
          && pcity->ai.urgency == 0
          && is_ground_unit(punit)) {
        struct packet_unit_request packet;
        packet.unit_id = punit->id;
        CITY_LOG(LOG_BUY, pcity,
                 "disbanding %s to increase production",
                 unit_name(punit->type));
        handle_unit_disband(pplayer, &packet);
      }
    } unit_list_iterate_end;
  } city_list_iterate_end;
```

```
do {
  int limit = cached_limit; /* cached_limit is our gold reserve */
  struct city *pcity = NULL;
  bool expensive; /* don't buy when it costs x2 unless we must */
  int buycost;

  /* Find highest wanted item on the buy list */
  init_choice(&bestchoice);
  city_list_iterate(pplayer->cities, acity) {
    if (acity->anarchy != 0) continue;
    if (acity->ai.choice.want > bestchoice.want &&
                                    ai_fuzzy(pplayer, TRUE))
        {
      bestchoice.choice = acity->ai.choice.choice;
      bestchoice.want = acity->ai.choice.want;
      bestchoice.type = acity->ai.choice.type;
      pcity = acity;
    }
  } city_list_iterate_end;

  /* We found nothing, so we're done */
  if (bestchoice.want == 0) break;

  /* Not dealing with this city a second time */
  pcity->ai.choice.want = 0;

  ASSERT_REAL_CHOICE_TYPE(bestchoice.type);

  /* Try upgrade units at danger location
        * (high want is usually danger) */
  if (pcity->ai.danger > 1) {
    if (bestchoice.type == CT_BUILDING &&
            is_wonder(bestchoice.choice)) {
      CITY_LOG(LOG_BUY, pcity,
                  "Wonder being built in dangerous position!");
    } else {
      /* If we have urgent want, spend more */
      int upgrade_limit = limit;
      if (pcity->ai.urgency > 1) {
        upgrade_limit = pplayer->ai.est_upkeep;
      }
      /* Upgrade only military units now */
      ai_upgrade_units(pcity, upgrade_limit, TRUE);
    }
  }
```

TeamLRN

```
/* Cost to complete production */
buycost = city_buy_cost(pcity);

if (buycost <= 0) {
  continue; /* Already completed */
}

if (bestchoice.type != CT_BUILDING
    && unit_type_flag(bestchoice.choice, F_CITIES)) {
  if (!city_got_effect(pcity, B_GRANARY)
      && pcity->size == 1
      && city_granary_size(pcity->size)
         > pcity->food_stock + pcity->food_surplus) {
    /* Don't build settlers in size 1
         * cities unless we grow next turn */
    continue;
  } else {
    if (city_list_size(&pplayer->cities) <= 8) {
      /* Make AI get gold for settlers early game */
      pplayer->ai.maxbuycost =
                              MAX(pplayer->ai.maxbuycost, buycost);
    } else if (city_list_size(&pplayer->cities) > 25) {
      /* Don't waste precious money buying settlers late game */
      continue;
    }
  }
} else {
  /* We are not a settler. Therefore we
       * increase the cash need we
   * balance our buy desire with to
       * keep cash at hand for emergencies
   * and for upgrades */
  limit *= 2;
}

/* It costs x2 to buy something with no shields contributed */
expensive = (pcity->shield_stock == 0)
            || (pplayer->economic.gold - buycost < limit);

if (bestchoice.type == CT_ATTACKER
    && buycost > unit_types[bestchoice.choice].build_cost * 2) {
   /* Too expensive for an offensive unit */
   continue;
}
```

```
if (!expensive && bestchoice.type != CT_BUILDING
    && (unit_type_flag(bestchoice.choice, F_TRADE_ROUTE)
        || unit_type_flag(bestchoice.choice, F_HELP_WONDER))
    && buycost < unit_types[bestchoice.choice].build_cost * 2) {
  /* We need more money for buying caravans. Increasing
     maxbuycost will increase taxes */
  pplayer->ai.maxbuycost = MAX(pplayer->ai.maxbuycost, buycost);
}

/* FIXME: Here Syela wanted some code to check if
 * pcity was doomed, and we should therefore attempt
 * to sell everything in it of non-military value */

if (pplayer->economic.gold - pplayer->ai.est_upkeep >= buycost
    && (!expensive
        || (pcity->ai.grave_danger != 0 &&
                assess_defense(pcity) == 0)
        || (bestchoice.want > 200 && pcity->ai.urgency > 1))) {
  /* Buy stuff */
  CITY_LOG(LOG_BUY, pcity, "Crash buy of %s for %d (want %d)",
           bestchoice.type != CT_BUILDING ?
                 unit_name(bestchoice.choice)
           : get_improvement_name(bestchoice.choice), buycost,
           bestchoice.want);
  really_handle_city_buy(pplayer, pcity);
} else if (pcity->ai.grave_danger != 0
           && bestchoice.type == CT_DEFENDER
           && assess_defense(pcity) == 0) {
  /* We have no gold but MUST have a defender */
  CITY_LOG(LOG_BUY, pcity,
           "must have %s but can't afford it (%d < %d)!",
     unit_name(bestchoice.choice),
           pplayer->economic.gold, buycost);
try_to_sell_stuff(pplayer, pcity);
if (pplayer->economic.gold - pplayer->ai.est_upkeep >=
         buycost) {
  CITY_LOG(LOG_BUY, pcity,
                  "now we can afford it (sold something)");
  really_handle_city_buy(pplayer, pcity);
}
if (buycost > pplayer->ai.maxbuycost) {
  /* Consequently we need to raise more money through taxes */
  pplayer->ai.maxbuycost =
                          MAX(pplayer->ai.maxbuycost, buycost);
```

TeamLRN

```
    }
  }
} while (TRUE);

/* Civilian upgrades now */
city_list_iterate(pplayer->cities, pcity) {
  ai_upgrade_units(pcity, cached_limit, FALSE);
} city_list_iterate_end;

if (pplayer->economic.gold + cached_limit <
          pplayer->ai.maxbuycost) {
  /* We have too much gold! Don't raise taxes */
  pplayer->ai.maxbuycost = 0;
  }

  freelog(LOG_BUY, "%s wants to keep %d in reserve (tax factor %d)",
          pplayer->name, cached_limit, pplayer->ai.maxbuycost);
}
#undef LOG_BUY

/**************************************************************************
 cities, build order and worker allocation stuff here..
**************************************************************************/
void ai_manage_cities(struct player *pplayer)
{
  int i;
  pplayer->ai.maxbuycost = 0;

  city_list_iterate(pplayer->cities, pcity)
    ai_manage_city(pplayer, pcity);
  city_list_iterate_end;

ai_manage_buildings(pplayer);

city_list_iterate(pplayer->cities, pcity)
  military_advisor_choose_build(pplayer, pcity,
                                &pcity->ai.choice);
  /* note that m_a_c_b mungs the seamap, but we don't care */
  establish_city_distances(pplayer, pcity);
       /* in advmilitary for warmap */
  /* e_c_d doesn't even look at the seamap */
  /* determines downtown and distance_
        * to_wondercity, which a_c_c_b will need */
  contemplate_terrain_improvements(pcity);
```

```
  contemplate_new_city(pcity);
       /* while we have the warmap handy */
  /* seacost may have been munged if we found
        * a boat, but if we found a boat we don't rely on the seamap
        * being current since we will recalculate. - Syela */

city_list_iterate_end;

city_list_iterate(pplayer->cities, pcity)
  ai_city_choose_build(pplayer, pcity);
city_list_iterate_end;

ai_spend_gold(pplayer);

/* use ai_gov_tech_hints: */
for(i=0; i<MAX_NUM_TECH_LIST; i++) {
  struct ai_gov_tech_hint *hint = &ai_gov_tech_hints[i];

  if (hint->tech == A_LAST)
    break;
  if (get_invention(pplayer, hint->tech) != TECH_KNOWN) {
    pplayer->ai.tech_want[hint->tech] +=
    city_list_size(&pplayer->cities) * (hint->turns_factor *
                          num_unknown_techs_for_goal
                          (pplayer,
                           hint->tech) +
                          hint->const_factor);
    if (hint->get_first)
  break;
  } else {
    if (hint->done)
      break;
    }
  }
}
```

On October 28, 2003, Activision® released the source code for *Call to Power II*, an offshoot from the main *Civilization* line. The game has been heralded by its many fans for the level of extensibility it allows. It contains a very powerful scripting system (in fact, before the source was released, a number of actual bugs in the game code had clever game players creating script-based workarounds and distributing them on the Internet).

TeamLRN

GOD GAMES

Another genre that is unique and virtually owned only by a few franchises, is the God game. They are called "God games" because the player takes the role of creator, overseer, and the force of change for the entirety of the game, yet does not have direct control over the other inhabitants of the game. In some ways, this makes the experience much like an artificial life (alife) game, but on a much larger scale. Alife games are usually about molding just one creature (or maybe a few) by training and caring for them somewhat directly. God games give you more global control, affecting the lives of many, but more indirectly. The two fathers of the genre, Will Wright and Peter Molyneux, designed and created the earliest God games. Wright's game, released in 1987, is called *SimCity*™ (see Figures 14.4 and 14.5 for screens from *SimCity* and *SimCity 2000*™). *SimCity* was a real-time game, in which the player builds an ever-growing city and tries to keep the AI-controlled city inhabitants happy and healthy. In 1989, Molyneux released *Populous*™ (screenshot in Figure 14.3), which took the concept one step further by casting the player in the position of the Supreme Being over the land.

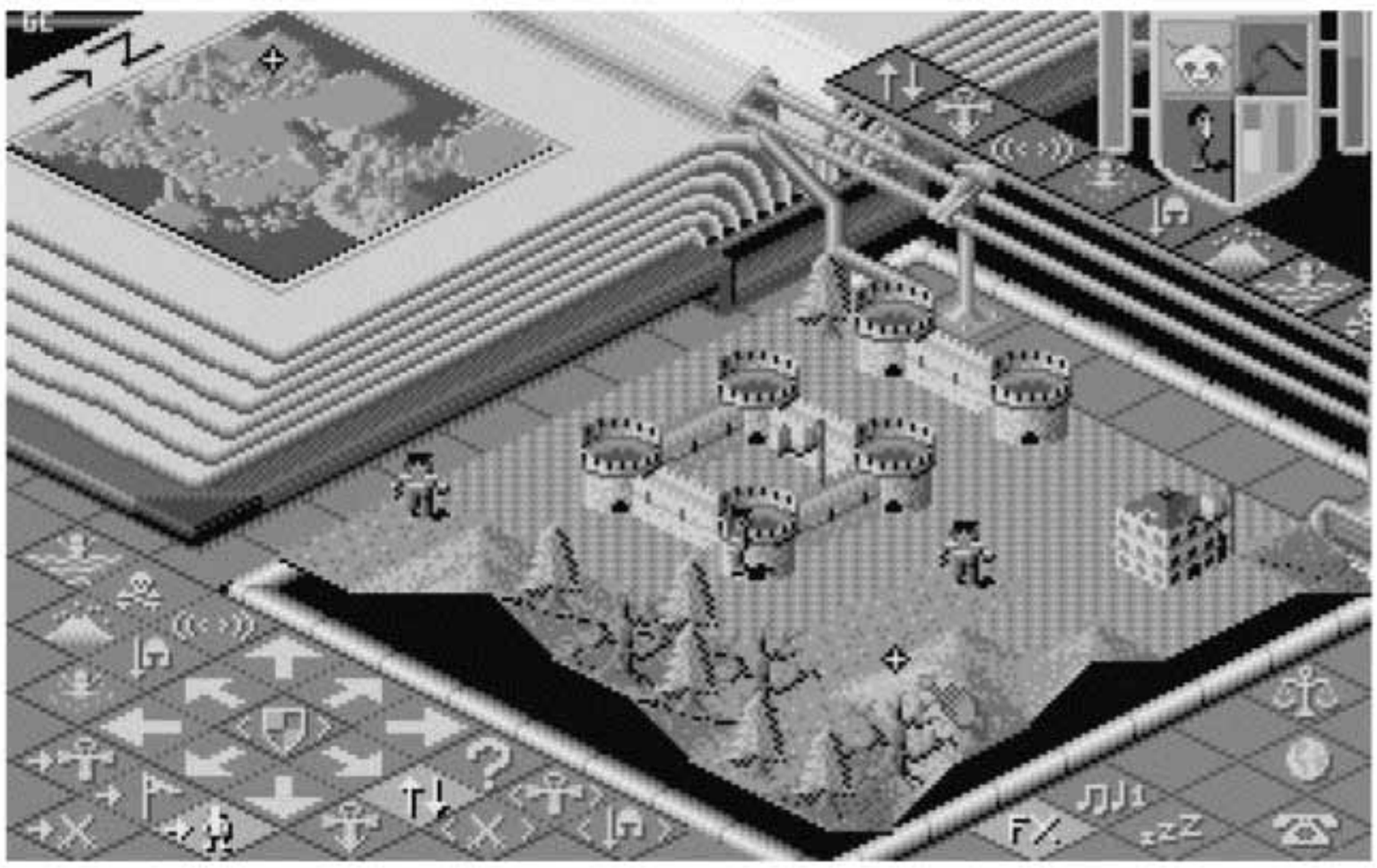


FIGURE 14.3 *Populous* screenshot. Populous, SimCity, SimCity 2000 and Ultima 7 screenshots © 2004 Electronic Arts Inc. Populous, SimCity, SimCity 2000, SimAnt, SimEarth, SimFarm, Dungeon Keeper, The Sims and Ultima are trademarks or registered trademarks of Electronics Arts Inc. in the U.S. and/or other countries. All rights reserved.

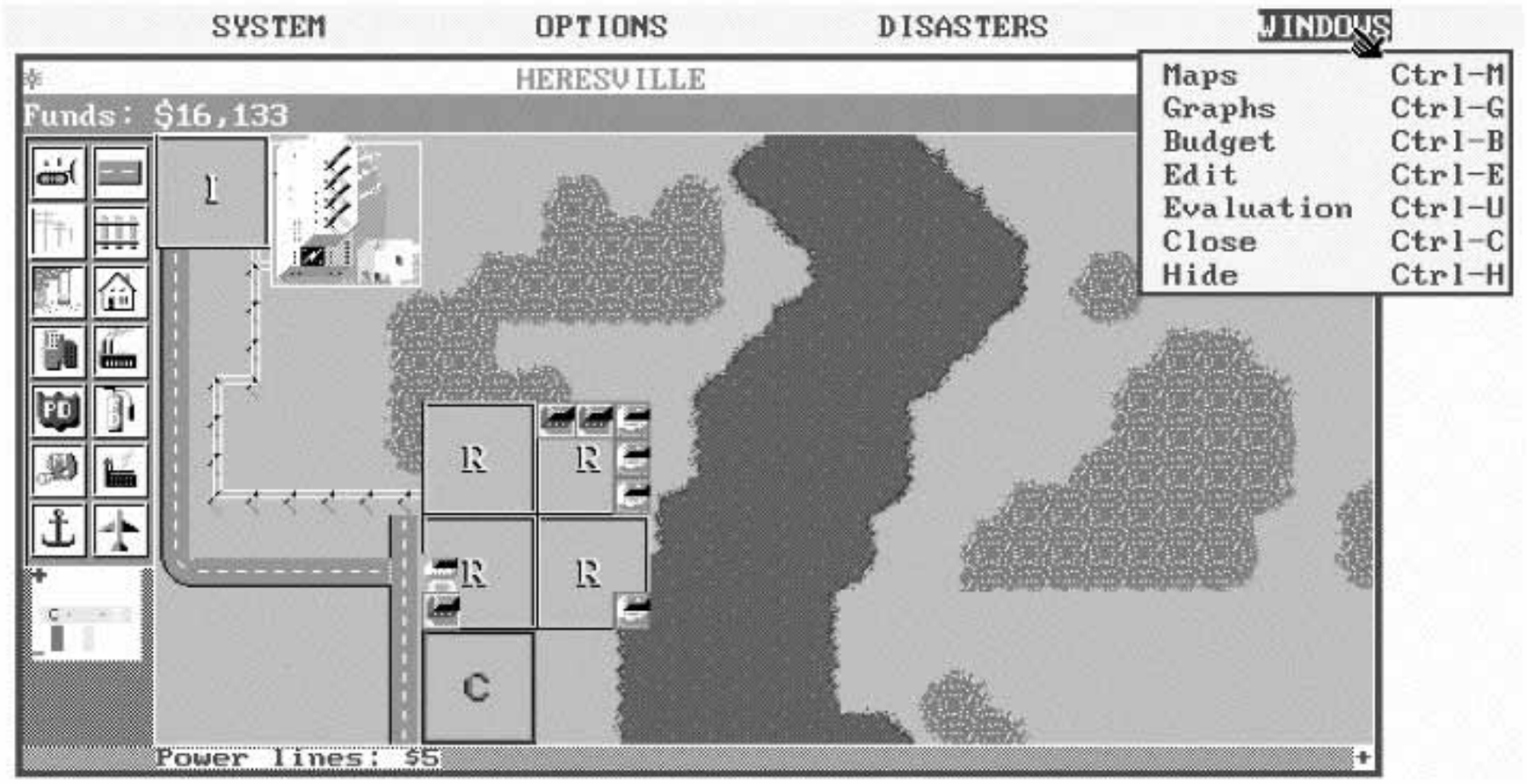


FIGURE 14.4 *SimCity* screenshot. Populous, SimCity, SimCity 2000 and Ultima 7 screenshots © 2004 Electronic Arts Inc. Populous, SimCity, SimCity 2000, SimAnt, SimEarth, SimFarm, Dungeon Keeper, The Sims and Ultima are trademarks or registered trademarks of Electronics Arts Inc. in the U.S. and/or other countries. All rights reserved.

The player could create and destroy land elements, could use god-scale powers to create plagues or volcanoes, and tried to get the inhabitants of the land to worship his side, giving him more power. Over the years, both Wright and Molyneux have both put out additional games in this genre, including *SimCity* variants (*SimAnt*™, *SimEarth*™, *SimFarm*™, etc.) from Wright's camp, and games like *Dungeon Keeper*™ and *Populous 2* from Molyneux. Both men are currently working on projects that have continued to evolve more into the alife genre and will be discussed later.

This style of game requires a good dose of strategic AI for the opponent God, if there is one. But in many of these games, especially the *SimCity* variants, there are no strategic AI systems at all. The human supplies all the strategic decisions for his side, and the "opponent" is merely the force of entropy, incrementally adding elements to the simulation that require player supervision, or constantly trying to tear down whatever structure, city, and so forth that the player is trying to build with random accidents, durability issues, increasing occupants, resources, and demands on the system, and the like.

All these games have one type of AI system in common, however—that of the somewhat autonomous characters that you are God over, be they humans, ants, or

TeamLRN

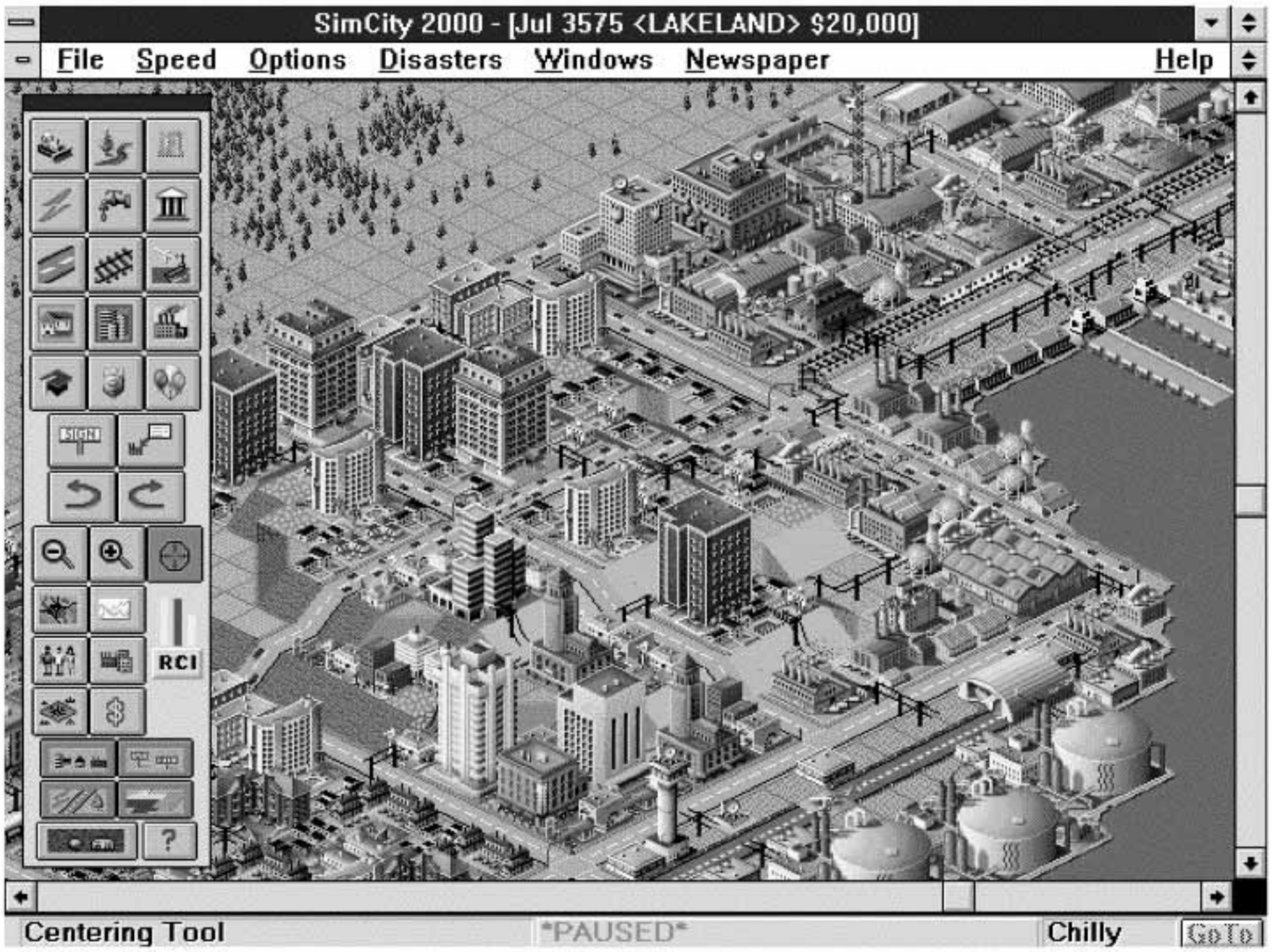


FIGURE 14.5 *SimCity 2000* screenshot. Populous, SimCity, SimCity 2000 and Ultima 7 screenshots © 2004 Electronic Arts Inc. Populous, SimCity, SimCity 2000, SimAnt, SimEarth, SimFarm, Dungeon Keeper, The Sims and Ultima are trademarks or registered trademarks of Electronics Arts Inc. in the U.S. and/or other countries. All rights reserved.

whatever. They are the beings that will inhabit and live under the light of your rule. Generally, these individual characters are brought into the game world as a collection of needs: each being needs X amount of food, Y amount of space, and Z amount of happiness (or the equivalent for any particular game). They will wander around, looking for ways in which to satisfy these needs, and if you have set up your city, world, or ant farm correctly, they will find it. If not, they get angry or leave, costing your simulation setbacks.

Typical AI systems used in God games are the following:

- Like civ games, this genre uses the same strategic AI systems as RTS games, but only if there is an opponent God that competes with you for followers or control of the world and that would require this kind of decision making ability.

- Autonomous characters most likely use a state-based system of needs. At the top level, each basic need would be tied to a state, such as `GetFood` or `GetAHouse`, the activation of which would be the perception that you were hungry, or homeless. The actions the character takes during each state would then get them the required resource, ending the perception that they need it, and thus, changing their state. A well-balanced game of this type will almost never have an autonomous character needing nothing, so he'll always be in a state of getting something, and always busy.
- The "world" AI level determines that the player's town is attractive enough so that more people would appear in it, or sets off random events to further challenge the player. This includes the so-called *rules* of the game, which in most games includes things like the physical laws as well as provisions for magic or respawning when you die. In God games, however, the rules might be the actual opponent that the player is competing with. So, the player must keep in mind rules such as "There must be 50 square feet of living space for each person in the city," and "For every 300 worshippers, you must build another temple," lest his control over the game start to slip away.

WAR GAMES

Not referring to the recent glut of war-themed FTPS games (like *Battlefield:1942* or *WW2Online*), this group instead pertains to the classic turn-based strategy war games with no or very indirect control of an economy to restock your armies. These games try to restage historic battles so that armchair generals can see if they have the same instincts as the professionals, or could have even done it better. These games have always been a niche market, even in their original form, which were very complex board games. Avalon Hill is the company that created most of the better-known board games, and most of the successful computer war games have some basis, or are actually renditions of, the classic Avalon Hill games.

These games require much more realistic simulation than do regular strategy games because the historic recreation is the entire point, and if things don't act the way they did in real life, the game is unacceptable to the tiny niche market you are shooting for in the first place. Because of this, things like terrain traversal, line of sight calculations, realistic weather simulations, and statistical modeling of almost every angle of combat are paramount to the success of the war simulation.

Some examples of good war games include the *Combat Mission* games and the *Airborne Assault* series. Listing 14.2 shows a function, `buildObjective()`, from the open source project *Wargamer: Napoleon 1813*. The game, originally published in 1999 by Empire® Interactive, is a deep simulation of some of Napoleon's most

TeamLRN

famous battles and has been taken over by the open source community. The sample function is part of a higher-level system that the AI is using to determine strategic plans for the future.

LISTING 14.2 `buildObjective()` from *Wargamer: Napoleon 1813.* Distributed under the GNU license.

```
bool AIC_ObjectiveCreator::buildObjective(const AIC_TownInfo& tInfo)
{
#ifdef DEBUG
   d_sData->logWin("Assigning units to %s", d_sData->campData()->
                                              getTownName(tInfo.town()));
#endif

   /*
    * Pass 1:
    *    build list of units and keep track of SPs removed from
    *    other objectives
    *
    *    Ony units that would not destroy an objective with
    *    a higher townImportance can be used
    */

   std::map<ITown, int, std::less<int> > otherObjectives;
   std::vector<TownInfluence::Unit> allocatedUnits;

   SPCount spNeeded = d_townInfluence.spNeeded();
   SPCount spAlloced = 0;
   SPCount spToAllocate = d_sData->rand(spNeeded,
                          d_townInfluence.spAvailable());

   TownInfluence::Unit infUnit;
   while((spAlloced < spToAllocate) &&
      d_townInfluence.pickAndRemove(&infUnit))
   {
      ASSERT(infUnit.cp() != NoCommandPosition);

      if(infUnit.cp()->isDead())
         continue;

      AIC_UnitRef aiUnit = d_units->getOrCreate(infUnit.cp());
      TownInfluence::Influence unitInfluence = infUnit.influence();
      // friendlyInfluence.influence(aiUnit.cp());
```

```
      float oldPriority = d_townInfluence.effectivePriority(aiUnit);
      if(unitInfluence >= oldPriority)
      {
         SPCount spCount = aiUnit.spCount();

#ifdef DEBUG
         d_sData->logWin("Picked %s [SP=%d, pri=%f / %f]",
            (const char*) infUnit.cp()->getName(),
            (int) spCount,
            (float) unitInfluence,
            (float) oldPriority);
#endif

            /*
             * If it already has an objective
             * Then update the otherObjective list
             */

            AIC_Objective* oldObjective = aiUnit.objective();
            if(oldObjective)
            {
               ITown objTown = oldObjective->town();

               if (spAlloced > spNeeded)
               {

#ifdef DEBUG
                  d_sData->logWin("Not using %s from %s because we already
have enough SPs",
                     (const char*) infUnit.cp()->getName(),
                     (const char*) d_sData->campData()->
                                 getTownName(objTown));
#endif
                  continue;
               }


               if (objTown != tInfo.town())
               {
                  const AIC_TownInfo& objTownInf =
                              d_towns->find(objTown);
                  if(objTownInf.importance() >= tInfo.importance())
                  {
                     int* otherCount = 0;
```

TeamLRN

```
                if(otherObjectives.find(objTown) ==
                   otherObjectives.end())
                {
                   otherCount = &otherObjectives[objTown];
                   *otherCount = oldObjective->spAllocated() -
                                 oldObjective->spNeeded();
                }
                else
                   otherCount = &otherObjectives[objTown];

                if(*otherCount >= spCount)
                   *otherCount -= spCount;
                else
                {
#ifdef DEBUG
                      d_sData->logWin("Can not use %s because it would
break objective at %s",
                         (const char*) infUnit.cp()->getName(),
                         (const char*) d_sData->campData()->
                                       getTownName(objTown));
#endif
                      continue;
                }
             }
          }
       }

       allocatedUnits.push_back(infUnit);
       spAlloced += spCount;
    }
 }

 if (spAlloced < spNeeded)
 {
#ifdef DEBUG
    d_sData->logWin("Can not be achieved without breaking more
important objective");
#endif
    return false;
 }

 /*
  * Assign the allocated Units to objective
  */
```

```
   Writer lock(d_objectives);

   AIC_Objective* objective = d_objectives->
                              addOrUpdate(tInfo.town(),
tInfo.importance());
   ASSERT(objective != 0);
   if(objective == 0)   //lint !e774 ... always true
      return false;

#ifdef DEBUG
   d_sData->logWin("Creating Objective %s", d_sData->campData()->
getTownName(tInfo.town()));
   d_sData->logWin("There are %d objectives", (int)d_objectives-
>size());
#endif

   objective->spNeeded(spNeeded);

   for (std::vector<TownInfluence::Unit>::iterator it =
            allocatedUnits.begin();
        it != allocatedUnits.end();
        ++it)
   {
      const TownInfluence::Unit& infUnit = *it;

      AIC_UnitRef aiUnit = d_units->getOrCreate(infUnit.cp());
      TownInfluence::Influence unitInfluence = infUnit.influence();
      // friendlyInfluence.influence(aiUnit.cp());

#ifdef DEBUG
         d_sData->logWin("Adding %s",
            (const char*) infUnit.cp()->getName());
#endif

      // Remove unit from its existing Objective
      // Unless it is already attached to this one

      AIC_Objective* oldObjective = aiUnit.objective();

      if(oldObjective != objective)
      {
         if(oldObjective != 0)
```

TeamLRN

```
{
   // Remove Unit from Objective
   // If objective does not have enough SPs then
   // remove the objective

   removeUnit(infUnit.cp());
}

ASSERT(aiUnit.objective() == 0);

// Add it to the objective table

aiUnit.objective(objective);
   objective->addUnit(infUnit.cp());
}

// Set priority to a higher value to
// reduce the problem of objectives being
// created and destroyed too quickly.

const float PriorityObjectiveIncrease = 1.5;
aiUnit.priority(unitInfluence * PriorityObjectiveIncrease);
}

#ifdef DEBUG
   if(d_objectiveDisplay)
      d_objectiveDisplay->update();
   if(campaign)
      campaign->repaintMap();
#endif

   return true;
}
```

Typical AI systems used in war games are the following:

- The same level of strategic AI found in civ games is used, but in war games, the AI is focused more on direct combat experiences.
- Data-driven systems are often employed because most of these games have huge numbers of battles for the player to engage in, as well as numerous statistical details for each piece of equipment, tactical unit, and location.
- Scripting comes into play quite regularly, to accurately model unusual or signature battle movements and strategies that were used by specific commanders in particular battles.

FLIGHT SIMULATORS (SIMS)

Another niche market, flight sims, try to accurately model the piloting of specific planes and give the player a realistic cockpit view and all the controls he would use in the actual aircraft. The most popular example is the *Microsoft Flight Simulator,* which originally came out in 1982 and is still going strong today. Even though pure flight sims have no real AI (you are basically fighting gravity, trying not to crash), some variants to the flight sim model were released, in an attempt to make a more mass appeal game. Some of the most famous of these were the Star Wars–based games, such as *X-Wing* and *Tie Fighter.* Both of these games were much lighter on their flight sim elements (there were only a handful of cockpit controls, and you were flying in outer space, so you didn't have stalls or strange atmospheric disturbances), but they gave the player enough of a simulation that they really immersed the player in the Star Wars world and gave many more people a taste for the flight sim experience than had ever tried it before. The *Wing Commander* series was also in this category. Other games, like *Descent,* took the flight sim to the world of the FTPS game because it was like deathmatch play with flying vehicles. The *Privateer* and *Freelancer* games added a full story to a light flight sim, and did very well. Also in this grouping are the numerous war-based flight sims, where you get to perform historic missions, just like in war games, but from the cockpit of one of the planes involved, for a much more personal feel.

Typical AI systems used in flight sims are the following:

- The pure flight sims have no competitive AI elements—you are simply fighting the forces of physics, mostly gravity and aerodynamics, to keep control over an aircraft. Some of these games do have a form of AI system for teaching the player how to pilot the plane, but it is usually just scripted sequences to show the various aircraft systems and abilities. Listing 14.3 shows the main AI loop for the open source flight sim project *FlightGear,* which has simple AI elements that will engage in dogfights with you.
- Action-oriented flight sims are like action racing games in that they need AI systems that can competently handle the vehicles of the game, as well as deal with the additional elements (combat, using powerups, etc.) that the game brings. These games might also include land-based AI-controlled enemies and require additional functionality beyond simple vehicular control. These games are much like other complex, genre-combining games and use a mixture of FSMs, messaging, and scripting.

LISTING 14.3 Main AI Loop from *FlightGear.* Distributed by the GNU license.

```
void FGAIAircraft::Run(double dt) {
```

TeamLRN

```
FGAIAircraft::dt = dt;

double turn_radius_ft;
double turn_circum_ft;
double speed_north_deg_sec;
double speed_east_deg_sec;
double ft_per_deg_lon;
double ft_per_deg_lat;
double dist_covered_ft;
double alpha;

// get size of a degree at this latitude
ft_per_deg_lat = 366468.96 - 3717.12 *
                 cos(pos.lat()/SG_RADIANS_TO_DEGREES);
ft_per_deg_lon = 365228.16 * cos(pos.lat() /
                      SG_RADIANS_TO_DEGREES);

// adjust speed
double speed_diff = tgt_speed - speed;
if (fabs(speed_diff) > 0.2) {
  if (speed_diff > 0.0) speed += performance->accel * dt;
  if (speed_diff < 0.0) speed -= performance->decel * dt;
}

// convert speed to degrees per second
speed_north_deg_sec = cos( hdg / SG_RADIANS_TO_DEGREES )
                       * speed * 1.686 / ft_per_deg_lat;
speed_east_deg_sec  = sin( hdg / SG_RADIANS_TO_DEGREES )
                       * speed * 1.686 / ft_per_deg_lon;

// set new position
pos.setlat( pos.lat() + speed_north_deg_sec * dt);
pos.setlon( pos.lon() + speed_east_deg_sec * dt);

// adjust heading based on current bank angle
if (roll != 0.0) {
  turn_radius_ft = 0.088362 * speed * speed
                    / tan( fabs(roll) / SG_RADIANS_TO_DEGREES );
  turn_circum_ft = SGD_2PI * turn_radius_ft;
  dist_covered_ft = speed * 1.686 * dt;
  alpha = dist_covered_ft / turn_circum_ft * 360.0;
  hdg += alpha * sign( roll );
  if ( hdg > 360.0 ) hdg -= 360.0;
  if ( hdg < 0.0) hdg += 360.0;
}
```

```
// adjust target bank angle if heading lock engaged
if (hdg_lock) {
  double bank_sense = 0.0;
  double diff = fabs(hdg - tgt_heading);
  if (diff > 180) diff = fabs(diff - 360);
  double sum = hdg + diff;
  if (sum > 360.0) sum -= 360.0;
  if (fabs(sum - tgt_heading) < 1.0) {
    bank_sense = 1.0;
  } else {
    bank_sense = -1.0;
  }
  if (diff < 30) tgt_roll = diff * bank_sense;
}

// adjust bank angle
double bank_diff = tgt_roll - roll;
if (fabs(bank_diff) > 0.2) {
  if (bank_diff > 0.0) roll += 5.0 * dt;
  if (bank_diff < 0.0) roll -= 5.0 * dt;
}

// adjust altitude (meters) based on current vertical speed (fpm)
altitude += vs * 0.0166667 * dt * SG_FEET_TO_METER;
double altitude_ft = altitude * SG_METER_TO_FEET;

// find target vertical speed if altitude lock engaged
if (alt_lock) {
  if (altitude_ft < tgt_altitude) {
    tgt_vs = tgt_altitude - altitude_ft;
    if (tgt_vs > performance->climb_rate)
      tgt_vs = performance->climb_rate;
  } else {
    tgt_vs = tgt_altitude - altitude_ft;
    if (tgt_vs  < (-performance->descent_rate))
      tgt_vs = -performance->descent_rate;
  }
}

// adjust vertical speed
double vs_diff = tgt_vs - vs;
if (fabs(vs_diff) > 1.0) {
  if (vs_diff > 0.0) {
    vs += 400.0 * dt;
```

TeamLRN

```
            if (vs > tgt_vs) vs = tgt_vs;
          } else {
            vs -= 300.0 * dt;
            if (vs < tgt_vs) vs = tgt_vs;
          }
        }

        // match pitch angle to vertical speed
        pitch = vs * 0.005;

        //##############################//
        // do calculations for radar //
        //##############################//

        // copy values from the AIManager
        double user_latitude  = manager->get_user_latitude();
        double user_longitude = manager->get_user_longitude();
        double user_altitude  = manager->get_user_altitude();
        double user_heading   = manager->get_user_heading();
        double user_pitch     = manager->get_user_pitch();
        double user_yaw       = manager->get_user_yaw();
        double user_speed     = manager->get_user_speed();

        // calculate range to target in feet and nautical miles
        double lat_range = fabs(pos.lat() - user_latitude) *
                                    ft_per_deg_lat;
        double lon_range = fabs(pos.lon() - user_longitude) *
                                    ft_per_deg_lon;
        double range_ft  = sqrt(lat_range*lat_range +
                                    lon_range*lon_range );
        range = range_ft / 6076.11549;

        // calculate bearing to target
        if (pos.lat() >= user_latitude) {
           bearing = atan2(lat_range, lon_range) * SG_RADIANS_TO_DEGREES;
             if (pos.lon() >= user_longitude) {
                bearing = 90.0 - bearing;
             } else {
                bearing = 270.0 + bearing;
             }
        } else {
           bearing = atan2(lon_range, lat_range) * SG_RADIANS_TO_DEGREES;
             if (pos.lon() >= user_longitude) {
                bearing = 180.0 - bearing;
             } else {
```

```
        bearing = 180.0 + bearing;
    }
}

// calculate look left/right to target, without yaw correction
horiz_offset = bearing - user_heading;
if (horiz_offset > 180.0) horiz_offset -= 360.0;
if (horiz_offset < -180.0) horiz_offset += 360.0;

// calculate elevation to target
elevation = atan2( altitude_ft - user_altitude, range_ft )
                    * SG_RADIANS_TO_DEGREES;

   // calculate look up/down to target
   vert_offset = elevation + user_pitch;

/* this calculation needs to be fixed
   // calculate range rate
   double recip_bearing = bearing + 180.0;
   if (recip_bearing > 360.0) recip_bearing -= 360.0;
   double my_horiz_offset = recip_bearing - hdg;
   if (my_horiz_offset > 180.0) my_horiz_offset -= 360.0;
   if (my_horiz_offset < -180.0) my_horiz_offset += 360.0;
   rdot =(-user_speed * cos(horiz_offset * SG_DEGREES_TO_RADIANS ))
               + (-speed * 1.686 * cos( my_horiz_offset *
                                        SG_DEGREES_TO_RADIANS ));
*/

   // now correct look left/right for yaw
   horiz_offset += user_yaw;

   // calculate values for radar display
   y_shift = range * cos( horiz_offset * SG_DEGREES_TO_RADIANS);
   x_shift = range * sin( horiz_offset * SG_DEGREES_TO_RADIANS);
   rotation = hdg - user_heading;
   if (rotation < 0.0) rotation += 360.0;

}
```

RHYTHM GAMES

A popular genre of game that has recently been developed is the rhythm game. In some ways, this style of game is the video game equivalent to the handheld game

TeamLRN

Simon, where the player is supposed to repeat increasingly long sequences of a musical and visual pattern. The first rhythm game was the 1997 game *PaRappa The Rapper*, and since then games have included everything from singing, to playing various instruments, to dancing. They all follow the same *Simon* formula, for the most part. These games are really puzzle games, but are much more patterned, so that players who continue to replay the games can get further and further along.

Although many of these games just have the player battling against the actual notes of the music, some do include opponents that are trying to out perform the player. Even *PaRappa* had a final freestyle stage to finish the game. But, the AI involved even in these opponents is at best very scripted. The script that is played might take into account the level of playing by the player, however, so that the opponent would step up to the challenge, as they say. But actual improvisational music using AI that would sample the types of things the human was doing and build on them with more complexity (like what humans do in real jam sessions) has definitely not been used in these games yet.

Typical AI systems used in rhythm games are the following:

- Scripting was used to match the AI-controlled character's movements and dialogue to the songs, as well as set up story elements.
- Data-driven gameplay, where a general lightshow system (or other visuals) might be tied to music analysis software, and a large number of songs are included with the game. Examples of this are *Vib Ribbon, Frequency,* and *Amplitude.*
- Some rhythm games had additional elements, like *Rez* (which was a scrolling shooter) and *Chu Chu Rocket* (a sort of puzzle or party game along the lines of *Bomberman*). These games used fairly simple state-based or scripted intelligence systems, which also worked with the music.

PUZZLE GAMES

Puzzle games are small, simple games of skill, which usually continue forever, but increase in difficulty over time. They usually have very simple interfaces, and even simpler descriptions of how to play. But, because of this simplicity, they are also some of the most addictive and widely played games in the world. It has been said that the main reason the Nintendo Gameboy became a worldwide phenomenon was because of a little game called *Tetris* (shown in Figure 14.6), and the most played computer game of all time is still *Freecell,* the card game that comes with Microsoft Windows. These games require very little of our attention, or time. You can play 10 minutes of a game, and then just shut it off. The very nature of these games allows the player to have a little taste of challenge, without having to commit himself to anything emotionally or time wise.

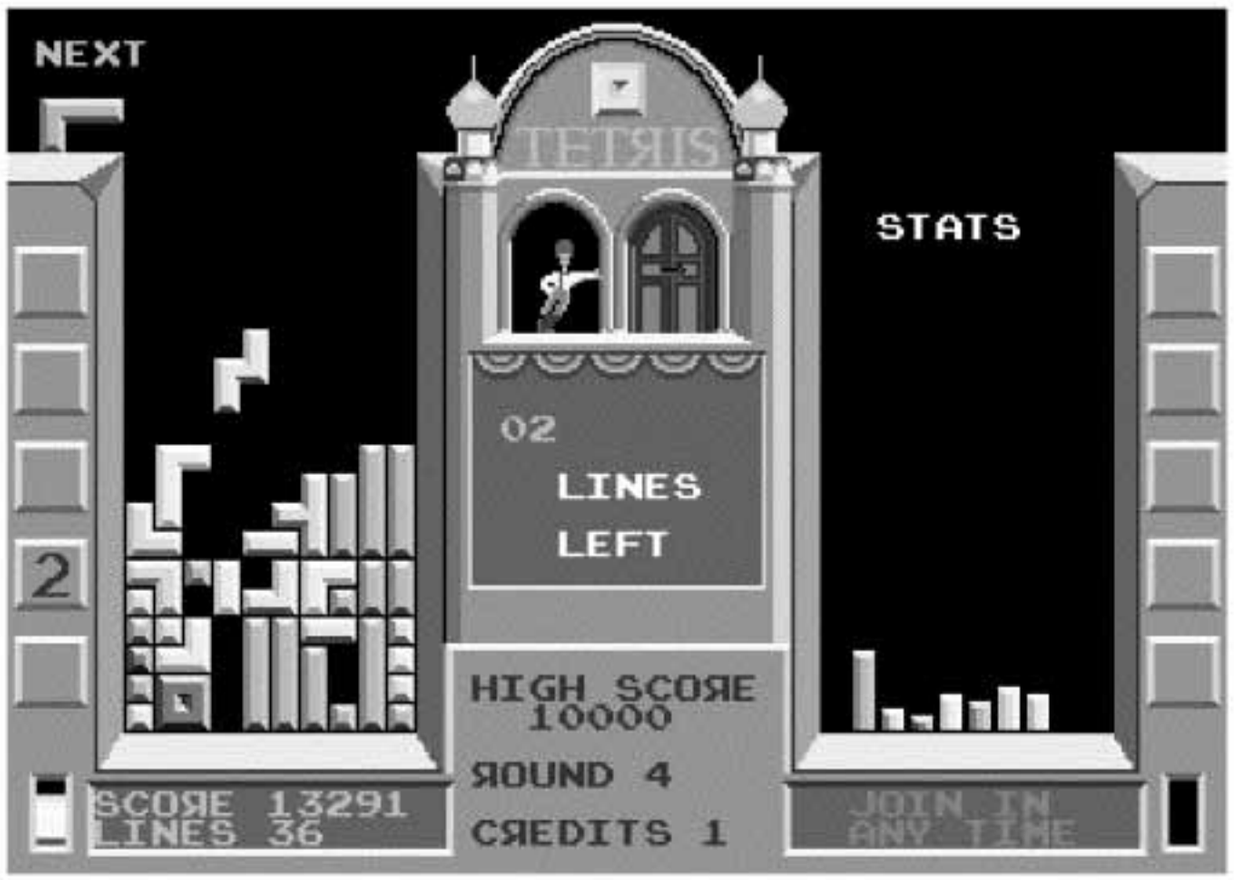


FIGURE 14.6 *Tetris* screenshot. Tetris®: Elorg 1987. Reprinted with permission.

Two areas have become major selling points for these games: the online world and cell phones and PDAs. Online, puzzle games make a lot of sense. You can code a puzzle game with minimal resources (perfect for keeping download speeds low) and allow people everywhere to come to your site to play the games for free, or next to nothing. This minimal game size also lends itself well to the space-restrictive world of cell phones and PDAs. People want some kind of distraction that they can use if they're stuck in an airport, or waiting for the bus, and most people have one of these devices already. It was a natural mix, once the hardware could support it. The bad news is that most puzzle games don't really use AI, the gameplay comprises simple patterns or specific setups that the player must overcome or unravel. However, some games do use AI in their games, such as PopCap's *Mummy Maze,* although it is usually very simple state-based behavior.

Typical AI systems used in puzzle games include the following:

- Very simple state-based behavior, if a game has any elements of AI usage at all.

ARTIFICIAL LIFE (ALIFE) GAMES

These titles are not considered games by some people, but are more like video game based pets of a sort. There are not many of these games, but some of them use some of the most cutting-edge game AI programming we have so far. These represent the

TeamLRN

pinnacle of exotic AI techniques in a real-time game experience. Other games in the alife genre are not so complex, AI wise, but represent an additional way of constructing AI systems to maximize traditionally difficult elements to model.

The first of these games were actually small electrical gadgets, called *Tamagotchi,* that were a huge craze in Japan. They were essentially small (key chain–sized) LCD-based units that had a lumpish looking creature pictured on it. The creature would demand to be fed, or to be petted, or whatever, based on a set of needs that it had. The human would then push the corresponding button that gave the creature what it wanted. If you failed to perform the correct tasks for too long, the creature might become angry with you, or even die. But if you did things right, it would flourish, and live a long, full life, all the while growing and getting small visual differences that people could use to differentiate their pets. Although this is a very strange concept by gaming standards, it was also a very popular one.

These toys eventually lead game developers to create video games using this premise. Some examples are *Seaman* (a game in which you are caretaker to a very rude fish with a man's head), the *Monster Rancher* games (which use random data from any CD to create unique creatures that you then train for battle), and the *Petz* games (pure *Tamagotchi*-style pets). Another series of products in this same line is the *Creatures* series developed by Cyberlife. These games are notable because of the actual systems they use to evolve their game characters. Whereas the other games use mostly some kind of advanced fuzzy-state machines (FuSMs), or just keep a lot of statistics about human interaction and hash that into large behavior lookup tables, the *Creatures* games have gone the high-tech route. Their games use advanced neural nets (NNs) to model learning and emotion and use a kind of genetic system to allow users to cross breed and evolve the creatures through genetic selection. The products are barely games, more like high-tech fish bowls, and even the developers consider it a technology demo. They are CPU intensive and have to run constantly for quite a bit to learn things, but they are quite impressive from a game AI standpoint.

Other types of alife games strive to make a bit more of a true game experience out of it, and this includes Wright's newest batch of games, *The Sims*™, as well as *Black & White,* from Molyneux.

In *The Sims,* the thing you are now controlling the simulation of is . . . a person's life. At the start of the game, you are given a Sim, a semiautonomous character that has a number of needs. Sims are semiautonomous in that they will perform need procurement to survive (if there's food around and he's hungry, the Sim will eat), but to really excel or progress, the human player has to basically baby-sit the Sim, getting it to perform its duties faster and more efficiently, and encouraging additional interactions, especially those with other Sims. The game has broken new ground by creating a simple AI paradigm known as *smart terrain.* In this concept, the agent has only basic needs that require fulfillment, is smart enough

to get around the world to reach things that can satisfy those needs, and has a fuzzy system that allows it to have some biases and rudimentary learning. But the true brains of the system are spread over the land by embedding it in the objects that populate the game world. Every object in the game that the Sims can interact with contains all the information about how this interaction will take place and what it will give the Sim, including the animation to play. In this way, new items can be added to the world at any time and can be instantly used by the Sims (which is easy to see, considering the number of expansion packs that have come out for the game). Because of its massive open endedness, its mass appeal because of (mostly) nonviolence, and the sheer customization and expansion capabilities of the game, *The Sims* has become one of the best-selling games of all time.

Black & White takes the God game concept (you have a small village of people that worship you, and you must take care of them), and adds an additional element: a totem animal. This character is controlled by a sophisticated (by game standards, anyhow) AI system, including dynamic rule building and decision tree creation, as well as the use of simple NNs (called perceptrons). This animal was supposed to learn most of it's behaviors directly from the user, and to facilitate this, the game allows a number of different ways for these totem animals to learn: by direct command, by observation, by reflection, and by direct feedback from the player (you could slap or stroke the creature, meaning he did something bad or good). By allowing the creature so many ways to learn things, and affect his beliefs and desires, the overall behavior set of the creature was very malleable, and thus, unique from creature to creature. It also led to more rapid learning than might be gained from any one method.

Typical AI systems used in alife games include the following:

- FuSMs are heavily used because they are easier to train and provide more directed behavior patterns.
- Neural nets are becoming increasingly researched and used, as developers find better and better ways to train them and to watch out for the wildly wrong behaviors they might cause.
- Genetic algorithms are being used in some of these games, facilitating breeding programs, and helping generations of game characters to evolve in various ways.
- A solid helping of standard game AI techniques are in use, including regular FSMs, messaging, and scripting.

TeamLRN

Part III

Basic AI Engine Techniques

In the following two parts of the book, the specific AI techniques present in today's game AI world will be discussed in detail.

This part will cover the more basic and commonly used techniques in the AI programmer's toolset. The following areas will be described for each methodology:

- A basic overview of the technique
- Explanation of the skeletal code
- Example implementation into the AIsteroids test bed
- Pros and cons of the method
- Commonly implemented extensions to the paradigm
- Optimizations to the technique
- How the method fits into the eight design considerations described in Chapter 2, "An AI Engine: The Basic Components and Design."

A portion of the code for each technique will be given in the text along with the explanation, but most of the code, as well as the project and makefiles, can be found on the CD-ROM.

TeamLRN

15 Finite-State Machines

In the world of game AI programming, no single data structure has been used more than the finite-state machine (FSM) (unless you count the switch statement as a data structure, perhaps, but a switch can actually be used as a simple form of state machine). This simple yet powerful organizational tool helps the programmer to break the initial problems into more manageable subproblems and allows programmers to implement intelligence systems with flexibility and scalability. Even if you haven't used a formal FSM class or functionality, you've probably used the principles that this structure follows because it is a basic way of thinking about software problems in general. Thus, even if your game uses a more exotic AI technique for some element of decision making, you will probably use some form of state system in your game.

FSM OVERVIEW

At its heart, a *state machine* is a data structure that contains three things: all the states inherent in this machine, several input conditions, and a transition function that serves as the lines of connectivity between these states. This book will somewhat shift some of these elements around by putting the transition logic into the actual states themselves. Thus, the machine will store the states, and update the current state, which may determine a transition is in order. This would then come back to the machine, which would instigate the transition. Figure 15.1 shows the differences between the classic system and the one used in this book. The reason for doing this is that it keeps the machine from becoming the repository of all the logic, with the states being simple data structures that denote connectivity. Instead, each state is a stand-alone module that has its update logic, transition logic, and special code such as enter and exit events.

Another difference between the classical implementation and the one this book will use is the transition system. In classic FSM methods, the transitions are expressed as pure events, usually an enumerated list of some kind, that the perception

TeamLRN

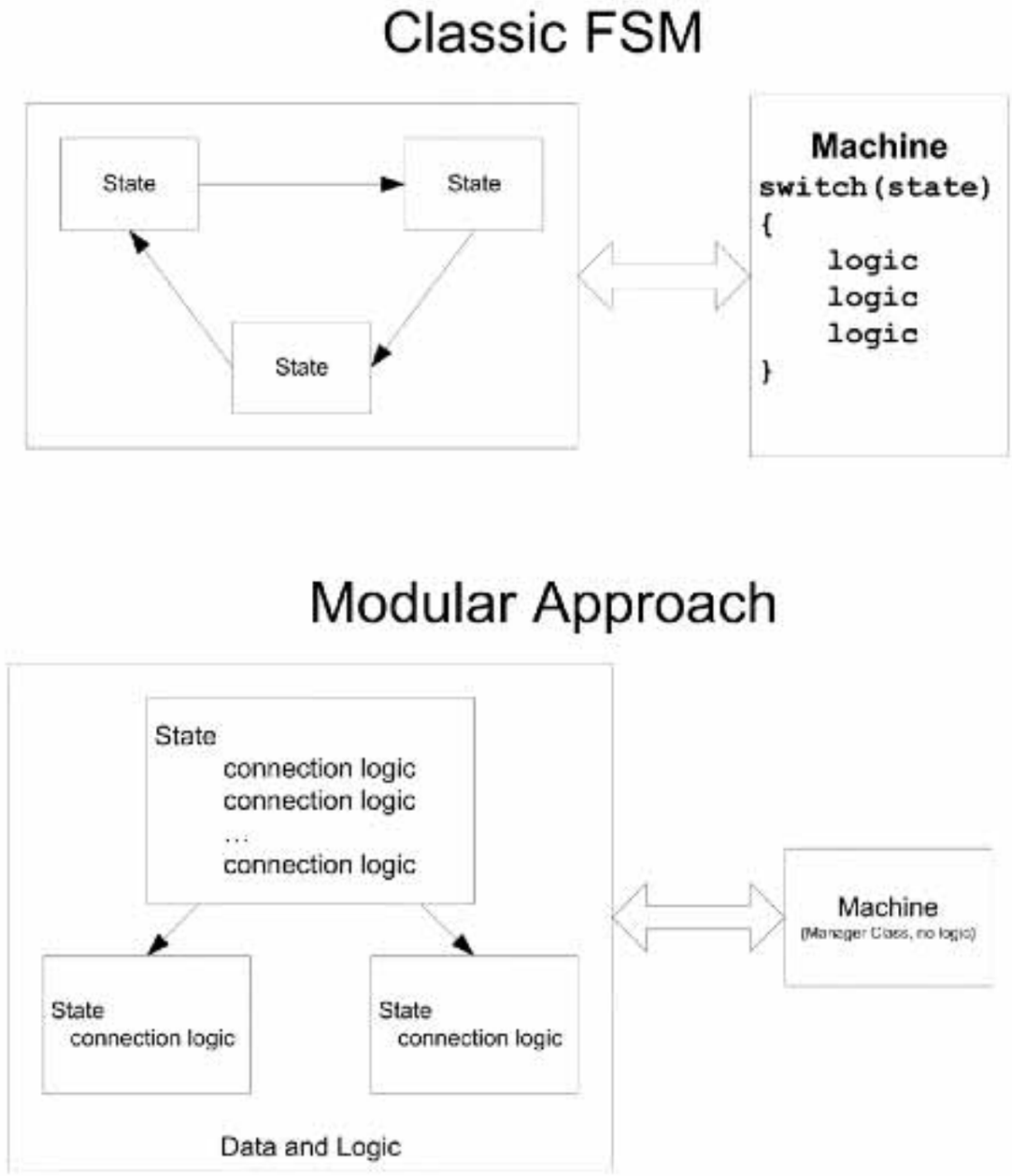


FIGURE 15.1 Comparison of execution flow between classic and modular FSMs.

system can use to trigger transitions. Each state then registers its transitions into a list constituting an input-output matching (e.g., `PLAYER_IN_RANGE` and `AttackState`, or `SHOT_IN_HEAD` and `DeathState`). The transition checking is then accomplished by sending all the states in the machine the current input event, and each state returns its output state, if it has any response to the particular input event.

This book will instead give each state a function for checking transitions (rather than a base class function that checks an internal list of registered transitions keyed from input trigger types). In this way, the skeletal framework given in this book is more than capable of performing the classical FSM systems by creating an enumeration of input types and then testing to see if any of them have been triggered in the transition function of the current state, but it also allows for much more complex computations to determine state transition, on a state-by-state basis. The given code framework is more flexible and allows the code modularity that we are striving for.

In classic electrical engineering terms (from which we're borrowing the concept of the FSM in the first place), most FSMs in games are coded using the Moore

model, which just means that you put your actions inside the state, instead of the transition between states (which is a Mealy machine model). Thus, during the `Sit` state, you have the character play a sit animation. In a Mealy machine, the character would play the sit animation during the transition between the `Stand` state and the `Sit` state and would do nothing during the `Sit` state except wait for a transition out. However, with just a bit of clever code placement, you can achieve either effect with the generic structures in this chapter.

Let's look at a simple example, in Figure 15.2.

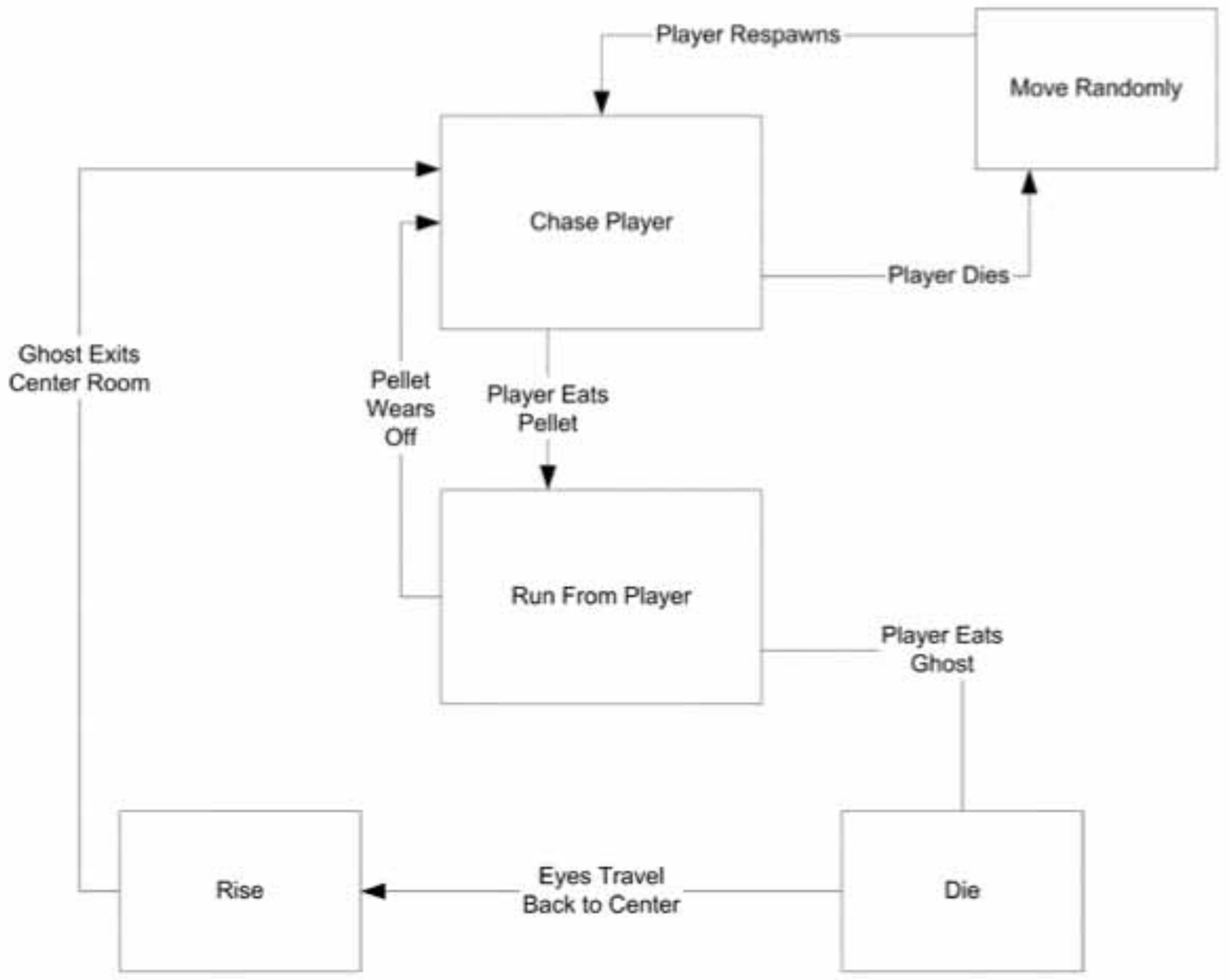


FIGURE 15.2 Simple FSM diagram of the red ghost from *Pac-Man*.

Here we see an FSM diagram for Blinky, the red ghost from *Pac-Man*. Blinky was the one that most directly chased the player. All the ghosts would start life in the `Rise` state because they're currently located in the center part of the maze. During this state, the ghost would get another body (if he doesn't have one), and then exit the center box. Doing this would trigger the FSM to transition to Blinky's primary state, that of `ChasePlayer`. He will stay in this state until one of two things happens: the player dies, or eats a power pellet. If the player dies, Blinky will then transition to `MoveRandomly`. The other exit is to the state `RunFromPlayer`, which will cause Blinky to flee now that he's been turned blue by the power pellet. When running away, if the

TeamLRN

power pellet wears off, he goes back to chasing the player. If he's eaten by *Pac-Man,* he then transitions to the `Die` state, which converts him to a set of eyeballs and walks him back to the center of the maze. As soon as he enters the center, he transitions to `Rise`, and the whole thing starts over again.

Thus, you can see the clear delineation between states of being, input conditions, and transition lines in the diagram. Also, by dividing it in this way, you can see the atomic behaviors that you will need to code to achieve the entire FSM. Dividing the behavior of your AI system into atomic units is very useful, especially if you are going to have different AI-controlled characters that differ in only a few ways, or have specific behaviors missing.

So the diagram for Inky, another ghost in *Pac-Man* that was not as aggressive as Blinky, might be very similar, but have different reasons for switching between the three movement states: `RunFromPlayer`, `ChasePlayer`, and `MoveRandomly`. He might transition between them randomly (totally erratic behavior), or based on some physical distance to the player (avoidance or limited line of sight simulation), or maybe just change his mind every so often (so that he appears to be single minded, but flighty). He would, of course, still need to have the same power pellet and death logic as Blinky because that is basic ghost behavior, rather than each ghost's "personality" (as defined by its movement within the maze).

This very simple FSM controlling Blinky's state *could* be coded as in Listing 15.1, using a simple switch statement. In fact, many games still use this type of free-form FSM for simple game elements. However, if this were not *Pac-Man* but, rather, *Madden Football,* and thus many hundreds of times more complex, you can imagine how this level of organization would be incredibly inadequate, and excessively complex. The priority of transitions becomes harder and harder to determine because it depends on the order of implementation. The function housing this switch statement will get progressively larger as more states are added to the game. The modular system this book uses will give you a formal organizational model for combating these problems.

LISTING 15.1 Free-form FSM Implementation for *Pac-Man*

```
switch(m_currentState)
{
case STATE_RISE:
    if(AtCenter())
        ExitCenter();
    else
        ChangeState(STATE_CHASEPLAYER);
    break;
```

```
case STATE_DIE:
if(!AtCenter())
    ChangeToEyesAndMoveBacktoCenter();
else
    ChangeState(STATE_RISE);
break;

case STATE_RUNFROMPLAYER:
if(!PoweredPacMan())
    ChangeState(STATE_CHASEPLAYER);
else if(Eaten())
    ChangeState(STATE_DIE);
else
    MoveAwayFromPacMan();
break;

case STATE_CHASEPLAYER:
if(PoweredPacMan())
    ChageState(STATE_RUNFROMPLAYER);
else if(!PacMan)
    ChangeState(STATE_MOVERANDOMLY);
else
    MoveTowardsPacMan();
break;

case STATE_MOVERANDOMLY:
if(PacMan)
    ChangeState(STATE_CHASEPLAYER);
else
    MoveRandomly();
break;

    default:
    PrintError("Bad Current State");
    break;

};
```

FSM SKELETAL CODE

The code will be implemented in the following classes:

- The `FSMState` class, the basic *state* in the system.
- The `FSMMachine` class, which houses all the states and acts as the *state machine*.

TeamLRN

- The `FSMAIControl` class, which houses the state machine, as well as game-specific code such as perception data.

The next sections will discuss these classes in more detail, and will then discuss the specific implementation of the `FSMAIControl` class and each `FSMState` needed for our AI test bed application.

The `FSMState` Class

When implementing a state system, it is best to code each state as if it is the only state in the world, with no knowledge of other states, or of the state machine itself. This leads to very modular states, which can be arranged in any order without prerequisite or future requirement. At its most basic, each state should have the following functions:

`Enter()`. This function is always run as soon as you enter the state. It allows the state to perform initialization of data or variables.

`Exit()`. This function is run when you are leaving the state and is primarily used as a cleanup task, as well as where you would run any additional code that you wanted to happen on specific transitions (for Mealy-style state machines).

`Update()`. This is the main function that is called every processing loop of the AI, if this state is the current state in the FSM (for Moore-style state machines).

`Init()`. Resets the state.

`CheckTransitions()`. This function runs through the logic by which the state will decide to end. The function should return the enumeration value of the state to run, coming back with the same state if no change is needed. Note that the order in which the logical state transitions are determined becomes the priority of the different transitions. So, if your function first checks for a switch to the attacking state, and then checks for the dodging state, the AI will be much more offensive than if those checks were reversed.

The skeletal code header for this class can be seen in Listing 15.2. The class complexity has been kept to a minimum, so that this code can be the foundation for any system that you might need to build using an FSM. The class also contains two data members, `m_type`, and `m_parent`. The type field is used by both the overall state machine and by the interstate code to make determinations based on which particular state is being considered. The enumeration for these values is stored in a file called `FSM.h`, and is currently empty, containing only the default `FSM_STATE_NONE` value. When you actually use the code for something, you would add all the state types to this enumeration, and go from there. The parent field is used by individual states, so they can access a shared data area through their `Control` structure.

LISTING 15.2 Base Class Header for State

```
class FSMState:
{
public:
        //constructor/functions
        FSMState(int type=FSM_STATE_NONE,Control* parent=NULL)
          {m_type = type;m_parent = parent;}
        virtual void Enter()                    {}
        virtual void Exit()                     {}
        virtual void Update(int t)              {}
        virtual void Init()                     {}
        virtual void CheckTransitions(int t) {}

           //data
      Control* m_parent;
           int      m_type;
};
```

The FSMMachine Class

The state machine class (see Listing 15.3 for the header) contains all the states associated with machine in an STL vector. It also has a general case `UpdateMachine()` function, the implementation of which is shown in Listing 15.4. It also contains functions for adding states to the machine and setting a default state. Notice that the state machine is actually derived from the state class. This is to facilitate a state that is actually a completely different state machine. Again, like the state class, the machine class has a type field, the types of which are declared in an enumeration in `FSM.h`, which is essentially empty for now.

LISTING 15.3 FSMachine Header

```
class FSMMachine: public FSMState
{
     public:
     //constructor/functions
     FSMMachine(int type = FSM_MACH_NONE)
          {m_type = type;}
     virtual void UpdateMachine(int t);
     virtual void AddState(FSMState* state);
     virtual void SetDefaultState(FSMState* state)
          {m_defaultState = state;}
     virtual void SetGoalID(int goal) {m_goalID= goal;}
     virtual TransitionState(int goal);
     virtual Reset();
```

TeamLRN

```
        //data
        int m_type;
    private:
        vector<FSMState*> m_states;
        FSMState* m_currentState;
        FSMState* m_defaultState;
        FSMState* m_goalState;
        FSMState* m_goalID;
    };
```

LISTING 15.4 The Machine Class `UpdateMachine()` Function

```
void FSMMachine::UpdateMachine(int t)
{
    //don't do anything if you have no states
    if(m_states.size() == 0 )
        return;

    //don't do anything if there's no current
    //state, and no default state
    if(!m_currentState)
        m_currentState = m_defaultState;
    if(!m_currentState)
        return;

    //update current state, and check for a transition
    int oldStateID = m_currentState->m_type;
    m_goalID = m_currentState->CheckTransitions();

    //switch if there was a transition
    if(m_goalID != oldStateID)
    {
        if(TransitionState(m_goalID))
        {
            m_currentState->Exit();
            m_currentState = m_goalState;
            m_currentState->Enter();
        }
    }
    m_currentState->Update(t);
}
```

The `UpdateMachine()` function is very simple. It has two quick optimizations: it will bail back out if the machine wasn't given any states, and will also return if

there is no current state set and no default state to fall back on. The next block calls the current state's `Update()` and `CheckTransition()` functions, and then the next block actually determines if the state triggered a transition. If so, the function `TransitionState()`queries the machine's list of states to see if the machine actually has the new state that was requested, and if it exists, calls `Exit()` on the state the system is leaving, and `Enter()` on the new state.

The `FSMAIControl` Class

The final part of the basic FSM system (and also the beginning of the game-specific code) is the `Control` class (which was covered briefly in Chapter 3, "AIsteroids: Our AI Test Bed"). As you recall, this class is the behavior controller for the main in-game ship and serves as the branching point between the human controls and the primary location for the AI framework. So, for an AI-controlled ship, we will be inheriting from `AIControl` and creating the child class `FSMAIControl` (see Listing 15.5 for the header).

LISTING 15.5 `FSMAIControl` Header

```
class FSMAIControl: public AIControl
{
     public:
     //constructor/functions
     FSMAIControl(Ship* ship = NULL);
     void Update(int t);
     void UpdatePerceptions(int t);
     void Init();

     //perception data
     //(public so that states can share it)
     GameObj*       m_nearestAsteroid;
     GameObj*       m_nearestPowerup;
     float          m_nearestAsteroidDist;
     float          m_nearestPowerupDist;
     Point3f        m_collidePt;
     bool           m_willCollide;
     bool           m_powerupNear;
     float          m_safetyRadius;

private:
     //data
     FSMMachine* m_machine;
};
```

TeamLRN

The `FSMAIControl` class contains the standard `Update()` function, which updates both the state machine and runs the `UpdatePerceptions()` method. This class also includes the game specific blackboard data members that will be shared by all the states in the machine. If this were a much more complex game, with large numbers of these kinds of global data members (or a variety of simple and very complex data members that required a lot of management on a particular perception level), it would be much better to construct a perception manager class and just have the `FSMAIController` contain the correct perception manager for this game. But for the simple needs of our test bed demo, this will do fine. Having only this minimal list of data members to maintain, we don't have to worry about the calculations taking too long, or having to wade through an unwieldy long perception update function.

IMPLEMENTING AN FSM-CONTROLLED SHIP INTO OUR TEST BED

To get our AIsteroids program to start using this method, we first need to determine the entire state diagram for the behavior exhibited by a ship during a game of asteroids that we want our system to model. For our purposes, Figure 15.3 should perform fine.

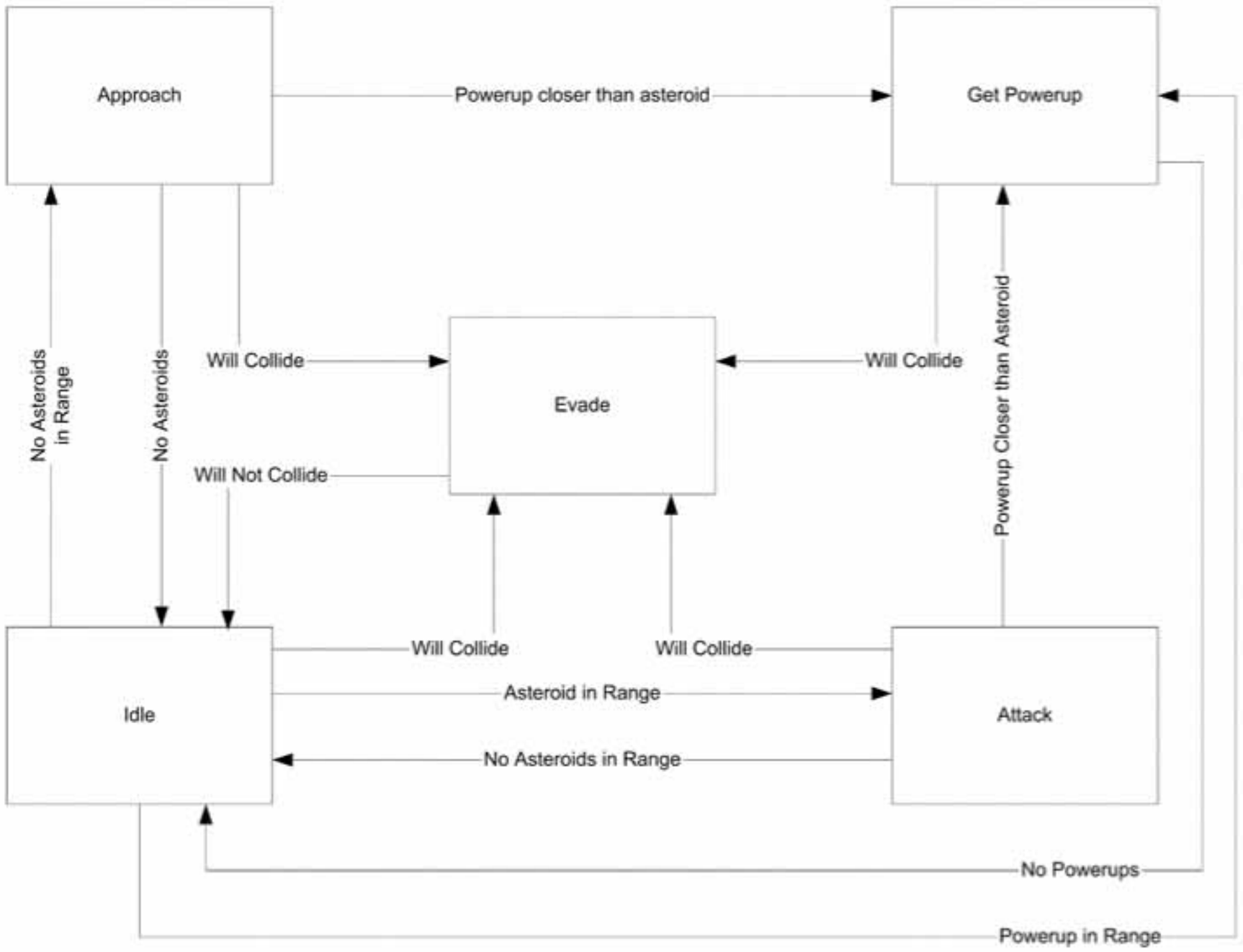


FIGURE 15.3 FSM diagram for asteroids.

As the figure shows, there are five basic states to an AI-controlled AIsteroids ship:

- `Approach`, which will get the ship within range of the closest asteroid.
- `Attack`, which will point the ship toward the closest asteroid within range, and then fire.
- `Evade`, which will initiate avoidance of an asteroid on a collision course.
- `GetPowerup`, which will try to scoop up powerups within some range.
- `Idle`, will just sit there if nothing else is valid.

The game also needs the following conditions to make the necessary logical connections between these states:

- Asteroid in firing range. A simple distance check, but it also requires that we keep track of the closest asteroid.
- Asteroid on collision course. Distance check, but also a trajectory intersection. The intersection is more costly, so we'll only do it if the asteroid is within the distance check.
- Powerup in pickup range. Another distance check, but this also requires that we keep track of the closest powerup.

You should also notice one other thing about the state diagram: every state needs to check for the condition "Asteroid on collision course," to then switch to the `Evade` state. This shows one of the inherent weaknesses of building the logic into each state because this type of determination would have to be repeated in each state. But, because we'll be using the `Control` class's `UpdatePerceptions()` function as a global data location (to the states under this particular `Control`'s influence), we are in effect using the control class as a central location that will hold calculations global to the entire state machine. This gives us the best of both worlds, by keeping the number of recalculations to a minimum (through a central storage location) and giving us the ability to separate out the nonrepetitious portions of the calculations to be done only when needed (by putting logic and calculations within specific states).

EXAMPLE IMPLEMENTATION

Now, we will take the FSM classes we have discussed and use them to construct a working AI ship for our test application. We will first set up the `Control` class, and then implement each of the requisite states for the system.

TeamLRN

Coding the Control class

The controller class for the FSM model (see Listing 15.5 for the header, and Listing 15.6 for the implementation of the important functions) contains the state machine structure, as well as the global data members for this AI model.

The constructor for the class builds the FSM structure, by instantiating the machine class, and then adding an instantiation of each requisite state. The constructor also sets the default state, which is also used as the startup state for the machine.

The `Update()` method is straightforward and ensures that the ship this class is controlling exists, and if so, updates the perceptions and the state machine.

The `UpdatePerceptions()` function is where all the action is. The closest asteroid and powerup are noted, the ship's distance to these objects is determined, and the status variables are set (`m_willCollide` and `m_powerupNear`). These perceptions allow all the transition checking in the individual states to be simple comparisons, instead of having to calculate these things individually. This approach also consolidates this code—better or faster methods can be implemented here and the effects will be seen throughout the states.

LISTING 15.6 `FSMAIControl` Function Implementations

```
//--------------------
FSMAIControl::FSMAIControl(Ship* ship):
AIControl(ship)
{
    //construct the state machine and add the necessary states
    m_machine = new FSMMachine(FSM_MACH_MAINSHIP,this);
    StateApproach* approach = new StateApproach(this);
    m_machine->AddState(approach);
    m_machine->AddState(new StateAttack(this));
    m_machine->AddState(new StateEvade(this));
    m_machine->AddState(new StateGetPowerup(this));
    m_machine->AddState(new StateIdle(this));
    m_machine->SetDefaultState(approach);
}

//--------------------
void FSMAIControl::Update(int t)
{
    if(!m_ship)
    {
        m_machine->Reset();
        return;
    }
```

```
    UpdatePerceptions(t);
    m_machine->UpdateMachine(t);
}

//---------------------
void FSMAIControl::UpdatePerceptions(int t)
{
    //store closest asteroid and powerup
    m_nearestAsteroid = Game.GetClosestGameObj
                        (m_ship,GameObj::OBJ_ASTEROID);
    m_nearestPowerup  = Game.GetClosestGameObj
                        (m_ship,GameObj::OBJ_POWERUP);

    //asteroid collision determination
    m_willCollide = false;

//small hysteresis on this value, to avoid
//boundary oscillation
    if(m_willCollide)
        m_safetyRadius = 30.0f;
    else
        m_safetyRadius = 15.0f;
    if(m_nearestAsteroid)
    {
        float speed = m_ship->m_velocity.Norm();
        m_nearestAsteroidDist = m_nearestAsteroid->
                    m_position.Distance(m_ship->m_position);
        float dotVel;
        Point3f normDelta = m_nearestAsteroid->m_position -
                            m_ship->m_position;
        normDelta.Normalize();
        float astSpeed = m_nearestAsteroid->
                            m_velocity.Norm();
        if(speed > astSpeed)
            dotVel  = DOT(m_ship->UnitVectorVelocity()
                        ,normDelta);
        else
        {
            speed = astSpeed;
            dotVel= DOT(m_nearestAsteroid->
                    UnitVectorVelocity(),-normDelta);
        }
        float spdAdj = LERP(speed/AI_MAX_SPEED_TRY
                            ,0.0f,50.0f)*dotVel;
```

TeamLRN

```
        float adjSafetyRadius = m_safetyRadius + spdAdj +
                                m_nearestAsteroid->m_size;

        //if you're too close, and I'm heading somewhat
        //towards you, flag a collision
        if(m_nearestAsteroidDist <= adjSafetyRadius
           && dotVel > 0)
            m_willCollide = true;
    }

    //powerup near determination
    m_powerupNear = false;
    if(m_nearestPowerup)
    {
        m_nearestPowerupDist = m_nearestPowerup->m_position.
                               Distance(m_ship->m_position);
        if(m_nearestPowerupDist <= POWERUP_SCAN_DIST)
        {
            m_powerupNear     = true;
        }
    }
}
```

Coding the States

The next five listings (15.7–15.11) are the implementations for the necessary states. They will be discussed separately, followed by the relevant listing.

`StateApproach`

This state's purpose is to turn to face the nearest asteroid and then thrust toward it. For simplicity's sake, the AI system for this demo doesn't try to deal with the wrap-around effect of the game world—that would require more math, and is not the focus of this text.

The `Update()` function does some calculations to find the approach angle to the nearest asteroid and will add a *braking* vector if the speed of the ship is overly high. This is to keep the AI-controlled ship from occasionally getting into trouble because of too much speed.

After the angle is computed, the code then turns the ship in the proper direction, or turns on the appropriate thruster if the ship is already pointing correctly. This type of movement is a bit more digital than most human players, so it looks a little more robotic than human. It could be made more natural looking by using the thrusters during turning (which is what most humans do), but again, this would

complicate the calculations and this example is being coded specifically for readability, not to show the optimal implementation.

The `CheckTransitions()` function is straightforward enough, checking in turn for the three possible transitions from this state, `FSM_STATE_EVADE` (if you're going to collide), `FSM_STATE_GETPOWERUP` (if there's one nearby), and `FSM_STATE_IDLE` (if there's no asteroid to approach).

The `Exit()` function assures the system that anything the state sets in the larger game world will be reset. In this case, the ship's turn and thrust controls may be turned on, so this function turns them both off.

LISTING 15.7 The `StateApproach` Class Functions

```
//--------------------
void StateApproach::Update(int t)
{
    //turn and then thrust towards closest asteroid
    FSMAIControl* parent = (FSMAIControl*)m_parent;
    GameObj* asteroid = parent->m_nearestAsteroid;
    Ship*    ship     = parent->m_ship;
    Point3f deltaPos  = asteroid->m_position -
                        ship->m_position;
    deltaPos.Normalize();

    //add braking vec if you're going too fast
    float speed = ship->m_velocity.Norm();
    if(speed > AI_MAX_SPEED_TRY)
        deltaPos += -ship->UnitVectorVelocity();

    //DOT out my velocity
    Point3f shpUnitVel = ship->UnitVectorVelocity();
    float dotVel = DOT(shpUnitVel,deltaPos);
    float proj = 1-dotVel;
    deltaPos -= proj*shpUnitVel;
    deltaPos.Normalize();

    //find new direction, and head to it
    float newDir = CALCDIR(deltaPos);
    float angDelta = CLAMPDIR180(ship->m_angle - newDir);
    if(fabsf(angDelta) <2 || fabsf(angDelta)> 172)
    {
        //thrust
        ship->StopTurn();
        if(speed < AI_MAX_SPEED_TRY ||
```

TeamLRN

```
            parent->m_nearestAsteroidDist > 40)
             fabsf(angDelta)<2? ship->ThrustOn() :
                                ship->ThrustReverse();
        else
            ship->ThrustOff();
    }
    else if(fabsf(angDelta)<=90)
    {
        //turn when facing forwards
        if(angDelta >0)
            ship->TurnRight();
        else
            ship->TurnLeft();
        }
        else
        {
            //turn when facing rear
            if(angDelta<0)
                ship->TurnRight();
            else
                ship->TurnLeft();
        }

        parent->m_target->m_position = asteroid->m_position;
        parent->m_targetDir = newDir;
        parent->m_debugTxt = "Approach";
    }

    //--------------------
    int StateApproach::CheckTransitions()
    {
        FSMAIControl* parent = (FSMAIControl*)m_parent;
        if(parent->m_willCollide)
            return FSM_STATE_EVADE;

        if(parent->m_powerupNear&&(parent->m_nearestAsteroidDist
    >parent->m_nearestPowerupDist)&& parent->m_ship->
    GetShotLevel() < MAX_POWER_LEVEL)
            return FSM_STATE_GETPOWERUP;

        if(!parent->m_nearestAsteroid ||
           parent->m_nearestAsteroidDist < APPROACH_DIST)
            return FSM_STATE_IDLE;
```

```
    return FSM_STATE_APPROACH;
}

//--------------------
void StateApproach::Exit()
{
    if(((FSMAIControl*)m_parent)->m_ship)
    {
        ((FSMAIControl*)m_parent)->m_ship->ThrustOff();
        ((FSMAIControl*)m_parent)->m_ship->StopTurn();
    }
}
```

StateAttack

This `StateAttack` class will turn the ship toward the nearest asteroid, and then fire the cannon. The class accounts for multiple guns (given to the player by getting powerups) by calling the ship function `GetClosestGunAngle()`, which will pass in the closest gun to an angle parameter.

`Update()` calculates the position of the nearest asteroid, but must also perform some additional calculations to find the projected position of the asteroid, to find the leading angle to fire a bullet toward to hit it while it's moving. After finding this position, it gets an angle to it, turns the ship, and fires the guns.

`CheckTransitions()` for this state is just like `StateApproach`, with branches to `FSM_STATE_EVADE`, `FSM_STATE_GETPOWERUP`, and `FSM_STATE_IDLE`.

This state only turns the ship, so the `Exit()` function need only concern itself with resetting that particular flag.

LISTING 15.8 The `StateAttack` Class Functions

```
//--------------------
void StateAttack::Update(int t)
{
    //turn towards closest asteroid's future position,
    //and then fire
    FSMAIControl* parent = (FSMAIControl*)m_parent;
    GameObj* asteroid    = parent->m_nearestAsteroid;
    Ship*    ship        = parent->m_ship;

    Point3f futureAstPosition = asteroid->m_position;
    Point3f deltaPos = futureAstPosition - ship->m_position;
    float dist  = deltaPos.Norm();
    float time = dist/BULLET_SPEED;
    futureAstPosition += time*asteroid->m_velocity;
```

TeamLRN

```
        Point3f deltaFPos = futureAstPosition - ship->m_position;
        deltaFPos.Normalize();

        float newDir   = CALCDIR(deltaFPos);
        float angDelta = CLAMPDIR180(ship->GetClosestGunAngle
                                     (newDir) - newDir);
        if(angDelta >1)
            ship->TurnRight();
        else if(angDelta < -1)
            ship->TurnLeft();
        else
        {
            ship->StopTurn();
            ship->Shoot();
        }

        parent->m_target->m_position = futureAstPosition;
        parent->m_targetDir = newDir;
        parent->m_debugTxt = "Attack";
    }

    //--------------------
    int StateAttack::CheckTransitions()
    {
        FSMAIControl* parent = (FSMAIControl*)m_parent;
        if(parent->m_willCollide)
            return FSM_STATE_EVADE;

        if(parent->m_powerupNear && parent->m_nearestAsteroidDist
           >parent->m_nearestPowerupDist && parent->m_ship->
           GetShotLevel() < MAX_POWER_LEVEL)
            return FSM_STATE_GETPOWERUP;

        if(!parent->m_nearestAsteroid ||
           parent->m_nearestAsteroidDist > APPROACH_DIST)
            return FSM_STATE_IDLE;

        return FSM_STATE_ATTACK;
    }

    //--------------------
    void StateAttack::Exit()
    {
```

```
    if(((FSMAIControl*)m_parent)->m_ship)
        ((FSMAIControl*)m_parent)->m_ship->StopTurn();
}
```

StateEvade

An important state which simply tries to stop collisions with asteroids, by both performing thrusting maneuvers, as well as firing the guns to possibly clear the way.

The `Update()` function computes a steering vector that comprises a sideways normal vector to the line between you and the asteroid and adds in a braking vector if you're headed at the asteroid. The `Update()` function then calculates the angle to this thrust vector, and like `StateApproach`, turns the ship and thrusts when appropriate, but will also fire the ship's guns when using its thrusters, which has the added benefit of sometimes clearing out the area.

`CheckTransition()` has only one state to check for, that of `FSM_STATE_IDLE`. We could check for transitions to the other states directly, but this is undesirable. By keeping the state connections to a minimum, we lessen the CPU requirements of running the state machine (especially if the transition determinations are more complex than simple comparisons) and make the overall state diagram simpler and easier to add to in the future when we want to insert more states into the system.

The `Exit()` method for `StateEvade` is like any other state that controls movement, in that it must reset the turning and engine status of the ship being controlled.

LISTING 15.9 The `StateEvade` Class Functions

```
//--------------------
void StateEvade::Update(int t)
{
    //evade by going to the quad opposite as the asteroid
//is moving, add in a deflection,
//and cancel out your movement
FSMAIControl* parent = (FSMAIControl*)m_parent;
GameObj* asteroid    = parent->m_nearestAsteroid;
Ship*    ship        = parent->m_ship;
Point3f vecSteer = CROSS(ship->m_position,asteroid->
                   m_position);
Point3f vecBrake = ship->postion - asteroid->m_position;
vecSteer += vecBrake;

float newDir = CALCDIR(vecSteer);
float angDelta = CLAMPDIR180(ship->m_angle - newDir);
if(fabsf(angDelta) <5 || fabsf(angDelta)> 175)//thrust
{
```

TeamLRN

```
            ship->StopTurn();
            if(ship->m_velocity.Norm() < AI_MAX_SPEED_TRY ||
               parent->m_nearestAsteroidDist< 20 +asteroid->
               m_size)
                fabsf(angDelta)<5?
                     ship->ThrustOn() : ship->ThrustReverse();
            else
                ship->ThrustOff();

            //if I'm pointed right at the asteroid, shoot
            ship->Shoot();
        }
        else if(fabsf(angDelta)<=90)//turn front
        {
            if(angDelta >0)
                ship->TurnRight();
            else
                ship->TurnLeft();
        }
        else//turn rear
        {
            if(angDelta<0)
                ship->TurnRight();
            else
                ship->TurnLeft();
        }


        parent->m_target->m_position=asteroid->m_position;
        parent->m_targetDir = newDir;
        parent->m_debugTxt = "Evade";
    }

    //--------------------
    int StateEvade::CheckTransitions()
    {
        FSMAIControl* parent = (FSMAIControl*)m_parent;

        if(!parent->m_willCollide)
            return FSM_STATE_IDLE;

        return FSM_STATE_EVADE;
    }
```

```
//--------------------
void StateEvade::Exit()
{
    if(((FSMAIControl*)m_parent)->m_ship)
    {
        ((FSMAIControl*)m_parent)->m_ship->ThrustOff();
        ((FSMAIControl*)m_parent)->m_ship->StopTurn();
    }
}
```

StateGetPowerup

This state recognizes the locality of a powerup and will attempt to force a collision with the powerup, to gain its effects.

`Update()` is much like in `StateApproach`, only we need more precise collision, instead of just moving in the general direction. So, this state must compute projected movement of the powerups. Also like the approach state, it tries to keep the maximum velocity of the ship under check, by imposing a braking factor if the ship is moving too fast. Like some of the other states, `Update()` then computes a new direction, turns to it, and fires up the engines.

`CheckTransitions()` has determinations for both exit clauses from this state, `FSM_STATE_EVADE and FSM_STATE_IDLE`.

`Exit()` must reset the ship's turn and thrust controls to ensure leaving them in a neutral manner.

LISTING 15.10 The `StateGetPowerup` Class Functions

```
//--------------------
void StateGetPowerup::Update(int t)
{
    FSMAIControl* parent = (FSMAIControl*)m_parent;
    GameObj* powerup     = parent->m_nearestPowerup;
    Ship*    ship        = parent->m_ship;

    //find future position of powerup
    Point3f futurePowPosition = powerup->m_position;
    Point3f deltaPos = futurePowPosition - ship->m_position;
    float dist  = deltaPos.Norm();
    float speed = AI_MAX_SPEED_TRY;
    float time = dist/speed;
    futurePowPosition += time*powerup->m_velocity;
    Point3f deltaFPos = futurePowPosition - ship->m_position;
    deltaFPos.Normalize();
```

TeamLRN

```
//add braking vec if you're going too fast
speed = ship->m_velocity.Norm();
if(speed > AI_MAX_SPEED_TRY)
    deltaFPos += -ship->UnitVectorVelocity();

//DOT out my velocity
Point3f shpUnitVel = ship->UnitVectorVelocity();
float dotVel       = DOT(shpUnitVel,deltaFPos);
float proj         = 1-dotVel;
deltaFPos         -= proj*shpUnitVel;
deltaFPos.Normalize();

float newDir   = CALCDIR(deltaFPos);
float angDelta = CLAMPDIR180(ship->m_angle - newDir);
if(fabsf(angDelta) <2 || fabsf(angDelta)> 177)//thrust
{
    ship->StopTurn();
    if(speed < AI_MAX_SPEED_TRY ||
       parent->m_nearestPowerupDist > 20)
        fabsf(angDelta)<2?
        ship->ThrustOn() : ship->ThrustReverse();
    else
        ship->ThrustOff();
}
else if(fabsf(angDelta)<=90)//turn front
{
    if(angDelta >0)
        ship->TurnRight();
    else
        ship->TurnLeft();
}
    else//turn rear
    {
        if(angDelta<0)
            ship->TurnRight();
        else
            ship->TurnLeft();
    }

    parent->m_target->m_position = futurePowPosition;
    parent->m_targetDir          = newDir;
    parent->m_debugTxt           = "GetPowerup";
}
```

```
//--------------------
int StateGetPowerup::CheckTransitions()
{
    FSMAIControl* parent = (FSMAIControl*)m_parent;

    if(parent->m_willCollide)
        return FSM_STATE_EVADE;

    if(!parent->m_nearestPowerup || parent->
       m_nearestAsteroidDist < parent->m_nearestPowerupDist)
        return FSM_STATE_IDLE;

    return FSM_STATE_GETPOWERUP;
}

//--------------------
void StateGetPowerup::Exit()
{
    if(((FSMAIControl*)m_parent)->m_ship)
    {
        ((FSMAIControl*)m_parent)->m_ship->ThrustOff();
        ((FSMAIControl*)m_parent)->m_ship->StopTurn();
    }
}
```

StateIdle

The last necessary state is merely a catch all, purely transitory state. The state machine for this simple demo has so few states that `StateIdle` actually has connectivity to every other state in the machine, but this doesn't have to be true. If we started adding additional behaviors to this game (such as specialized attack states) then these would be more isolated in the tree. But the simple nature of this game provides for this state to be a common return point from the other states, so that the ship will always fall back into this state when done with the others.

The `Update()` function of this state does nothing, except provide the debugging system with a label to use when drawing debug information to the screen.

`CheckTransitions()` has determinations for all the other states in the game because of the foundation nature of the idle state in this game.

There is no `Exit()` function for this state, as it changes nothing in the greater game sense.

TeamLRN

LISTING 15.11 The `StateIdle` Class Functions

```
//--------------------
void StateIdle::Update(int t)
{
    //Do nothing
    FSMAIControl* parent = (FSMAIControl*)m_parent;
    parent->m_debugTxt = "Idle";
}

//--------------------
int StateIdle::CheckTransitions()
{
    FSMAIControl* parent = (FSMAIControl*)m_parent;

    if(parent->m_willCollide)
        return FSM_STATE_EVADE;

    if(parent->m_nearestAsteroid)
    {
        if(parent->m_nearestAsteroidDist > APPROACH_DIST)
            return FSM_STATE_APPROACH;

        if(parent->m_nearestAsteroidDist <= APPROACH_DIST)
            return FSM_STATE_ATTACK;
    }

    if(parent->m_nearestPowerup)
        return FSM_STATE_GETPOWERUP;

    return FSM_STATE_IDLE;
}
```

PERFORMANCE OF THE AI WITH THIS SYSTEM

The AI is quite able to play a good game of asteroids with this simple framework, being able to occasionally achieve scores well over 2 million. The added behavior of shooting while in the `StateEvade` state seems to be key to the ability of the system to survive later levels because the craft is almost continuously evading the extreme numbers of asteroids. However, by just watching it for a while, you will notice a number of things that could use improvement:

- *The addition of some specialty states.* Getting the first powerup significantly improves the AI's chance of survival, so this could be a priority state. Specifically filling up on powerups when the number of asteroids is low would be a big help, so that it will start the next level with maximum guns. Also, humans can play this game forever if they just get full powerups and then sit in the middle of the screen and continuously rotate and fire. This "spiral death blossom" attack is something that the AI could do at appropriate times, such as when it's surrounded. Taking advantage of invincibility would be another state—the AI ship could make a beeline for powerups or ignore evasion tactics when invincible.
- *Increased complexity of the math model.* This gives the AI system the ability to deal with the world coordinates wrapping. Right now, the AI's primary weakness is that it loses focus when things wrap in the world, and considering this during targeting and collision avoidance would greatly increase the survivability of the AI ship.
- *Bullet management for the ship.* Right now, the ship just points, and then starts firing. There is no firing rate on the guns, so it tends to fire clumps of shots toward targets. This is somewhat advantageous; when it fires a clump of shots into a large asteroid, the remaining shots will sometimes kill the pieces as the asteroid splits. But this can get the ship in trouble when it has fired its entire allocation of bullets and must wait for them to collide or expire before it can shoot again, leaving it temporarily defenseless.
- *Better positioning the ship for attacks.* This means the ship doesn't miss fast moving targets quite so often. Humans tend to move to some position that the asteroid will eventually travel by, and then stop at that position and wait for the asteroid to come. Because the math was specifically kept simple for the demo, the system moves directly toward the asteroid. Even this simple method is really only a problem because of the world wrapping effect. This method of play doesn't really look as intelligent as the human scheme.
- *Better evade behaviors.* Right now, the ship is using simple steering behavior (modified slightly, because we can only thrust forward and reverse) for obstacle avoidance. Humans use a much more complex determination for avoidance, including shooting though a potential collision (not making any thrust adjustments), noting clumps of asteroids coming and evading them as a group, preemptive positioning before an asteroid gets too close, or even braking to a stop to just slow down the action a bit. A bit of simple playfield analysis would help the AI with some of these actions. By knowing which parts of the map had the lesser concentrations of asteroids, it could perform evasion tactics in the general direction of "more space," or even set itself up in low-concentration areas preemptively to give itself a better chance for survival.

TeamLRN

Pros of FSM-Based Systems

FSMs are easy and intuitive to picture, especially when dealing with Moore-style machines. Our implementation into the test bed, which used a Moore-style state machine, where the actions are in the states (rather than the transitions), is how most people tend to think about AI behaviors. Even within this paradigm, however, you could have coded the FSM in many ways for the demo game to achieve similar performance.

FSMs are also easy to implement, as you've seen in this chapter. Given a well thought out state diagram, the structure of the state machine practically writes itself. Its simplicity is its greatest strength because the nature of the methodology lends itself well to splitting AI problems into specific chunks and defining the linkages between them. After a while, writing FSM structures becomes a fairly rote task for most programmers.

State-based systems are easy to add to because the game flow is very deterministic and connections between states are so explicit. In fact, it is a good idea to make a paper copy of your FSM diagram (or specific portions of it, if it is very large) and continue to keep it current as you extend the system. This will augment your ability to maintain a mental picture of the overall FSM structure and will help you find logical holes or areas where you need a connection but don't have one. This kind of bookkeeping could even be achieved by inserting special debugging code into your states, so that the state diagram could effectively be written to a file by your game and looked at offline, to look for any transitions that you missed or are misplaced.

FSM methods are also very straightforward to debug. The deterministic nature of state machines makes it easy (usually, that is) to replicate bugs, and the centralized nature of the `FSMMachine` class makes an easy code location to trap specific AI characters or behaviors when they occur. Visual debugging is also made easy in this paradigm because it is trivial to output state information to the screen on an individual character basis and watch the AI make determinations on the fly. This kind of information can also be useful written to a file as a log of the state transitions leading up to a certain condition.

Finally, because of their nonspecific nature, FSM systems can be used for any number of problems, from simple game flow between screens, to the most intricate of NPC dialogues. This inherent general purpose quality means that at some level, almost every game will have some sort of state-based element to them. Not that very simple state systems need a full, formal framework to run, but almost every game will use FSMs in some form simply because they can be applied to such a vast array of different game issues.

Cons of FSM-Based Systems

The primary strength of FSM systems, their ease of implementation, sometimes tends to be their greatest weakness as well. Projects can run into problems when state-based systems weren't initially designed with a static framework from the start and, instead, used more "switch and case" based FSMs, mixed in with more formal-state machines. Programmers sometimes code a behavior quickly (during a crunch period, or during a moment of experimentation) and then don't bother to go back and reimplement it correctly into the overall game structure. This leads to fragmented systems that have logic spread out in directions and places that are not organizationally sound, leading to maintenance problems.

FSM systems also tend to grow in complexity during the project, as more specialized behaviors are found (such as those mentioned earlier that could improve the asteroids-playing FSM from the start of this chapter). Although it is good to try to improve the abilities of your AI systems over time, FSMs tend to not scale well to this kind of iterative work. The state diagram will become incredibly complex as the number of transitions will grow exponentially to the number of new states and as such transition determination and priority of actions becomes unwieldy to resolve.

Another downfall of the state-based model is the issue of state oscillation. This occurs when the perception data that separates two or more states is too crisp—that is, there is no room for overlap. For example, let's say a creature in a game (see Figure 15.4) has only two states, Flee and Stand. Flee runs directly away from any enemy less than four feet from him, and Stand just causes the creature to sit there. Now, an enemy character enters the scene, and stands 3.99 feet from the creature. The creature enters his Flee state, but as he starts his animation, his position changes slightly, and suddenly, he's 4.001 feet from the enemy. So he transitions to Stand. The Stand state plays a different animation, so it might move him back a touch, and start the whole thing over again. This is a very specific and simplistic example, but the lesson is that the inherent crispness of the state system can lead to vacillating states like this unless care is taken. Some ways to fight this problem will be given in the following section.

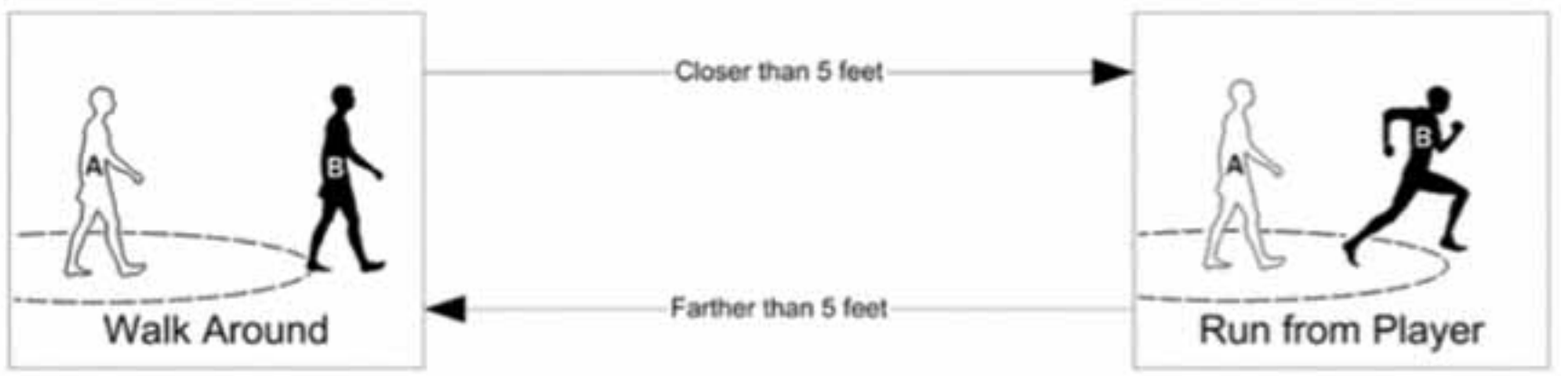


FIGURE 15.4 Common state-based problem of oscillation.

TeamLRN

EXTENSIONS TO THE PARADIGM

Because of the extremely open-ended implementation of FSMs, a number of useful variants have been used over the years to combat the weaknesses of FSM systems. Some of the more useful of these extensions will be covered here.

Hierarchical FSMs

Sometimes, a given state in an FSM will be quite complex. In our AIsteroids example, the evade state could be made much more complicated in an attempt to make it more foolproof. Special case code could be written to separate situations such as when you're surrounded, or a tight grouping of asteroids is coming your way. Other code could try to preempt collisions by moving to more open areas, or shooting your way straight through oncoming traffic. Some of these things could be taken care of within the current evade `Update()` method, but a better way to approach this would be to make the evade state an entirely different state machine. Within this state machine, you could deal with threats iteratively and separate code into more manageable sections. So, the evade state machine would contain states for first dealing with being surrounded, then deal with any immediate threats by either shooting or dodging, and then try to get to a safer location so that it can exit the evade state completely.

This technique is a great way to add complexity to an FSM system without creating undue connectivity within the greater state machine. In effect, you are grouping states into more locally scoped areas, and taking advantage of similarities among these local states. By grouping similar states within their own state machine, the "super state" that contains this new machine can also house common functionality and shared data members, much like the `FSMAIControl` structure does for the AIsteroids example.

Substates do not have to be true states, either. Another commonly used technique is to have a state in the larger FSM contain many substates, which it then randomly (or because of some combination of the perception triggers) sets off one of the equivalent substates. This is the same as having two or more states being equal branches in a classic state diagram, but having the logic for which branch to take being embedded in a state update method instead of indirectly through perception order priority or some other roundabout manner.

Message- and Event-Based FSMs

In some games (or merely some states), transitions may happen infrequently. If this is the case, and if your game also contains numerous states or the computations to determine transitions are complex, then it becomes computationally expensive to

check for transitions in a polling model. Instead, an FSM system can be implemented easily that uses messages as triggers instead of having to poll.

The overall structure of our state-machine framework could be converted to use this type of system. The game (most likely through the `Control` class in some way) would have to pass messages down to the state machine, which would then distribute them to the various states. The `FSMMachine::UpdateMachine()` method would become the message pump for the state machine, and each state's `CheckTransitions()` function would become a switch statement for handling the various messages that it wants to consider. The rest of the code would remain mostly unchanged. Even the `Enter()`, `Exit()`, and `Update()` functions could be triggered by automatically sent messages through the system. Note that combination systems could be implemented, where each state could store a flag indicating whether it is a polling or event-driven state, and the `UpdateMachine()` function could handle it appropriately.

FSMs with Fuzzy Transitions

FSMs can be written so that instead of events or some kind of perception trigger causing transitions in the machine, fuzzy determinations (such as simple comparisons or calculations) can be used to trigger state transitions. Because of the way the framework in this chapter has been coded, this technique requires no code changes to implement. In fact, the implementation of AIsteroids laid out earlier in the chapter uses this technique. If it had been coded using the more traditional style of FSM, then all state transition logic would have been performed in the `Control` class, and each state's `CheckTransition()` method would have just have been triggered by input events. For example, in the `StateIdle` state, the `CheckTransition()` function checks whether there is a nearby asteroid, and if so, then checks the distance to it, and then assigns a transition. A classically designed FSM would have done the existence and distance checking from the `Control` class, and passed (or set a Boolean value that the function could check for) the input type `ASTEROID_ASTERIOD_CLOSE_TO_PLAYER`, which the idle class would have then used to assign the transition to the attack state. In this example the transitions are still crisply defined, but they could have a fuzzier determination that takes into account a ramping phase (so that it wouldn't notice the asteroid for some set *reaction time*), or some set minimum time (so that the ship couldn't change its mind out of this state until after the minimum has passed), or any other types of calculations you might want.

By allowing this much more flexible means by which to assign transitions, the code framework opens the door to other, richer methods of assigning transitions. It also keeps some of the proprietary logic calculations within the confines of the state itself, instead of within a large controller class that would perform all the logic within its perception functionality.

TeamLRN

Stack-Based FSMs

Another variation on regular FSM layout is to extend the `m_currentState` member in the state machine class to instead be a stack data structure. As the machine makes transitions from state to state, it keeps a history of the preceding states by pushing them onto the stack. Once a state is completely finished, it is popped off the stack, and the next topmost state is made current again. This allows characters to have a limited form of memory, and their tasks can be interrupted (by a command from another character, or to deal with more pressing concerns, like being shot suddenly), but after the interruption is taken care of, they then return to whatever it was they were doing before.

Care must be taken when using this variant that interruptions clean up any errant stack problems when entering and leaving current status. So, let's say that an AI-controlled character that was in a Patrol state is interrupted by being sniped by the player and immediately switches to a Take Cover state. If the character were hit, it really wouldn't make sense for him to go back to Patrol after the sniping danger is clear. The Patrol state being interrupted by the Take Cover state should actually be flagged as a *replacement* behavior, in that it replaces Patrol as the topmost behavior on the stack. This new state might also want to set an exit behavior, based on whether or not he was wounded, so that the AI will have some state to go to that makes more sense. In that way, when the character comes out of hiding, he won't just blindly start patrolling again but would, instead, call for help (if wounded), or investigate the area that the shot came from. Unless, of course, that's what you want your game to do.

Multiple-Concurrent FSMs

The question of synchronizing or coordinating multiple FSMs is split into two categories: FSMs between multiple characters, and multiple FSMs controlling a single character. Multiple-character coordination is usually handled by a manager of some type, an observer class that gives both characters orders from above and can set up complex scenarios as a puppeteer of sorts. Some games handle this kind of activity with clever use of regular FSM systems that simply play off each other, state-wise, but really don't know anything about each other.

A situation that is a bit uncommon is multiple intracharacter FSM interaction. This requires that characters can be truly doing two things at once. This could be as simple and straightforward as a *Robotron* AI character using one FSM for movement and another for shooting (although these two systems are so completely separate in *Robotron* that it might be better to use a fuzzy state machine here; see Chapter 16, "Fuzzy-State Machines"). It could also be as complex as a series of FSMs running alongside each other for a real-time strategy (RTS) game AI opponent. This opponent would need separate decision state machines for resource

management, research, and combat, and so on. These FSMs might communicate with one another through an observer of some kind (possibly even another FSM, a "general" FSM that uses output from the other FSMs as transition conditions), through a shared data area (like in our AIsteroids FSM implementation), or by passing messages and event data between states and state machines.

Things to watch for in this kind of system would be problems that network code or parallel processing systems encounter. One state machine might overwrite a shared data member that a different state machine needs, two state machines might be in a feedback loop with each other, causing oscillation, there might be an inherent order to some calculations that cannot be guaranteed because of process timing issues, or the like.

Data-Driven FSMs

The push toward more richly defined AI behavior sets has led many developers to think about creating their FSM systems such that their construction is mostly done by nonprogrammers (likely designers and producers). This means that new (or improved) behaviors can be added to the system without much programmer involvement, giving more people on the project the ability to shape gameplay. There have been many different methods for implementing a data-driven FSM system. Some of the more popular ways to accomplish this are by using the following:

- Scripted FSMs, using actual text files, or a simple macro language from within a regular code environment. This is probably the simplest to create, but also calls for a greater technical effort from the designers, especially because most scripting languages end up being subsets of a regular language anyway (most are generally a light version of C, although Python, LISP, or even assembly code style scripting languages are not unheard of). A simplified version of a scripting system might comprise solely generic comparison evaluators (>, <, ==, !, =, etc.), and the script writer would set up the state machine by defining the transition connections between states by using predefined variables and values. Macro languages are a bit simpler to implement than a full language parser is (except for extremely simple languages) and have the advantage of being actual code, making them easier to debug. They have the disadvantages of code as well: your designers now have to compile the game to run their new scripts (as well as obviously requiring the company to buy additional copies of the programming environment), although this is offset by being able to use modern source control tools on these macro files and, hence, provide for things like multiple people working on the same file with automatic merging, as well as setting up protected files that cannot be changed without permission.

TeamLRN

- Visual editors have been written that allow designers to set up FSMs in much the same way as they would prototype them using standard FSM diagrams to show state connectivity and flow of the system. This kind of system is very easy for designers to use, but calls for a much greater commitment to coding than other systems do. The regular game has to be written to expose states, transition conditions, and other information to the editor, so the designers can build the FSM diagrams from these elements as this list grows or changes in the game. In addition to this, the editor itself must be written and maintained over the life of the product (and beyond, in some cases).

Inertial FSMs

One of the problems with FSMs is the concept of state oscillation (as detailed earlier in the chapter). This is caused when the events that cause transitions between states are too close in onset. An example might be a perception in a basketball game that keeps track if a player has an open lane to the basket. This perception *could* be created by doing a line of sight check between the player and the basket, and then checking that line of sight for collisions against all the other team's players. If this check is being performed very often (let's assume you have no optimizations in yet, and it is actually being checked every frame), then you can see how it would be very easy for this player to fluctuate wildly between the `Stand` state, and the `DriveToTheBasket` state because the line of sight collisions might vary slightly on each frame as other players moved about the court. This is exactly the kind of behavior you have to avoid, otherwise your characters will look very twitchy indeed.

The way to combat this is to introduce the notion of *inertia* into the system. This simply means that if a state has been actuated, it stays actuated for some time, or that new states have to overcome some inertia before they can fire in the first place. This can be done at either (or both) of two levels: the states themselves, or the perceptions that fire the states.

At the state level, the state machine itself can keep track of the current state and allow minimum running times (inertia to change, or what could be thought of as the single-mindedness of the AI system: how often does it change its mind), or that oncoming states need to request for promotion to the current state several times before actually becoming the current state (static inertia; analogous to some kind of environmental awareness or what might be called reaction time). In this way, the perceptions would be kept as raw as possible, and the state machine would sample the perception stream to take notice of *trends* (instead of individual data change *spikes*) in the perception variables, and use this to make state changes. At the state level, you could also employ time functions when checking for transitions; the longer the state has been the current state in the machine, the more possible the transitions out of it become. The transitions out still exist, they just become more freely accessible as time goes on.

Inertia at the perception level is precisely the opposite. The state transitions are crisp, but the actuations of perception events are modeled in such a way that they represent the inertia in the system. Perceptions can take multiple firings to actuate (reaction time), require a certain level of perception to fire (sensitivity), continue to keep actuation after the perception has finished (ramp down, or extinction sensitivity), or even require another perception to fire before they themselves will fire, even in the event of the first perception's values becoming true (prerequisite conditions, or cascading actuation).

Inertia from the perception side is sometimes more desirable because perceptions might be shared as triggers across many different states, and so building inertia into a single, commonly used perception might stop oscillation in a large part of the system. But, state side inertia is more general and has the potential to be quicker to implement. A combination of the two methods can be used quite easily to get the exact level of smoothness (versus reactivity) that you want from your system.

Finally, remember that if your AI system requires extreme reactivity (in an action game, for instance, with very fast gaming requirements and instant AI player reactions), you might need to forgo these kinds of decision-smoothing techniques to rely instead on things such as the animation engine to help smooth out twitchy character artifacts. If the animation engine has a degree of inertia built into the blending system, or simply doesn't change the animation for a tick or three when actions change, the AI system could effectively jump around quite a bit and the overall look of the game wouldn't be too harmed. In the end, however, this level of reactivity is rarely necessary because enemies that react at 1/60th of a second (or less) are not usually considered more intelligent and rarely end up being much fun. Now, a boss monster with superhuman reactions, and you have to use an item on him that will slow him down, however . . .

OPTIMIZATIONS

FSMs are easy to code and are most probably the most efficient of all AI methodologies because they logically break the code into manageable chunks, both organizationally and computationally. But there is room for optimization for both the algorithm (in speed of processing) and the overall data structure (for memory usage and such). The common techniques include the following:

Load Balancing Both FSMs and Perceptions

Load balancing refers to spreading the amount of computation to be done over time to lessen the immediate load on the processor. Think of it as buying something on credit: you get the object, but there's an increased cost. In purchasing, that cost

TeamLRN

is interest payments. In our system, the cost is overhead of having to create either time scheduling systems for our AI and Perception systems or having to create incremental algorithms.

Load balancing is generally tackled by one of two ways (both methods working just as well at both the AI and perception level): by having the system run at a set or *scheduled* rate (e.g., twice a second, or every other second), or by having a system that gives incrementally better results the more time it is given. Many pathfinding systems work under the latter system, where they initially just give a rough direction to move toward, then give better and better paths as the time spent in the algorithm increases. Another kind of system along this path is an interruptible FSM system, in which the entire machine can be stopped after a set time limit, and then will start right where it left off when it gets another time slice from the system.

This kind of computational complexity isn't necessary for everything because simple time scheduling will work fine for most perceptions (we're modeling human behavior, and humans' own perception systems rarely work at 60+ frames per second), as well as for general AI decision-making systems (again, humans also rarely change their minds at 60+ fps). If the number of things that need to be scheduled becomes large, a good way to handle the spreading out all the computations is to use an automated load balancing algorithm to try to minimize the spikes in processing that invariably occur, while the system programmer keeps the rough control over update scheduling. These kinds of algorithms keep statistical data on computation times and use extrapolation to predict future needs by the various game elements, and then use this data to determine the order in which to update objects to try to smooth out the processing.

Level of Detail (LOD) AI Systems

Level of detail systems were originally (and still are) used by 3D graphics programmers to ease the amount of work that the rendering pipeline needs to perform, by having objects that are far away be displayed using models comprising fewer polygons and textures because the player won't notice the difference anyway. In some games, where the player can see a very long way off, some LOD systems will actually reduce a game character to a single triangle with a certain color. But because it's so far off, the player can't tell, and the rendering engine isn't spending all the time it would take to compute everything for the 2,000 polygon model that it would usually use for that character.

This same sort of thinking is starting to migrate into AI work because we are now struggling with CPU-intensive AI routines, and we still have a limited player view of the world. So, why not simplify things for the AI when the player might not notice? Instead of generating a real path from A to B using the pathfinding system, a character in another part of the world from the human might just estimate how

long it would take him to get there, and just teleport there after that time was up (a better way would be to teleport there in chunks, to minimize the chances of this behavior screwing things up). Or a retreating character that manages to escape the human player might just get his health back after a set time, instead of actually having to hunt down health powerups and use them. This sounds a bit like cheating, and it can be if overused. But, by simulating (using simple estimations rather than the more expensive methods that are usually used) the effect of things over time, as well as assuring that the human won't run into somebody in the wrong LOD or that the AI uses it too soon after the human is out of view, the feeling of cheating can be mitigated.

The problem with LOD systems in the AI world, as opposed to the graphics world, is that LOD systems for graphics rendering were mostly automatic. Some of them do require special art be worked out for each step of LOD, but others would autogenerate these additional detail levels as well. Then, the graphics engine just had to determine line of sight and distance from the player to determine the correct LOD to display the character at. But in AI programming, LOD usage usually needs to be specially written for each LOD, so it should only be used where there will be a significant savings in genre and gameplay. Consider a game that has dynamic crowds that mill about and interact with the environment. At the closest LOD, the crowd members could use full avoidance, collision response, interact with each other using facial expressions and animations, and spawn other objects like trash that they throw away. At the farthest LOD, they would still probably look pretty good as single polygons that have no collision at all, don't animate, and are simply moving along set path lines laid down in the city.

Shared Data Structures

This is one of the most basic and powerful techniques to optimize FSM computation speed. FSMs (at some level) need a system in which environmental conditions are triggering state transitions, and these conditions may be in some way shared by differing states, so an immediate speedup can be to ensure that different conditions are not recomputed by each state but, rather, are computed in some common area that is shared by the states. This is done in the AIsteroids demo by having some determinations directly in the states' `CheckTransitions()` methods, while having other calculations being done in the `FSMAIControl` structure's `UpdatePerceptions()` function.

Sometimes this functionality is so basic to the engine of the game that an entire framework paradigm, the *blackboard*, is used. This provides a formal way for any game object to publish information to a central data area, and interested objects can request this information or be given an event message with a location to look if they are concerned.

TeamLRN

DESIGN CONSIDERATIONS

Before deciding to plunge fully into a state-based system, you should consider all the factors discussed in Chapter 2, "An AI Engine: The Basic Components and Design" concerning your game, and note the types of systems that FSMs model well: types of solutions, agent reactivity, system realism, genre, content, platform, development limitations, and entertainment limitations.

Types of Solutions

Because of their general-purpose nature, FSMs can be adapted to any kind of solution type, both strategic and tactical. They are most at home with (obviously) *state* types of solutions, however, so note that the more specific the solution you require from your system, the more specific the state will have to be that provides that solution. Or, this means that you will require hierarchical FSMs to achieve more specificity. In general, FSMs really show their power if the number of states in a game is relatively small and the states themselves are much more separate and discrete. A system comprising 400 states that are all the same with small differences is going to incur quite a bit of overhead by an FSM structure, with little benefit.

Agent Reactivity

FSM systems can be tuned to provide the system with any level of agent reactivity because of the simple nature of their processing models. In fact, most FSM systems run fast enough that decision stability needs to be a factor when you build FSMs (discussed with state oscillation in the Cons of FSMs section). The time it takes to make a transition decision by an FSM is practically instantaneous; the real cost is in the perception calculations. This isn't how humans make decisions, however (except for very simple, hardwired behaviors like reflex actions or instinctive acts). Humans are deliberative, have reaction times, and are affected by their environments when making decisions. When an AI makes decisions too fast, it seems robotic and jittery. This type of decisional jitter can be dealt with at either (or both) of two levels: the state machine itself, or at the perception level. Given that FSMs make all their transition determinations as a result of changes in perception, we can stop jitter in the state machine by stopping jitter in the perceptions. You can deal with this by implementing some of the techniques discussed in Chapter 2 under "Input Handlers and Perception," or this chapter's section on Inertial FSMs. Thus, the reactivity of the AI-controlled characters can be explicitly controlled at many levels in an FSM system.

System Realism

FSM-based decision making, unless the FSM system involved is very complex and the modeled behaviors wanted from the system are somewhat narrow, tends to not

be very realistic. FSMs are static, and unless you have a complex hierarchical system that covers every possible event, they will respond in the manner in which the subset of possibility is shown to them through their perceptions. By their very nature, they can only respond to changes in the game with the states they've been provided with. Humans tend to be very good at finding AI patterns of FSM behavior and can locate "missing" perceptions or states that can be exploited by the player very quickly. This might be what your game requires (for instance, in coding the boss monster in a shooter game, the boss might follows a set pattern of states for the duration of the battle, and finding this pattern is the player's key to getting past the boss). Thus, FSM behavior models are usually used for more static behavior sets, or where unchanging lines of reaction are the goal of the system.

Genre

FSMs have been used in every genre of game, again because of their lack of problem-specific context. They thrive in genres with perceptions that can be calculated in simple terms, as well as unique sets of terms, so that the input space can be divided into usable states by the system. Our demonstration program, AIsteroids, is actually not an ideal candidate for FSMs because the gameplay is mostly similar across the whole of each wave (attack everything and get powerups), and the types of behaviors are so similar (usually turning and thrusting toward some target). However, FSMs can be built in such a modular way that they can be used for a given subset of a game's decision structure, and not bleed into the rest of the AI engine. This means that if your game has a specialized element that *is* very state oriented, you can use this type of paradigm for just that part. This is usually the case in most games and is one of the reasons that FSMs find themselves into almost every game in some form or another.

Content

This varies depending on the game being created. Does your game require decision-making elements that follow a state driven flow? Can this additional behavior be split into specific states, that are connected in some way by a system of transitions? If so, then an FSM can be used to control it. But if not, then you might need other types of control structures to capture the behavior of specialized systems that result from specific game content designs. One of the other techniques in this book might be a fit.

Platform

FSMs are also platform independent because they don't have large demands of computing power or memory footprint. Old-style arcade games used to be somewhat more FSM dependant, because of these low demands. In fact, some very old arcade games used actual solid-state logic for their AI opponents (or patterns of enemy movement), and used FSMs in the electrical engineering sense.

TeamLRN

Development Limitations

FSMs lend themselves well to games with heavy development limitations because of their speed of development and debugging. Especially in very short projects, FSMs don't usually have the time to get convoluted by excessive additions and tweaking, which can plague FSM systems in the long run. Also, smaller scale games that only have one AI programmer (or possibly a few) are also good candidates for FSMs, if everything else is a match, of course. It is easier for a limited number of people to remember the changing structure and connectivity of a developing state machine than it is for large teams or extremely separated teams. Additional gameplay elements can be folded into FSMs much more easily than some systems, simply because if you can fit a new state into the state diagram completely, then the system can be coded to incorporate this change. FSM systems are easy to understand by incoming programmers; unlike more exotic AI systems that may require extended learning curve periods by new staff. Quality assurance is also generally quite painless with state based models—behavior is usually quite simple to reproduce, and behavior logs and the like are trivial to implement and use.

Entertainment Limitations

Entertainment concerns, especially difficulty levels and game balancing, are easily handled by state-based systems. If the difficulty level of gameplay is going to change during the game, then this setting itself might be controlled by an FSM that is responding to particular happenings in the game to respond with difficulty level switching. Game balance is made more straightforward because the system requires a state to respond to a change in any given perception state, in effect enforcing a rock-paper-scissors scenario. Thus, if your opponent is coming at you in the Rock state, you should be transitioning to the Paper state. Obviously, this assumes that your FSM model is working under reactive conditions, instead of predictive conditions, but there's no rule that says that the perceptions being fed into the state machine cannot be computed using predictive methods.

SUMMARY

FSMs are the duct tape of the game industry. They are simple, powerful, easy to use, and can be applied to almost any AI problem. However, just like duct tape, the resulting solution may work, but won't be pretty, is marginally hard to extend and modify, and might break if flexed too often.

- A state machine is defined as a list of states, and a structure that defines connectivity between the states given certain conditions.

- The FSM framework given in this book is more modular than most, in that it encapsulates the types of transitions and the transition logic within a single state. Each state is modular because it contains everything it needs to interact with the other states. This also allows more complex transition determinations than the classical input event method.
- The FSM system in this book comprises three main classes: `FSMState`, `FSMMachine`, and `FSMAIControl`.
- Our implemented FSM, in the AIsteroids test bed, uses only five states (Approach, Attack, Evade, GetPowerup, and Idle) to achieve fairly high performance, if a little superhuman.
- Extensions to the test bed for better performance include the addition of states, better math to handle wrapping, bullet management, and better attack and evade maneuvers.
- The pros of FSM systems are their ease of design, implementation, extension, maintenance, and debugging. They are also such a general problem-solving methodology that they can be applied to a broad range of AI issues.
- The cons of FSM systems are organizational informality, inability to scale, and state oscillation problems.
- Hierarchical FSMs allow increased complexity while allowing the overall state machine to maintain a level of organization through grouping. Code and data can also be shared locally to these states, instead of cluttering the global FSM structure.
- Message-based FSMs are great for systems that have a large number of states, or sporadic transition events. This system will broadcast transitional information instead of individual states having to poll perception systems for transition triggers.
- Stack-based FSM variants allow states to be interrupted by more pressing activities, and then returned to by means of the simple "memory" of a state stack.
- Multiple FSMs can control different aspects of a single AI-controlled character and tackle separate portions of the character's decision-making problems but still keep the system simple from an organization point of view.
- Data-driven FSMs using scripts or visual editors are a great way to empower designers to take control of the AI decision flow of a character, as well as add to the speed of creation and the extensibility of the product.
- Load balancing algorithms can be applied to FSM systems, as well as to their perception systems, to achieve more stable CPU usage.
- Level of detail (LOD) AI systems can dramatically reduce CPU usage in games with many AI-controlled characters or large worlds that may be partially hidden to the human player.
- Shared data structures help curtail repetitive condition calculation in transitional logic for the various states in an FSM.

TeamLRN

16 Fuzzy-State Machines (FuSMs)

In the last chapter, we covered finite-state machines, which involved transitions between distinct states, only one of which could be occupying the system at a time. This chapter will cover a variant, but fairly far removed, version of state machines called fuzzy-state machines (FuSMs).

FUSM OVERVIEW

FuSMs are built on the notion of fuzzy logic, commonly defined as *a superset of conventional (Boolean) logic that has been extended to handle the concept of partial truths.* It should be noted that FuSMs build on this notion, but do not represent actual fuzzy logic systems.

The concept of partial truths is a very powerful notion, but unlike regular FSMs, FuSMs are much less general in scope. Like FSMs, FuSMs keep track of a list of possible game states. But, unlike FSMs, which have a singular current state and then respond to input events by transitioning into a different state, FuSMs instead have the possibility of being in any number of their states at the same time, so there are no transitions. Each state in a fuzzy system calculates an *activation level,* which determines the extent to which the system is in any given state. The overall behavior of the system is thus determined by the combination of the currently activated state's contributions.

FuSMs are really only useful for systems that *can* be in more than one state at a time and have more than simple digital values, such as on or off, closed or open, and alive or dead. Fuzzy values are more like halfway on, almost closed, and not quite dead. Another method of quantifying these kinds of value types is to use a unitary coefficient (a number between 0.0 and 1.0) that represents the condition's membership to each end state (0.0 == fully off, 1.0 == fully on, for example), although being unitary is not necessary to the workings of the FuSM. It is simply an easy way to not have to remember proprietary limits on set membership, as well as ensuring ease of comparison between set membership values.

TeamLRN

There is some confusion about what exactly FuSMs are (in the game AI community), because there are several FSM variants that are in the same family as FuSMs. These variants (which will be covered in further detail later in the chapter) include the following:

- *FSMs with prioritized transitions.* In this model, the activation level of each applicable state (this model is still an FSM, so each state still has a list of possible transitions) is computed, and whoever has the highest activation level wins and becomes the new current state. This is how many programmers use the concept of fuzziness to enhance their decision state machines, but the reality is that the system is still an FSM, and the predictability of the behaviors output by a system like this is only mildly smaller than that of a regular FSM.
- *Probabilistic FSMs.* In this form of FSM, there are probabilities placed on transitions out of states, so that the traversal of the FSM is more nondeterministic and thus less predictable. These probabilities could change over time, or could be set within an FSM, with the game using multiple FSMs to group together different probability sets. This is sometimes used when certain transitions have a number of equivalent output states. For example, approaching an enemy might cause him to want to switch to one of three states (of equivalent value): `Punch`, `Kick`, or `HeadButt`. If there is only one output state in a given transition, the FSM functions as normal. But if there are multiple states, then probabilities are assigned to the multiples (either evenly, for total equivalence of choice, or biased toward certain states, or more complex determinations that consider whether one branch was recently taken or if the human keeps blocking a certain move, etc.).
- *Markov models.* These are like probabilistic FSMs, but the transition logic is completely probability based, so they are useful for inducing some change in coupled states. As an overly simplistic example, say you have two states, `Aim` and `FireWeapon`. In this game, these two states are normally totally linked, in that whenever you're done aiming, you will fire your weapon. But, suppose instead you wanted to model a more realistic gun model, and so 2% of the time, `Aim` will instead transition to `WeaponJam`. This type of state transitioning is sometimes referred to (in other fields that use Markov models) as *reliability modeling.* In this example, the weapon is 98% reliable. Markov models are mainly used for these kinds of statistical modeling because one of the assumptions of the system is that the next state is related through probability to the current state. Thus, Markov models are very useful in fields such as risk assessment (in determining rates of failure), gambling (in finding ways to increase house profits), and engineering (to determine the tolerances necessary in fabrication to ensure reliability of the finished product to acceptable levels). A reactive video game may have some elements that fall under this category, but because the

main reasons that AI opponents may be changing states is in answer to the folly of a human player's actions, this kind of state prediction is rarely the norm. An interesting usage of this kind of system might be to actually model the *accidents* expressed by humans occasionally—an AI opponent could occasionally trip, drop the ball, or shoot himself in the foot. All these accidents could be handled at the basic run, hold ball, or shooting action level and could just happen from time to time by taking very unlikely branches in the tightly coupled animations of these activities. Whether or not this kind of realistic behavior fits in your game simulation, or is entertaining to the player at all, is left up to you.

- *Actual fuzzy logic systems.* Contrary to popular belief, FuSMs are not really fuzzy logic systems. Fuzzy logic is actually a process by which rules expressed in partial truths can be combined and inferred from to make decisions. It was created because many real-world problems couldn't always be expressed (with any degree of accuracy) as finite events, and real-world solutions couldn't always be expressed as finite actions. Fuzzy logic is merely an extension of regular logic that allows us to deal with these kinds of rule sets. The simplest form of actual fuzzy logic in games (which is very common), is straightforward `if...else` statements (or their equivalents, through a data table or some kind of combination matrix) that describe changes in behavior. For example, the statement "If my health is low, and my enemy's health is high, I should run away" is a straightforward fuzzy rule. It compares two perceptions (my health and my enemy's health) in a fuzzy manner (low versus high) and assigns it an action (run away). This statement has probably been written as an `if` statement in hundreds of games over the years. This represents the barest minimum of an actual fuzzy system. A real fuzzy logic system would comprise many general fuzzy guidelines for any given combination of my health, my enemy's health, and all the other variables of concern into matrices of rules that will give me a response action through algorithmic combination. This tends to be a powerful way of getting results from a fuzzy system, but suffers when there are many fuzzy variables (each of which may have numerous possible value states or ranges) by creating a quickly unmanageable necessary rule set size, a problem called *combinatorial explosion*. This can be worked around using a statistics technique called Comb's Method, which can reduce the required rule set, but also reduces accuracy.

FuSMs (as well as the previously mentioned similar variants) are rapidly becoming common in game AI usage. The predictability of FSMs is becoming undesirable, and the overall content of many games is becoming rich enough to warrant the additional design and implementation complexity of FuSMs.

FuSMs definitely require more forethought than their finite brothers do. The game problem must really be broken into the most independent elements that the

TeamLRN

problem allows. An FSM could be implemented within the confines of an FuSM system, by calculating digital activation levels and designing the system so that there is no overlap in state execution. Some people do this by accident (or through ignorance) when setting up a fuzzy system. It is much more natural for many problem situations to think in a finite way, so if you are finding it hard to come up with a methodology for FuSMs in your game, then it's probably because you shouldn't be using the fuzzy method in the first place. FuSMs are not as suited to the general range of problems as FSMs are. FuSMs are a kind of FSM that simply allows for the activation of multiple states as the current state, as well as being able to have a *level* of activation equivalent to the degree that the game situation merits each state.

In fact, many people will contend that FuSMs are not even really state machines at all (because the system isn't in a solitary state) but, rather, are more like fuzzy knowledge bases where multiple assertions can be partially true at the same time. But, by coding independent states to take advantage of these multiple assertions, we can use FuSMs to accomplish our AI goals that require this kind of mechanism.

A very simple example of how a system like this might be used would be in coding a decision-making system for an AI-controlled enemy in *Robotron*. An FSM state diagram for a straightforward *Robotron* player is shown in Figure 16.1. There are three main states (this game is very similar to *Asteroids*, so they should look familiar): Approach, Evade, and Attack. In a strict FSM-based system, to move and shoot at the same time, the code would need to be written so that the Approach and Evade states start movement in a particular direction, but don't stop movement when the state is changed. Thus, when the Attack state is in control, the player would still be moving from the last movement state that it was in. This works, but isn't very clean. The Attack state would have to keep checking for transitions to the other states, so that the player wouldn't run into enemies while shooting in another direction, or end up in a corner far away from all the enemies. A better way would be to create a FuSM for this game. Then, the player could Approach, Evade, *and* Attack all at the same time.

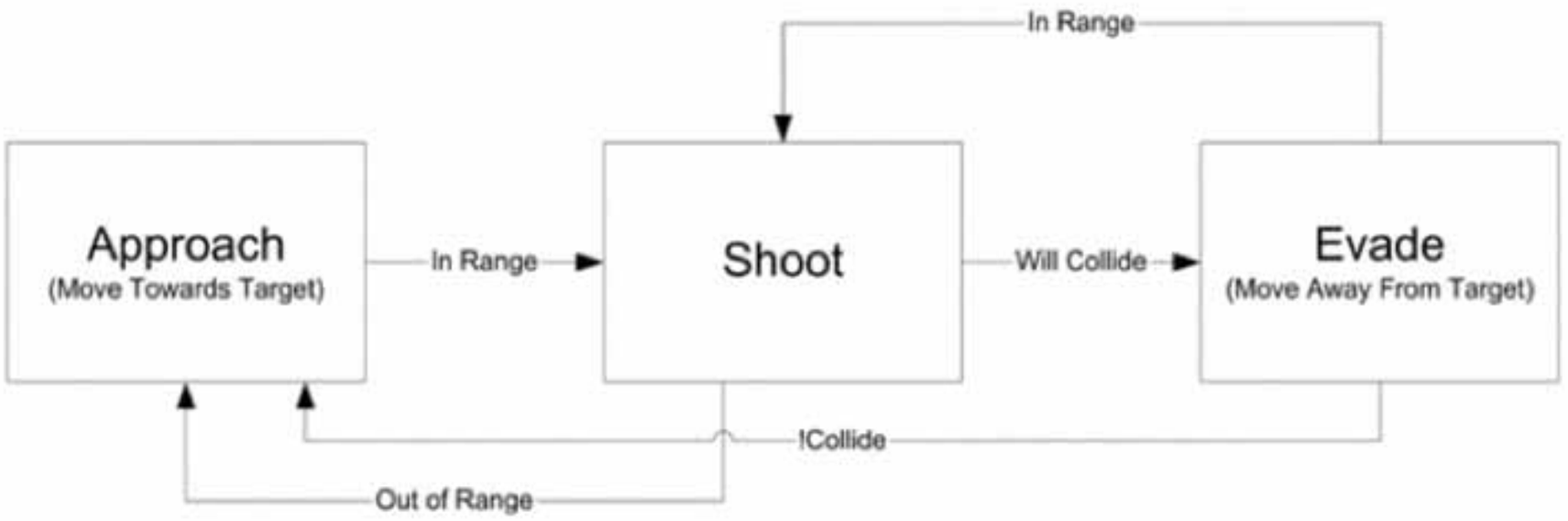


FIGURE 16.1 FSM diagram for a *Robotron* player.

Like FSMs, FuSMs can be written in a free-form way. You could write an FuSM to better accomplish the FSM *Robotron* behavior as shown in Listing 16.1. Here you see the `Update()` function for a *Robotron* player using three different functions that will update if a condition has been met: you've seen code like this if you've written games before because it is very common. The player class encapsulates both the separate methods to handle the different aspects of the behavior and the determination functions that establish whether to use each method. This is fine for relatively simple examples like this one, but in a complex system (consider an real-time strategy [RTS] game where you have an FuSM running the decision-making engine; it would divide the time it has for computation based on the activation levels of each independent decision-making system that needs updating, be it combat, resource, building, strategic, or what have you) you would want to separate this logic into the various modules, making the system more organized, readable, and approachable by more than one programmer at a time.

LISTING 16.1 Update Code for a Free-form FuSM *Robotron* Player

```
void RobotronPlayer::Update(float dt)
{
    float urgency;
    if(CalculateApproachUrgency(urgency))
        Approach(dt,urgency);
    if(CalculateEvadeUrgency(urgency))
        Evade(dt, urgency);
    if(CalculateAttackUrgency(urgency))
        Attack(dt, urgency);
}
```

Another thing to notice about our *Robotron* example is that one of the states, Attack, really can't be completely fuzzy. The player is either shooting, or not shooting, because you cannot partially fire a laser. This is not the case with the other states, where movement can be expressed as a smooth gradient between not moving and moving at full speed. This defuzzification of the Attack state doesn't hurt the rest of the system, however, and doesn't invalidate the method. FuSMs can easily blend in more digital states by having the activation level be calculated in a digital way; the system will still respond to this digital state just like the others.

FUSM SKELETAL CODE

Like FSMs, the code for FuSMs will be implemented in three main classes:

TeamLRN

- The `FuSMState` class, the basic fuzzy *state*.
- The `FuSMMachine` class, the fuzzy state machine.
- The `FuSMAIControl` class, the `AIControl` class that handles the working of the machine, and stores game-specific information and code.

Each of these classes will now be discussed fully.

The `FuSMState` Class

At their most pure level of implementation, states in an FuSM system are wholly disconnected systems. Each state will use perception variables (from the `Control` class, or a more complex and dedicated perception system) to determine *activation level* (which will be represented in this book by a number between 0 and 1), which is the measure of how fully active the state needs to be to respond to the perceptions. In the simplest fashion, the activation level could correspond to the amount of some value in the game, such as aggression; an activation level of 0.0 means you're not aggressive at all, 1.0 means you are completely consumed with aggression. The minimum requirements for an FuSM state are much like an FSM state:

`Enter()`. This function is always run as soon as you enter the state. It allows the state to perform initialization of data or variables.

`Exit()`. This function is run when you are leaving the state and is primarily used as a cleanup task, or where you would run (or start running) any additional code that you wanted to occur on specific transitions (for Mealy-style state machines).

`Update()`. This is the main function that is called every processing loop of the AI, if this state is the current state in the FSM (for Moore-style state machines).

`Init()`. This function initializes the state.

`CalculateActivation()`. This function determines the fuzzy activation level of the state. It returns the value, and stores it in the state as the `m_activationLevel` data member. As you will see later in the chapter, more digital states (such as the attack state in our test bed) can be modeled here by returning Boolean values instead of the normal unitary value.

The header for this class is given in Listing 16.2. Again, this class has been created to be as general as possible to allow for the maximum flexibility in implementing it into your game. As you can see, it is very similar to the FSM class, with the exception of the `m_activationLevel` data member. In fact, this data member could be combined into the FSM class, and a hybrid system could be developed that uses both kinds of states interchangeably.

LISTING 16.2 FuSMState Header

```
class FuSMState
{
public:
    //constructor/functions
    FuSMState(int type = FUSM_STATE_NONE,
              Control* parent = NULL)
       {m_type = type;m_parent = parent;
        m_activationLevel = 0.0f;}
    virtual void Update(float dt){}
    virtual void Enter()          {}
    virtual void Exit()           {}
    virtual void Init()           {m_activationLevel = 0.0f;}
    virtual float CalculateActivation()
                    {return m_activationLevel;}

    virtual CheckLowerBound(float lbound = 0.0f)
            {if(m_activationLevel < lbound)
             m_activationLevel = lbound;}
    virtual CheckUpperBound(float ubound = 1.0f)
            {if(m_activationLevel > ubound)
             m_activationLevel = ubound;}
    virtual CheckBounds(float lb = 0.0f,float ub = 1.0f)
            {CheckLowerBound(lb);CheckUpperBound(ub);}


    //data
    Control*   m_parent;
    int        m_type;
    float      m_activationLevel;
};
```

The class has three bounds checking functions, which are really just floor and ceiling checkers for your activation levels. You can call any of these from your states, or none at all if you want totally raw activation levels.

Like normal FSMs, the class also contains two data members, `m_type`, and `m_parent`. The type field can be used by both the overall state machine and the interstate code, to make determinations based on which particular state is being considered. The enumeration for these values is stored in a file called `FSM.h` and is currently empty, containing only the default `FSM_STATE_NONE` value. When you actually use the code for something, you would add all the state types to this enumeration, and go from there. The parent field is used by individual states, so they can access a shared data area through their `Control` structure.

TeamLRN

The FuSMMachine Class

This class (the header is Listing 16.3), like the equivalent `FSMMachine` class, contains all the states that the machine needs to keep track of. It also contains a list of all the currently activated states, for query purposes. Like the `FSMMachine`, the fuzzy machine is a child of the `FuSMState` class, so that hierarchical FuSMs can be constructed by making a particular fuzzy state be an entire FuSM.

LISTING 16.3 FuSMMachine Header

```
class FuSMMachine: public FuSMState
{
public:
    //constructor/functions
    FuSMMachine(int type = FUSM_MACH_NONE,Control* parent = NULL);
    virtual void UpdateMachine(float dt);
    virtual void AddState(FuSMState* state);
    virtual bool IsActive(FuSMState* state);
    virtual void Reset();

    //data
    int m_type;
protected:
    std::vector<FuSMState*> m_states;
    std::vector<FuSMState*> m_activatedStates;
};
```

`UpdateMachine()`, which runs the general fuzzy machine, is shown in Listing 16.4. As you can see, the system is simple: run each state's `CalculateActivation()` function, separate out the activated states, `Exit()` all the nonactivated states as a group, and then call `Update()` for all the activated states. Although it might seem attractive to simply call the exit or update method for each state in turn, rather than store the states in separate vectors, it would be very restrictive to do so. It needs to be done in this manner because the `Exit()` function from some nonactivated states might reset some things that activated states have turned on or need to change while updating.

LISTING 16.4 FuSMMachine::UpdateMachine() Function

```
void FuSMMachine::UpdateMachine(float dt)
{
    //don't do anything if you have no states
    if(m_states.size() == 0)
        return;
```

```
//check for activations, and then update
m_activatedStates.clear();
std::vector<FuSMState*> nonActiveStates;
for(int i =0;i<m_states.size();i++)
{
    if(m_states[i]->CalculateActivation() > 0)
        m_activatedStates.push_back(m_states[i]);
    else
        nonActiveStates.push_back(m_states[i]);
}

//Exit all non active states for cleanup
if(nonActiveStates.size() != 0)
{
    for(int i =0;i<nonActiveStates.size();i++)
        nonActiveStates[i]->Exit();
}

//Update all activated states
if(m_activatedStates.size() != 0)
{
    for(int i =0;i<m_activatedStates.size();i++)
        m_activatedStates[i]->Update(dt);
}
}
```

The `FuSMAIControl` Class

Finally, Listing 16.5 shows the control class for the FuSM system. It is virtually identical to the FSM control class and contains the global data members necessary to run the system, as well as a pointer to the fuzzy machine structure. In more formalized games, with many global data members, or complex perception update calculations, it would probably be better to create a dedicated perception system (controlled through the control class) instead, but this small list being updated directly with the `UpdatePerceptions()` method is fine for our test application.

LISTING 16.5 `FuSMAIControl` Header

```
class FuSMAIControl: public AIControl
{
public:
    //constructor/functions
    FuSMAIControl(Ship* ship = NULL);
```

TeamLRN

```
    void Update(float dt);
    void UpdatePerceptions(float dt);
    void Init();

    //perception data
    //(public so that states can share it)
    GameObj*    m_nearestAsteroid;
    GameObj*    m_nearestPowerup;
    float       m_nearestAsteroidDist;
    float       m_nearestPowerupDist;
    bool        m_willCollide;
    bool        m_powerupNear;
    float       m_safetyRadius;

private:
    //data
    FuSMMachine* m_machine;
};
```

IMPLEMENTING AN FUSM-CONTROLLED SHIP INTO OUR TEST BED

The AI system necessary to run our AIsteroids main ship doesn't really lend itself to the fuzzy system because you really do have to perform some states to perform others (you have to turn to shoot, but also turn differently to thrust). So, in our test bed example, imagine that we have a second kind of ship, the Saucer, which is dramatically different from our main ship. The Saucer doesn't require turning to thrust. It flies with antigravity, and thus doesn't suffer from inertia or slow acceleration. It can thrust in any direction it wants and has dampeners internally to keep the pilot safe. Because of this amazing ability, it has also been equipped with a gun turret that can fire in any direction. It also has a tractor beam that it can use to drag object toward itself.

This kind of craft has independent systems and is relatively free from having to connect the different parts of its decisions (movement is almost completely separate from attacking, and grabbing objects has also been decoupled), so it is a prime candidate for an FuSM system to run it. Given some basic perceptions, each system (guns, engines, tractor beam) can operate independently, and concurrently. Thus, our ship will no longer be using a state system, in that it is progressing from one state to another but, rather, will operate under the fact that each independent activity will be controlling whether or not it is contributing to the overall behavior of the ship.

EXAMPLE IMPLEMENTATION

In the following sections, the necessary classes to implement the Saucer and an FuSM controlling its behavior will be introduced and fully described.

A New Addition, The Saucer

The Saucer is the game implementation of the new ship type (see the header in Listing 16.6). As you can see, it is very similar, although the `GetClosestGunAngle()` method just returns the passed in angle because the turret can fire in any direction.

LISTING 16.6 Saucer Header

```
class Saucer : public Ship
{
public:
    //constructor/functions
    Saucer(int size = 7);
    void Draw();
    void Init();

    //bullet management
    virtual void Shoot();
    virtual float GetClosestGunAngle(float angle)
                                        {return angle;}
};
```

Other Game Modifications

To allow the saucer to work, several other systems were included. The base ship class was given controls to deal with the tractor beam and the AG thruster (anti-gravity, or noninertial drive). It was also given a vector for the direction of the AG drive `m_agNorm`. This vector can be assigned in two different ways: you can use `AGThrustOn(vector)` to turn on the drive and set the direction to the normalized value of the passed in vector, or you can use `AGThrustAccumulate(vector)`, which will turn on the drive but then *add* the vector into the `m_agNorm` variable. It will then be normalized as it is used by the ship's update method for movement. This is an important part for the fuzziness of the system. Each state that requires movement will use the `AGThrustAccumulate()` method to request ship movement and will scale the vector it will pass in by multiplying it by its current activation level. By doing this, a state with a high activation level will contribute more to the ship's direction of movement than will a state with a low activation level. The base class ship `Update` function then checks whether the AG drive is turned on, and if so, applies

TeamLRN

the `m_agNorm` vector to the position of the ship, thereby giving it instant acceleration and the ability to ignore inertia.

Another addition to the code is the new `GameSession::ApplyForce()` functions. This function is overloaded twice, the first takes as parameters an object type, a force vector and a delta time to apply the force. It will run through the game's object list and add the force to any objects of the types passed in. The second `ApplyForce()` method takes an object type, a force line, the force vector, and a delta time to apply the force. We will be using this method to simulate the tractor beam, as it first checks if the object has collided with this force line before it will apply the force.

The FuSM System

In Figure 16.2, you can see the diagram of the FuSM. Unlike the FSM implementation for the asteroids game, there are only four states instead of five. An FSM system is essentially a closed loop and must have a current state at all times. In the FSM implementation, the Idle state worked as the primary branching point for all the other states in the system, serving as the state of last resort. But, an FuSM can run any number of states (including none), so this state isn't necessary in the fuzzy system. As seen in Figure 16.2, these basic states are the following:

- `Approach`, which will get the ship within range of the closest asteroid.
- `Attack`, for the saucer, is merely firing the guns in the direction of the nearest asteroid. The ship had forward firing weapons and needed to turn and face its target, but the saucer has a gun turret.
- `Evade`, which will initiate avoidance of an asteroid on a collision course by monitoring the ship's speed.
- `GetPowerup`, which will try to scoop up powerups within some range. Unlike the ship, however, the saucer has a tractor beam that it will use to grab the powerups.

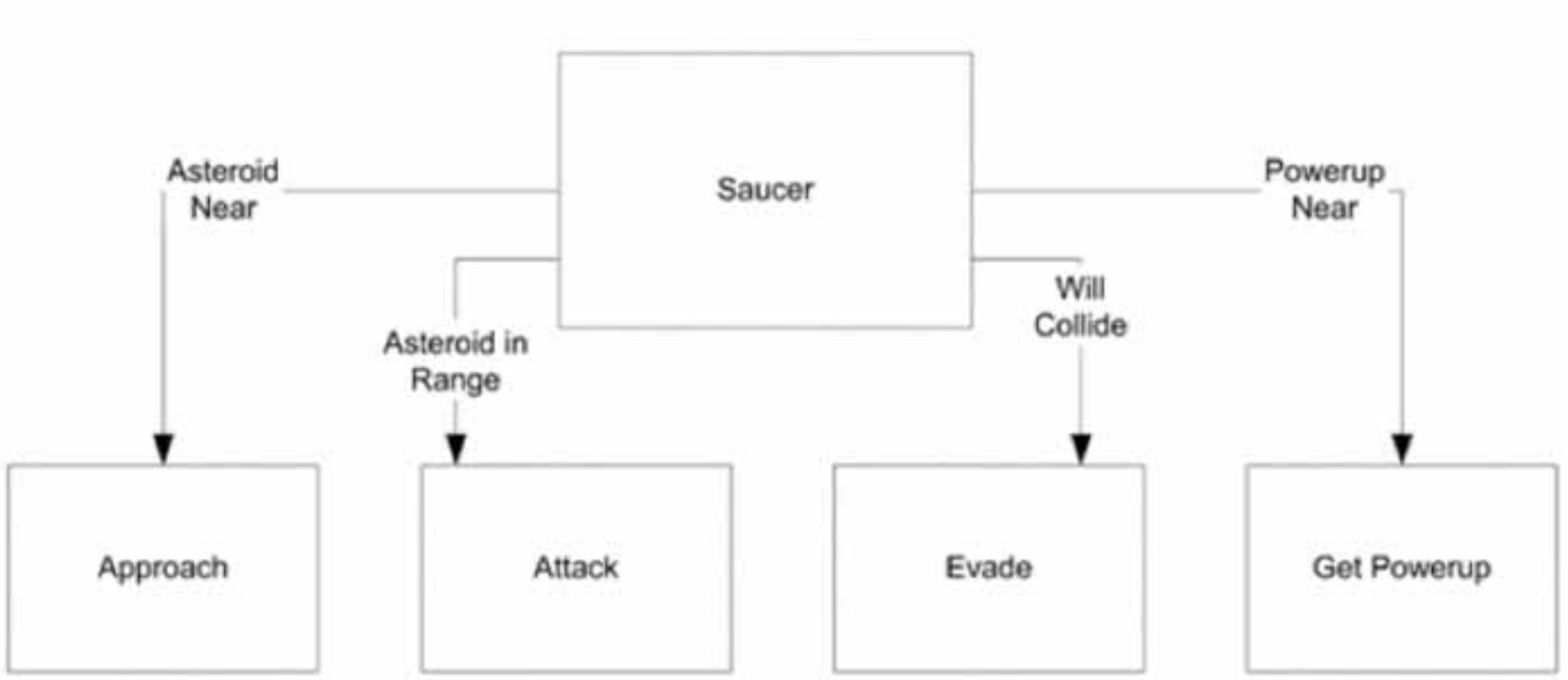


FIGURE 16.2 FuSM diagram for the asteroids game.

The FuSM requires a few bits of data so it can calculate each state's activation level. These are the following:

- The distance to the nearest asteroid is used to determine the activation of three of the states, `Approach`, `Evade`, and `Attack`. The closer an asteroid is, the more the craft will evade and attack; the further away, the greater the activation of the approach behavior.
- The distance to the nearest powerup. This affects the activation of the `GetPowerup` state. The closer the saucer is to the powerup, the more it will try to get it.

There are a few things to notice about the system. Each fuzzy state has no information about other states in it. Each state is only concerned with the perception checks that directly deal with itself only. In the FSM implementation, almost every state needed to watch for the `m_willCollide` field to be true, to transition to the evade state. The reduction of redundant state transition checks that are found in the finite system is an important observation. Many of the states in our asteroids FSM example were interconnected because of the somewhat even priority rating of all the states in the FSM. If you find that your FSM is employing an almost completely connected state diagram, your system may be a good candidate for an FuSM. This is not always the case, but if your game can traverse from any state to any other, the likelihood is that there isn't too much in the way of prerequisite, linear behavior being exhibited by your system.

CODING THE CONTROL CLASS

The controller class for the FuSM model (see Listing 16.7 for the header, Listing 16.8 for the implementation of the important functions) contains the state machine structure, as well as the global data members for this AI model.

LISTING 16.7 `FuSMAIControl` Class Header

```
class FuSMAIControl: public AIControl
{
public:
    //constructor/functions
    FuSMAIControl(Ship* ship = NULL);
    void Update(float dt);
    void UpdatePerceptions(float dt);
    void Init();
```

TeamLRN

```
    //perception data
    //(public so that states can share it)
    GameObj*    m_nearestAsteroid;
    GameObj*    m_nearestPowerup;
    float       m_nearestAsteroidDist;
    float       m_nearestPowerupDist;
    bool        m_willCollide;
    float       m_safetyRadius;

private:
    //data
    FuSMMachine* m_machine;
};
```

The fuzzy control class is much simpler, from a perception point of view. This is attributed to breaking the rules of asteroids, however (such as the saucer having no inertia, a gun turret, and a tractor beam), not because we're using an FuSM. It is simply easier, mathwise, to get the saucer to move to and avoid specific locations because it doesn't have to worry about its own velocity as much.

The FSM AI data member `m_powerupNear` is no longer necessary; it was more of an event trigger that the FSM could respond to, but the fuzzy system uses the distance from the powerup to directly relate to the activation level of the `GetPowerup` state.

The `Update()` method is exactly the same as in the FSM implementation. It won't run the controller if there is no ship to control, and it simply updates the perceptions and the fuzzy machine itself.

LISTING 16.8 `FuSMAIControl` Important Function Implementations

```
FuSMAIControl::FuSMAIControl(Ship* ship):
AIControl(ship)
{
    //construct the state machine and add the necessary states
    m_machine = new FuSMMachine(FUSM_MACH_SAUCER,this);
    m_machine->AddState(new FStateApproach(this));
    m_machine->AddState(new FStateAttack(this));
    m_machine->AddState(new FStateEvade(this));
    m_machine->AddState(new FStateGetPowerup(this));
}

//--------------------
void FuSMAIControl::Update(float dt)
```

```
{
    if(!m_ship)
    {
        m_machine->Reset();
        return;
    }

    UpdatePerceptions(dt);
    m_machine->UpdateMachine(dt);
}

//--------------------
void FuSMAIControl::UpdatePerceptions(float dt)
{
    if(m_willCollide)
        m_safetyRadius = 30.0f;
    else
        m_safetyRadius = 15.0f;

    //store closest asteroid and powerup
    m_nearestAsteroid = NULL;
    m_nearestPowerup  = NULL;
    m_nearestAsteroid = Game.GetClosestGameObj(m_ship,
                                GameObj::OBJ_ASTEROID);
    if(m_ship->GetShotLevel() < MAX_SHOT_LEVEL)
        m_nearestPowerup  = Game.GetClosestGameObj(m_ship,
                                GameObj::OBJ_POWERUP);

    //asteroid collision determination
    m_willCollide = false;
    if(m_nearestAsteroid)
    {
        m_nearestAsteroidDist = m_nearestAsteroid->
                m_position.Distance(m_ship->m_position);
        float adjSafetyRadius = m_safetyRadius +
                               m_nearestAsteroid->m_size;

        //if you're too close,
        //flag a collision
        if(m_nearestAsteroidDist <= adjSafetyRadius )
            m_willCollide = true;
    }
```

TeamLRN

```
        //powerup near determination
        if(m_nearestPowerup)
            m_nearestPowerupDist = m_nearestPowerup->
                        m_position.Distance(m_ship->m_position);
    }
```

Coding the Fuzzy States

The four states' implementations (Listings 16.9–16.12) will be discussed separately in the following sections.

FStateApproach

This state merely computes the vector to the closest asteroid and uses it as a thrust vector for the antigravity drive of the saucer. There's no magic here; the antigravity drive simply works as discussed earlier by directly affecting position instead of acceleration.

The `CalculateActivation()` method returns a zero if there aren't any nearby asteroids; otherwise it returns a normalized value that is between 0.0f (when the distance to the asteroid is almost zero) and 1.0f (when the distance is at or above `FU_APPROACH_DIST`). The `CheckBounds()` call ensures that the activation value falls in this range.

Finally, the `Exit()`function stops the AG drive because this is the only mode that the state dealt with.

LISTING 16.9 `FStateApproach` Implementation

```
//--------------------
void FStateApproach::Update(float dt)
{
    //turn and then thrust towards closest asteroid
    FuSMAIControl* parent = (FuSMAIControl*)m_parent;
    GameObj* asteroid = parent->m_nearestAsteroid;
    Ship*    ship     = parent->m_ship;
    Point3f deltaPos  = asteroid->m_position -
                        ship->m_position;

    //move there
    ship->AGThrustAccumulate(deltaPos*m_activationLevel);

    parent->m_target->m_position = asteroid->m_position;
    parent->m_debugTxt = "Approach";
}
```

```
//--------------------
float FStateApproach::CalculateActivation()
{
    FuSMAIControl* parent = (FuSMAIControl*)m_parent;
    if(!parent->m_nearestAsteroid)
        m_activationLevel = 0.0f;
    else
        m_activationLevel = (parent->m_nearestAsteroidDist -
         parent->m_nearestAsteroid->m_size)/FU_APPROACH_DIST;
    CheckBounds();
    return m_activationLevel;
}

//--------------------
void FStateApproach::Exit()
{
    if(((FuSMAIControl*)m_parent)->m_ship)
        ((FuSMAIControl*)m_parent)->m_ship->StopAGThrust();
}
```

FstateAttack

This state is also a bit simpler than the FSM version. Again, the saucer doesn't have to turn like the regular ship, so all it needs to do is calculate a leading angle and fire.

The activation function for this state is digital, either zero or one, because you cannot partially fire a gun at something. The state is simply on if there is an asteroid and it is within firing range, or it is off.

There is no `Exit()` method for this state because the shoot command is not an on/off toggling command. It only fires one shot at a time.

LISTING 16.10 `FStateAttack` Implementation

```
//--------------------
void FStateAttack::Update(float dt)
{
    //turn towards closest asteroid's future position, and then fire
    FuSMAIControl* parent = (FuSMAIControl*)m_parent;
    GameObj* asteroid = parent->m_nearestAsteroid;
    Ship*    ship     = parent->m_ship;

    Point3f futureAstPosition = asteroid->m_position;
    Point3f deltaPos = futureAstPosition - ship->m_position;
    float dist  = deltaPos.Norm();
```

TeamLRN

```
        float time = dist/BULLET_SPEED;
        futureAstPosition += time*asteroid->m_velocity;
        Point3f deltaFPos = futureAstPosition - ship->m_position;

        float newDir = CALCDIR(deltaFPos);
        ship->Shoot(newDir);

        parent->m_target->m_position = futureAstPosition;
        parent->m_debugTxt = "Attack";
    }

    //--------------------
    float FStateAttack::CalculateActivation()
    {
        FuSMAIControl* parent = (FuSMAIControl*)m_parent;
        if(!parent->m_nearestAsteroid)
            m_activationLevel = 0.0f;
        else
            m_activationLevel = parent->m_nearestAsteroid &&
               parent->m_nearestAsteroidDist < FU_APPROACH_DIST;
        return m_activationLevel;
    }
```

FstateEvade

This state follows suit with the other movement states. It calculates a vector away from the nearest asteroid and sets up the AG drive to thrust in that direction.

Its activation level goes up the closer it gets to the nearest asteroid, to simulate getting more single minded about evasion as it closes in on a collision.

It turns off the AG engine when exiting, like other states that use the antigravity system.

LISTING 16.11 FStateEvade Implementation

```
    //--------------------
    void FStateEvade::Update(float dt)
    {
        //evade by going away from the closest asteroid
        FuSMAIControl* parent = (FuSMAIControl*)m_parent;
        GameObj* asteroid = parent->m_nearestAsteroid;
        Ship*    ship     = parent->m_ship;
        Point3f vecBrake = ship->m_position - asteroid->
                                                m_position;
        ship->AGThrustAccumulate(vecBrake*m_activationLevel);
```

```
    parent->m_target->m_position = parent->
                                m_nearestAsteroid->m_position;
    parent->m_debugTxt = "Evade";
}

//--------------------
float FStateEvade::CalculateActivation()
{
    FuSMAIControl* parent = (FuSMAIControl*)m_parent;
    if(!parent->m_nearestAsteroid)
        m_activationLevel = 0.0f;
    else
        m_activationLevel = 1.0f - (parent->
                            m_nearestAsteroidDist - parent->
                            m_nearestAsteroid->m_size)/
                            parent->m_safetyRadius;
    CheckBounds();
    return m_activationLevel;
}

//--------------------
void FStateEvade::Exit()
{
    if(((FuSMAIControl*)m_parent)->m_ship)
        ((FuSMAIControl*)m_parent)->m_ship->StopAGThrust();
}
```

FstateGetPowerup

Unlike the normal ship, the saucer is equipped with a powerful tractor beam that drags powerups toward itself when activated. It still will approach the powerup, and the urgency of the approach will be controlled by the state's activation level. The state will also turn on the tractor beam to drag the powerup in.

The activation calculation method is much like the `FStateEvade` state, in that the closer to the powerup, the stronger the activation. This is so that the saucer will make more of an effort (with its maneuvers) to pick up the powerup if it is very close by. Otherwise, the tractor beam will do most of the work.

The `Exit()` method needs to turn off both the tractor beam and the AG engine because it uses both.

LISTING 16.12 `FStateGetPowerup` Implementation

```
//--------------------
void FStateGetPowerup::Update(float dt)
{
```

TeamLRN

```
    FuSMAIControl* parent = (FuSMAIControl*)m_parent;
    GameObj* powerup = parent->m_nearestPowerup;
    Ship*    ship    = parent->m_ship;

    Point3f deltaPos = powerup->m_position -
                       ship->m_position;

    ship->AGThrustAccumulate(deltaPos*m_activationLevel);
    ship->TractorBeamOn(-deltaPos);

    parent->m_target->m_position = powerup->m_position;
    parent->m_debugTxt = "GetPowerup";
}

//--------------------
float FStateGetPowerup::CalculateActivation()
{
    FuSMAIControl* parent = (FuSMAIControl*)m_parent;
    if(!parent->m_nearestPowerup)
        m_activationLevel = 0.0f;
    else
        m_activationLevel = 1.0f - (parent->
                    m_nearestPowerupDist - parent->
                    m_nearestPowerup->m_size)/
                    FU_POWERUP_SCAN_DIST;
    CheckBounds();
    return m_activationLevel;
}

//--------------------
void FStateGetPowerup::Exit()
{
    if(((FuSMAIControl*)m_parent)->m_ship)
    {
        ((FuSMAIControl*)m_parent)->m_ship->StopAGThrust();
        ((FuSMAIControl*)m_parent)->
                                m_ship->StopTractorBeam();
    }
}
```

PERFORMANCE OF THE AI WITH THIS SYSTEM

With the FuSM system in place, as well as the much more lenient gameplay rules that the saucer has to follow, it is all but unstoppable at destroying the asteroids in

the test bed game. It will play as long as you let it, and it has survived several *hours* of continuous play in testing. Figure 16.3 shows the saucer going to work. It does still die occasionally and could be made completely unstoppable with the same kinds of improvements that would help the FSM system:

- Increase the complexity of the math model to give the AI system the ability to deal with the world coordinates wrapping. Right now, the AI's primary weakness is that it loses focus when things wrap in the world, so accounting for this during targeting and collision avoidance would greatly increase the survivability of the AI ship. Even this weakness is considerably lessened by the saucer's capabilities over the regular ship because it never floats across a border like the ship does.
- Bullet management for the ship. Right now, it just points, and then starts firing. There is no firing rate on the guns, so it tends to fire clumps of shots toward targets. This is somewhat advantageous; when it fires a clump of shots into a large asteroid, the remaining shots will sometimes kill the pieces as the asteroid splits. But this can get the ship in trouble when it has fired its entire allocation of bullets, and must wait for them to collide or expire before it can shoot again, leaving it temporarily defenseless.

FIGURE 16.3 FuSM implementation of the AIsteroids test bed.

TeamLRN

Pros of FuSM-Based Systems

FuSMs are very straightforward to design, for the right problems. If your AI situation involves independent, concurrent systems, then this model allows you to design the separate systems as just that: separate systems without any concern for each other. Therefore, you don't incur the effort of designing the transition events and links between states that FSM systems require. The model provides a simple way in which to activate each state according to a scale that you can define for the particular problem. FuSMs also allow digitally activated states to be mixed in freely with the more fuzzy ones by simply setting up the activation calculator to return digital values.

Implementation of an FuSM system is even easier than FSMs because of the lack of transitions. Each state can be implemented in a pure vacuum, with only the global perception data (stored in the control class) as the glue holding the system together.

Extending a fuzzy system is as uncomplicated as finding other states that will freely mix with the system. In our asteroids example, another state could be added to aid evasion in the form of a repulsion beam, the opposite of the tractor beam. This would shoot out from the ship and deflect incoming asteroids. Adding a state that controlled the use of the repulsion beam to the FuSM would be almost effortless; by copying the `GetPowerup` state and changing a few lines to affect the nearest asteroid instead of powerups and changing the direction of the force that will be applied to the rocks.

Debugging a fuzzy system is also quite straightforward. Because of the uncoupled nature of the states, you can disable any that you are not concerned with at the time, and then concentrate on the remaining active states. You can see how minimal the evasion code for the saucer is by disabling the attack state. The saucer will try to evade the rocks, but because it is taking only one asteroid into account at a time, it will invariably become surrounded and be crushed. To extend the abilities of the craft, advanced evasion techniques (possibly involving moderate pathfinding or some form of influence map analysis) could be implemented and tested, without having to worry about the very efficient attack behavior mowing everything down and clearing the way for the saucer.

FuSMs scale very well, again because of the disconnected nature of the states. The only problem that you have to deal with is the notion of too much blending, which might lead to very average or muddy behaviors on the whole. Say that our test bed not only had its current approach, evade, and get powerup behaviors vying for the movement of the ship, but also had states trying to dock with floating bases, maneuvering for the use of transportation gates of some kind, responding to formation requests from other friendly saucers, and maybe even responding to emergencies like wormholes. Eventually, so many states would be affecting the direction

of thrust for the AG drive that the ship might not be able to move at all. The more states that are blending into a particular trait of the system, the more diluted each individual state's contribution becomes. This dilution can be overcome by trying to combine states into like-minded groups (the previous example of a transportation gate handling state could possibly be considered a different kind of powerup, and the wormhole handler could be grouped into evade, for instance).

Fuzzy systems allow a much greater range of behavioral personality to be exhibited by your AI-controlled agents. The current FuSM asteroids implementation can be made more "aggressive" by lowering the `FU_APPROACH_DIST` define. By upping the priority of the evasion behavior and raising the overall activation level of the powerup state, you would end up with a more defensive character, which would even appear greedy when powerups were present. Different saucers could be coded using separate classes that redefined the `CalculateActivation()`methods of the various states, or they could use a data-driven interface that would access a list of attributes to tweak the overall mix of behaviors toward specific personality traits.

The FSM problem of state oscillation is nonexistent in the FuSM world. FuSMs can actually be in every state at once, or none at all, so there is no real concept of switching back and forth between states. The problem is somewhat replaced by the notion of behavior oscillation, however, and is discussed in the next section.

Cons of FuSM-Based Systems

FuSMs are not as general a problem solver as FSMs. FSMs are a way of modeling behaviors that happen, one after another, in sequence; they represent a circular, progressive system that allows reactivity, proactive tasking, and prerequisite actions. FuSMs are better suited to a complex behavior system that can be constructed by blending smaller, unconnected behaviors together. This concept of blending is key. FuSMs are uniquely qualified for dealing with gradients of behavior. This is not always the kind of notion that games require, or even want, because subtle behavioral differences are often lost in the fast movement, low graphical resolution, fixed animation, and art assets of the game world. In the future, when advances in facial animation and physics-based movement systems (which would model movement based on the forces acting on a person, rather than a hand made or motion captured animation that is being played by a character) are the norm, FuSMs will be an integral part of bringing the full range of emotion and ambiance to AI-controlled characters. For right now, pure FuSM systems are a niche technique useful for specific groups of behaviors.

Badly designed FuSMs can exhibit behavior oscillation, which was mentioned in the last section. In our asteroids saucer, we don't have to worry because the only states that might fight each other are exact opposites, the approach and evade states. However, they cancel each other out if both states are at maximum values, and the

TeamLRN

ship will sit still. But if approach and evade used nonopposite vectors when in use, and approach wanted to get closer than evade wanted to allow, the ship might behave oddly: it might move in circles or with some kind of cyclical diagonal zigzagging. The way to solve this is precisely the way that our asteroids saucer does: model behaviors like the human body uses its muscles, with complementary yet opposite states that get the job done and work together to mute inconsistencies in activation.

EXTENSIONS TO THE PARADIGM

FuSMs are somewhat misunderstood, as discussed at the start of this chapter. The various methods for which people employ so-called FuSMs are many. Some of the more useful of these extensions and variants will be covered here.

FuSMs with a Limited Numbers of Current States

You might have a system where you want a series of behaviors that have a smooth gradient of activation, but only one or possibly a few behaviors are going to be able to update. FuSMs can be easily extended to treat the activation level of each state as a priority function, and the winner (or some number of the highest priority states) will end up being the only one to update. With a single state, this system becomes more like the FSM with fuzzy transitions variant discussed in Chapter 15, "Finite-State Machines." If you still allow multiple current states, you could think of this method as a means of fighting the dilution problem discussed earlier in the previous section. Particular fuzzy states could be tagged with subtypes, and the highest priority subtype would win for that particular subtype category. In our AIsteroids example, attack would be a subtype, along with movement and tractor beam. So, approach and evade would fight to be the winner of the sole movement state that gets to function. This works to help dilution, but also defuzzies the system because you are taking out additional blended elements into the overall behavior. This type of system might also be employed as a computation cost-saving optimization for games in which this is a concern.

An FuSM Used as a Support System for a Character

Although fully fuzzily controlled characters are somewhat rare (look at how many rules we had to break in the original AIsteroids example to get a good candidate for FuSMs), specific parts of a character might be extremely good places for this method. A facial expression system might be very useful for this kind of scheme. Each state would be a particular emotion: happy (would curl the mouth and squint the eyes), sad (would arch the eyebrows and droop the mouth), mad (bares the teeth, brings together eyebrows, opens eyes), and so on. Each emotion would activate

to a level based on separate perceptions, and the whole system would run concurrently with whatever the rest of the AI system was doing.

An FuSM Used as a Single State in a Larger FSM

Even though not all the states or behaviors a given character employs might be independent or fuzzy, specific sections might. A simple example might be a character that runs a normal state machine while running around the map, getting items and interacting with others. But when he stands still, a fuzzy state might start up that would blend together three separate behaviors: looking around (the shorter time he's been in this environment, the more inquisitive he is about it), fidgeting (the more tasks he has, or the longer he's waited, or the less time since his last enemy encounter, the more nervous he is), and whistling (the more safe he feels, the noisier he'll be when standing around). This state is the FSM's current state, but it will be running any or all of these fuzzy substates to model the standing behavior of the character.

Hierarchical FuSMs

Just like FSMs, FuSMs can easily be made hierarchical. The skeletal code has the `FuSMMachine` class inheriting from the `FuSMState` class to facilitate this. However, this isn't the most useful notion, design wise. Multiple states could be running simultaneously, so there is little reason to group states together, except for organization. If you are combining some of these variant methods, this would be more useful. You could use an FuSM to contain additional FuSMs that use the "limited number of current states" method mentioned earlier. Each sub-FuSM would return the highest priority state within its subtype, and then all the winners would run under the parent FuSM.

Another type of this might be an FSM where each state is an FuSM. This becomes, in effect, a fuzzy system that can switch out its entire fuzzy state system based on game events or perception changes. This is a very powerful and general-purpose system.

Imagine a hierarchical FSM containing states that are either FuSMs (for more dynamic and emergent behavior), or regular FSMs (for more static or semiscripted reactions to game events), giving the programmer the ability to use the exact system that best suits the specific state of the game.

Data-Driven FuSMs

FuSM systems have generally seen a bit less data-driven implementations than FSMs, but they also haven't been used to such a great degree. Data driving an FSM usually means allowing designers some method (either in script or through a visual

TeamLRN

interface of some kind) to set up states and be able to show transition connectivity between the states, as well as assign conditions to the transitions. In FuSMs, the control is changed, in that the designers would instead decide which states they want to add to the total machine (which will become the different elements that are blended to become the end behavior), and then control the activation calculations of each state, either by laying down conditions and simple equations directly, or by affecting a standard calculation with modifiers (such as adjusting the state's activation level boundaries, or by applying some scale factor). This kind of data could be tweaked on a per character level, to get different personality types out of the system, or on a difficulty basis, to affect how behaviors are selected to affect the overall difficulty of the game.

OPTIMIZATIONS

FuSMs have the potential of running many different states concurrently, and so are a bit more computationally expensive than their FSM brothers. FuSMs do not incur the transition calculations of a finite system, but have their own activation computation costs. The same kinds of optimizations that FSMs use apply to fuzzy systems: load balancing, LOD systems, and shared data. See Chapter 15 for the discussion of these techniques.

DESIGN CONSIDERATIONS

FuSMs are good for AI problems that are quite a bit different from their FSM brothers. The checklist of considerations when deciding on an FuSM based system include types of solutions, agent reactivity, system realism, genre, platform, development limitations, and entertainment limitations.

Types of Solutions

FuSMs are another very general problem-solving tool and can be used to implement most kinds of solution types. FuSMs are a bit more paradoxical, in that they work very well for very high-end solution types, and for very low-end solution types. The reason being is that both tend to be organic solutions that combine several elements to achieve a final solution. The two ends of the spectrum also tend to be less numerous, so are also less prone to the dilution problems that can affect FuSMs. More stylized or scripted behaviors (the kinds that end up being in the middle of the road, behavior wise) tend to be more suited to state-based systems because they tend to have a lot of prerequisite activity and are usually quite crisply

determined by the perceptions. A high-level decision maker for an RTS game might combine the output of several fuzzy states such as reconnaissance, resource gathering, diplomacy, combat, and defense to determine its overall activity. An even higher-level decision process could have a *counselor* state for each of these areas, and then blend the advice from these counselors to form an overall decision about how to run the civilization as a whole. Lower-level, or tactical decision-making examples might include blending immediate orders or goals (go here, attack this unit, gather this resource) with secondary states of behavior (motioning to other units for support, combat evasion when you're not a combat unit, fleeing when badly hurt, etc.).

Agent Reactivity

Given a sparsely connected state structure, FuSMs are generally more reactive than FSMs because there isn't a transition structure that the character has to traverse to reach a goal. But, with simple FSMs or interconnected FSMs, there is very little cost difference between the two methods, and almost any level of reactivity can be built into each state of the system. The techniques described in the section on Inertial FuSMs can be used to help tune the level of agent reactivity that your game requires.

System Realism

Games based on FuSMs have a much greater sense of realism because the final behavior of the system is a continuous curve of perception reaction. This feels much more realistic than does a character hitting some threshold and then changing to some other state. A well-designed FuSM will react to perception changes in a realistic manner, by adjusting their current behavior, not completely changing it. Most people respond to a new situation by slightly modifying their ongoing behavior (unless the new situation is life threatening or very shocking, although even then the new behavior is initiated as a delta from what the person was already doing, and this kind of quick change in behavior can be modeled by an FuSM as well).

Genre

Being a fairly general technique, FuSMs will work with any genre of game in at least a limited fashion. When considered as a *primary* game-wide AI framework, they are definitely limited by genre. You wouldn't want to try to implement a linear, scripted game using a fuzzy state system. But even in a game that doesn't require this kind of problem solving generally, there might be a use for the kind of fuzzy determinations that an FuSMs can accord. The perception system of a game could, for example, be written using an FuSM as the framework. Perceptions are usually independent and can usually be coded with very little thought to any other perception. The fact that

TeamLRN

perceptions have arbitrary output values (Booleans, continuous floating point values, enumerated types, etc.) is fine with the FuSM system. An FuSM doing this kind of work would use the different states to represent each perception, with the state's `Update()` method computing the perception value, and the activation level operating as the indicator that the game needs to update the perception. All the secondary perception calculations, such as reaction time, load balancing, and so on could be handled through the `CalculateActivation()` function, although truly time scheduled updates could be handled with special data members of the `FuSMState` class, which could keep records for any scheduling system, so that the fuzzy machine could decrement timers or determine triggers for updating states.

Platform

The memory and CPU requirements for FuSMs are minimal, and so FuSMs are generally platform independent. However, they do lend themselves to more subtle behavior, which is usually the realm of PC games. Whether to use them or not is usually much more a game design concern.

Development Limitations

If your AI problem falls into the kinds of situations that FuSMs handle well, then there is no better means by which to implement them. FSMs are easy to understand and implement, but FuSMs are not much more difficult and provide a much richer and more dynamic product. FuSMs are just as straightforward to debug as FSMs because even though they have a greater range of behavioral outputs, they are still deterministic.

Entertainment Limitations

Tuning difficulty settings, balancing specific behaviors, and other entertainment concerns are generally quite simple with FuSMs. They can be tuned from a state-by-state basis, at the perception level, or any combination. Some behaviors might have a synergistic effect with another behavior (such as the attack state's ability to bail out the simplistic evade state in the AIsteroids implementation), and make some tuning a careful affair, but usually individual states can be tuned separately.

SUMMARY

FuSMs build on the straightforward FSM system, by allowing complex behaviors that can be broken into separate, independent actions to be constructed by blending these actions together at different levels of activation. This powerful extension

to the FSM concept gives the FuSM method the ability to create a much broader range of output behavior, but adds the requirement of this style of aggregate behavior building.

- The definition of FuSMs is somewhat hazy, with confusion existing between real FuSMs and similar systems such as FSMs with fuzzy transitions, probabilistic FSMs, Markov models, and actual fuzzy logic systems.
- FuSMs do not use a single current state but, rather, can have any number of active states, each with a variable level of activation when active.
- Some states in an FuSM can have digital activation levels, and this defuzzification of some part of the system is fine and will not affect the overall method.
- The skeletal FuSM framework discussed in this book is built on three base classes: `FuSMState`, `FuSMMachine`, and `FuSMAIControl`.
- The original game doesn't fit well into the FuSM model, we added a new ship class, the `Saucer`, that flies with antigravity (no inertia or acceleration), has a gun turret that can fire in any direction, and a tractor beam to drag powerups toward itself. This provides us with a much more ideal candidate for an FuSM control structure because the saucer uses mostly independent systems, most of which have variable levels of activation.
- The implementation of an FuSM into the AIsteroids test bed needs only four states: `Approach`, `Attack`, `Evade`, and `GetPowerup`. Its state implementations are much simpler than those of the FSM system, and the perception calculations are also simpler, but this is more because of the saucer breaking some of the game rules that the regular ship was following, rather than because of the switch in AI techniques. However, the saucer is superior to the FSM implementation in performance and can play almost indefinitely.
- Extensions to the AIsteroids game for better performance would be to figure world wrapping into attacking and evasion, and bullet management routines.
- The pros of FuSM systems are their ease of design (for the right style of problems), implementation, extension, maintenance, and debugging. They allow a much greater range of behavioral personality and do not suffer from the FSM problem of state oscillation.
- The cons of FuSM systems are that they are not as general a solution system as FSMs are, and they can have behavioral oscillation problems if designed poorly, but this can easily be countered with forethought.
- FuSMs with a limited number of current states can be written to tune the level of fuzziness you want to use in your game. You can have one current state, a few, or limit current states within subtypes of states.
- An FuSM used as a support system for a character is a great way of adding fuzziness only where it is needed in the implementation of complex characters, such as in a facial expression system.

TeamLRN

- An FuSM used as a single state in a larger FSM can be used to represent a character that has very fuzzy behavior determination, but only within the confines of a larger finite game state.
- Hierarchical FuSMs are usually quite rare in their most pure form because they don't make much sense, but when combined with other state machine variants, their true power begins to be seen.
- Data driving FuSMs involves designer control over the particular states a character might use, as well as affecting activation level calculation.
- FuSMs can benefit from the same kinds of optimizations used in regular FSMs.

17 Message-Based Systems

In the world of modern game programming, only one technique is used more than state machines. That technique is the use of messaging (or events, as they are also called). The concept of messaging is simple. Instead of game entity A checking game entity B for particular changes every tick, or even on some time schedule, A is informed of changes from a *message* that is delivered to A from B only when the change has occurred. This means that nobody has to waste computation cycles or code space by making checks throughout the game engine to determine if things are happening. The game informs the entity with messages about the kinds of occurrences it is interested in, and then goes about its merry way, not worrying about it until another message comes in for delivery.

MESSAGING OVERVIEW

Unlike many of the other techniques discussed in this book, messaging is not a decision-making structure, per se. It is more of a communication technique that can be used in the game to help with organization, optimization, and ease of communication between disparate objects and classes in the game. Messaging serves as a secondary system that resides below the underlying decision structure of your game. It is used in games where this type of communication is cost effective, and as games become more complex, that category is growing. Most modern games can benefit from using messaging systems in their engines.

AI systems have two main traits that make them good candidates for using message-based communication:

- AI controlled characters are most often created to be *reactive*, in that they regularly depend on an outside perception change to affect the behavior of the character. This makes sense; we are reacting to the human player interacting with the game. But this also means that AI systems can do a lot of waiting for perception changes or perform many computations determining those perception changes. AI characters might be completely inactive, especially if they

TeamLRN

are not visible to the human player, for large chunks of gameplay time. Spending time performing calculations during these periods would be wasteful.

- AI is a very *high-level* part of game development. The AI programmer might have to communicate with many other game systems (including animation, character and world physics, gameplay, controls, sound, etc.) when creating the condition checks and behaviors for the AI system. Without some form of abstraction when performing this communication between parts of the engine, your game will be strapped with an AI system that has access into every other area in the game engine. Although this gives the AI programmer a lot of power (to do harm as well as good), this is generally considered bad programming methodology and can lead to unmaintainable systems that are all but impossible to extend, debug, and understand.

Messaging is uniquely qualified to conquer these issues. It creates a system that is completely reactive because the system is only responding to event messages. It also decouples data from the code, so that AI systems can request data from other areas in the game, and not have to have full access to the underlying class structure of those systems. It provides a central way of moving data and events between AI code sections and the greater game, so that the underlying AI system can change, without having to recode the entire process of getting information from the rest of the game.

This chapter will lay the framework for a general case event messaging system that you can use for your entire game, or for parts of a game. This general framework will comprise three main parts: a *message object,* the *message pump,* and *client handlers.* See Figure 17.1 for a visual representation of this architecture.

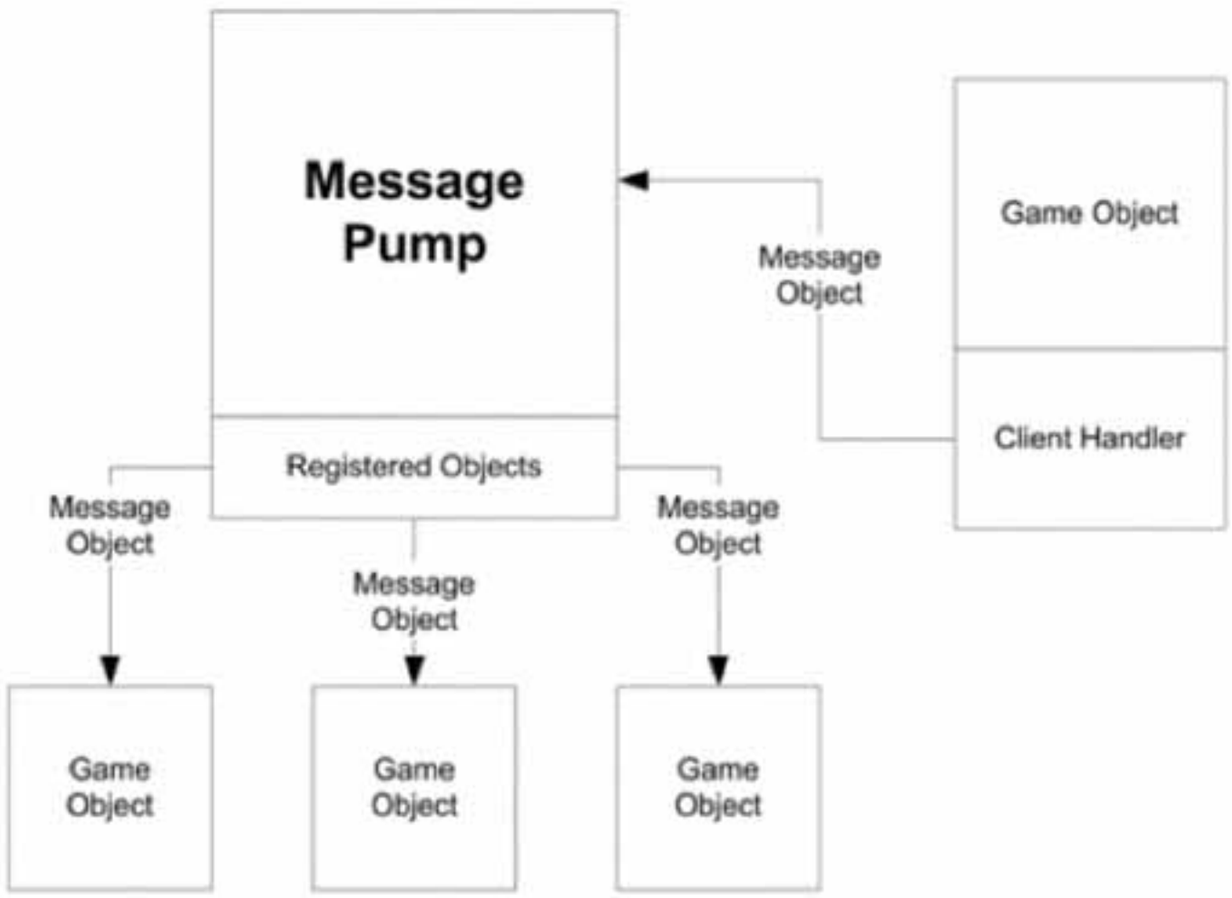


FIGURE 17.1 Messaging system overview.

MESSAGING SKELETAL CODE

The general messaging system this chapter will introduce is implemented using these base classes:

- The `Message` class that stores the individual information requirements of a message.
- The `MessagePump` class, which is the central message router.
- Client handlers, which run code to accomodate any given incoming message.

In the following sections, each of these classes will be fully discussed.

The Message Object

The message object is a general structure that is used to store a message. You can see that the header in Listing 17.1 contains only a few data fields:

- `m_typeID` is the type of message.
- `m_fromID` is the unique ID of the object that sent the message. This is an optional data field for a message because messages can be sent anonymously.
- `m_toID`, is the ID of the object that the message is to be delivered to. Again, this is an optional field because the message pump can also have messages that have been registered for delivery by a given object, so the message itself doesn't need to specify.
- `m_timer` is used for setting delays in delivering messages.
- `m_delivered` is used by the message pump to mark messages that have been processed so that they can be removed from the queue.
- `DataMessage` is also included in the file and is a simple template class that has a single data field of whatever type you pass in. You can use this class to pass messages with simple data fields, but you'll have to implement additional types of messages if you require more complex data sending. In any case, a message handling callback function would just cast the incoming message to the type it knows it is (from the message ID type) and access the data through the cast pointer.

LISTING 17.1 Message Class Header

```
class Message
{
public:
    //constructor/functions
    Message(int type = MESSAGE_DEFAULT){m_typeID =
                type;m_delivered = false;m_timer = 0.0f;}
    ~Message(){}
```

TeamLRN

```
    //data
    int   m_typeID;
    int   m_fromID;
    int   m_toID;
    float m_timer;
    bool  m_delivered;
};

//simple template message class for simple data passing
template <typename T>
class DataMessage: public Message
{
public:
    DataMessage(int type, T data):Message(type){m_dataStorage = data;}
    ~DataMessage(){}

    //data member
    T m_dataStorage;
};
```

The MessagePump

The messagepump is the class that will store all the possible message types, as well as be the central location that messages are delivered to and from. The messagepump will keep track of delayed messages, broadcast messages to interested objects, and generally act as the post office for the system. Listing 17.2 shows the header for the class, and Listing 17.3 shows the important implementation.

As the header shows, we will be implementing the messagepump as a singleton, which is a software design pattern that just means that there will be only one instance of this class for the entire game. The #define at the bottom of the header provides clean access to the singleton class structure.

LISTING 17.2 MessagePump Header

```
typedef std::list<Message*> MessageList;
typedef std::map<int,MessageType*> MessageTypeMap;

class MessagePump
{
public:
    static inline MessagePump& Instance()
    {
```

```
        static MessagePump inst;
        return inst;
    }

    static void Update(float dt);
    static void AddMessageToSystem(int type);
    static  int RegisterForMessage(int type, int objected,
                                   Callback& cBack);
    static void UnRegisterForMessage(int type, int objectID);
    static void SendMessage(Message* newMessage);

protected:
    MessagePump();
    MessagePump& operator= (const MessagePump&){}

private:
    static MessageTypeMap m_messageTypes;
    static MessageList    m_messageQueue;
};

#define g_MessagePump MessagePump::Instance()
```

The implementation listing shows the important functions for the `pump` class.

The `Update()` method checks each message in the queue, and either decrements its timer if it is a delayed message, or delivers the message to anyone that has registered for that message by supplying a callback function. The function then removes all the delivered messages from the queue.

`AddMessageToSystem()` is used to insert message types into the pump's list of possible messages. This can be performed at any time, whether it is class creation or as you get new objects that require the system to store information on new message types.

`RegisterForMessage()` requires two things: that the message type be in the system, and that you aren't already registered for the message. If both of these things are true, it will add you to the notification list for the specific message type that you passed in.

`UnRegisterForMessage()` does just that. It cycles through all the registrations for a specific message and removes the player from the list.

LISTING 17.3 MessagePump Implementation

```
//--------------------
void MessagePump::Update(float dt)
{
```

TeamLRN

```
if(m_messageQueue.size() == 0)
    return;

//process messages
MessageList::iterator msg;
for(msg=m_messageQueue.begin();
    msg!=m_messageQueue.end();++msg)
{
    if((*msg)->m_timer > 0)
    {
        //delayed message, decrement timer
        (*msg)->m_timer -= dt;
    }
    else
    {
        //check for registrations
        MessageTypeMap::iterator mType;
        mType = m_messageTypes.find((*msg)->m_typeID);
        if(mType == m_messageTypes.end())
            continue;

        MessageRegList::iterator msgReg;
        for(msgReg=(*mType).second->
            m_messageRegistrations.begin();
            msgReg!=(*mType).second->
            m_messageRegistrations.end();++msgReg)
        {
            //deliver message by launching callback
            if((*msgReg)->m_callBack)
                (*msgReg)->m_callBack.function
                        ((*msgReg)->m_objectID,(*msg));
            }
            (*msg)->m_delivered = true;
        }
    }

    //remove all delivered messages from queue
    MessageList::iterator end    = m_messageQueue.end();
    MessageList::iterator newEnd = std::remove_if
                (m_messageQueue.begin(),m_messageQueue.end(),
                 RemoveIfDelivered);

    if(newEnd != end)
        m_messageQueue.erase(newEnd,end);
}
```

```
//--------------------
void MessagePump::AddMessageToSystem(int type)
{
    //ensure that this type isn't already in the system
    MessageTypeMap::iterator mType;
    mType = m_messageTypes.find(type);

    if(mType == m_messageTypes.end())
    {
        MessageType *newType = new MessageType;
        newType->m_typeID = type;
        m_messageTypes[type] = newType;
    }
}

//--------------------
void MessagePump::SendMessage(Message* newMessage)
{
    m_messageQueue.push_back(newMessage);
}

//--------------------
int MessagePump::RegisterForMessage(int type, int objectID,
                                    Callback& cBack)
{
    //only register once
    MessageTypeMap::iterator mType;
    mType = m_messageTypes.find(type);

    if(mType == m_messageTypes.end())
        return REGISTER_ERROR_MESSAGE_NOT_IN_SYSTEM;

MessageRegList::iterator msgReg;
for(msgReg=(*mType).second->
    m_messageRegistrations.begin();
    msgReg!=(*mType).second->
    m_messageRegistrations.end();++msgReg)
{
    if((*msgReg)->m_objectID == objectID)
        return REGISTER_ERROR_ALREADY_REGISTERED;
}
//add new registration
MessageReg* newRegistration  = new MessageReg;
newRegistration->m_callBack  = cBack;
newRegistration->m_objectID  = objectID;
```

TeamLRN

```
        (*mType).second->m_messageRegistrations.
                            push_back(newRegistration);
        return REGISTER_MESSAGE_OK;


    }

    //--------------------
    void MessagePump::UnRegisterForMessage(int type, int objectID)
    {
        //find entry
        MessageTypeMap::iterator mType;
        mType = m_messageTypes.find(type);

        if(mType == m_messageTypes.end())
            return;

        MessageRegList::iterator msgReg;
        for(msgReg=(*mType).second->
            m_messageRegistrations.begin();
            msgReg!=(*mType).second->
            m_messageRegistrations.end();++msgReg)
        {
            if((*msgReg)->m_objectID == objectID)
            {
                (*mType).second->
                   m_messageRegistrations.erase(msgReg);
                delete (*msgReg);
                //you can exit out here, there is only one
                //registration allowed per message type
                return;
            }
        }
    }
```

CLIENT HANDLERS

In this implementation, the message handler functions will be written as callbacks. Callbacks are functions that, when the message is to be delivered to a particular game object, represent the code that the game object wants to run to respond to the message.

Another type of general message handling system that is often used is to simply have a `ProcessMessage()` function, which would in essence be a big switch statement with either code or function calls to answer any passed in message type. Using callbacks offers a more flexible system than this, and avoids the cumbersome, all-encompassing processing function.

C++ doesn't allow direct member functions to be used as callbacks, so we will be using the common method of having a dummy callback class (with a virtual function representing the form of the callback method), for callback objects to inherit from. These callback objects are then used in the place of traditional C-style callbacks. Listing 17.4 shows the dummy callback header, as well as an example function using the interface.

LISTING 17.4 `Callback` System, with Example

```
class Callback
{
public:
    virtual void function(int pid, Message* msg);
};

class EvadeCallback : public Callback
{
    void function(int pid, Message* msg);
};
```

EXAMPLE IMPLEMATION IN OUR AISTEROIDS TEST BED

In this section, we will rework the FSM version of our test bed to use messaging to perform all state transitions, as well as use the system to effect other game changes, such as giving orders to the ship from a state. To do this, we will need to make some changes to the original finite-state system and incorporate the new code for the transition callbacks and messaging functions.

Note that the AIsteroids test bed really isn't a game that is demanding this technique. The perception changes and, thus, state transitions happen frequently. There is very little time when the main ship is waiting for something to do. Very few objects in the game world require a clear channel of communication or interaction. Messaging in this game will actually add overhead, and probably slow down the system a bit, because it will be continually registering and unregistering for messages as it changes states. This implementation is really just to show practical application of the method, but should not be taken as a good implementation of messaging in a game environment.

TeamLRN

The MessState Class

Very little has to change to the state class itself. We no longer need the `CheckTransition()` function because this logic will no longer be in each individual state. Instead, the `Control` class will do the calculations and send off the correct messages to set off the transition callbacks.

The `Enter()` and `Exit()` methods will now be responsible for setting up the message registrations for the state, as well as any other cleanup functionality. In this way, messages are scoped to the particular state because any given state will then respond only to those messages that have meaning within the state.

The `Init()` method can be used to add any additional message types to the system that the state requires. This is useful for self-directed messages that a state might use. For example, a state called `Flee` might first notice the enemy, wait for a half second (simulating a reaction time), and then start to flee. You could set this up as two states (`Notice` and then `Flee`), or the `Flee` state itself could, upon entry, put itself in a wait mode (during which the previous behavior of the character wouldn't change) and send itself a wakeup message delayed for half a second. When this message comes back, it would change the wait status of the character, and he would then flee.

LISTING 17.5 MessState Header

```
class MessState
{
public:
    //constructor/functions
    MessState(int type = FSM_STATE_NONE,Control* parent = NULL)
                                    {m_type = type;m_parent=parent;}
    virtual void Enter()            {}
    virtual void Exit()             {}
    virtual void Update(float dt)   {}
    virtual void Init()             {}

    //data
    Control*   m_parent;
    int m_type;
};
```

The MessMachine class

The state machine itself has only one change from the finite version, in the `UpdateMachine()` function. As Listing 17.6 shows, there is a single line missing from

this method, where the current state's `CheckTransitions()` function is called. This model will not poll for transition changes. Instead, messages for state transitions will be sent from the individual states to the control class, which will respond to those messages by directly setting the `m_goalID` member of the `MessMachine`.

LISTING 17.6 `MessMachine` Update Implementation

```
void MessMachine::UpdateMachine(float dt)
{
    //don't do anything if you have no states
    if(m_states.size() == 0 )
        return;

//don't do anything if there's no current
//state, and no default state
if(!m_currentState)
    m_currentState = m_defaultState;
if(!m_currentState)
    return;

//check for transitions, and then update
int oldStateID = m_currentState->m_type;

//switch if there was a transition
if(m_goalID != oldStateID)
{
    if(TransitionState(m_goalID))
    {
        m_currentState->Exit();
        m_currentState = m_goalState;
        m_currentState->Enter();
    }
}
    m_currentState->Update(dt);

}
```

The `MessAIControl` Class

Listing 17.7 contains the header for the `MessAIControl` class, as well as the callback class that the controller will use to respond to requests to change the state of the machine. You will notice that it is very similar to the regular FSM controller. The real difference between the controllers is in their implementation; Listing 17.8 shows the relevant functions in the message controller file.

TeamLRN

LISTING 17.7 MessAIControl Header

```
class ChangeStateCallback : public Callback
{
    void function(int pid, Message* msg);
};

class MessAIControl: public AIControl
{
public:
    //constructor/functions
    MessAIControl(Ship* ship = NULL);
    void Update(float dt);
    void UpdatePerceptions(float dt);
    void Init();
    void SetMachineGoalID(int state);

    //perception data
    //(public so that states can share it)
    GameObj*    m_nearestAsteroid;
    GameObj*    m_nearestPowerup;
    float       m_nearestAsteroidDist;
    float       m_nearestPowerupDist;
    bool        m_willCollide;
    bool        m_powerupNear;
    float       m_safetyRadius;

private:
    //data
    MessMachine* m_machine;
    ChangeStateCallback m_changeStateCallback;
};
```

The implementation differences are much more noticeable. The constructor must also set up the messaging system for the game, so it has to add all the applicable message types to the pump. The constructor also registers for the change state message because the controller will now be causing the state-machine transitions by responding to message requests from the states.

The biggest change is in the `UpdatePerceptions()` method. Here you can see that some of the logic that was contained in the `CheckTransitions()` state functions has been transferred here instead. This does go against the modular organization model that the initial FSM system was written to incorporate, and this code showing up here is a sure sign that you shouldn't be using this method for your game.

Although this function is straightforward enough in this simple test application, even a moderately complex game would require a great deal more logic to generate the messages necessary to perform all the transitions. Again, this implementation is merely showing the code in use, but is not warranting this as a good game usage.

LISTING 17.8 `MessAIControl` Implementation

```
//--------------------
MessAIControl::MessAIControl(Ship* ship):
AIControl(ship)
{
    //construct the state machine and add the necessary states
    m_machine = new MessMachine(FSM_MACH_MAINSHIP,this);
    MStateApproach* approach = new MStateApproach(this);
    m_machine->AddState(approach);
    m_machine->AddState(new MStateAttack(this));
    m_machine->AddState(new MStateEvade(this));
    m_machine->AddState(new MStateGetPowerup(this));
    m_machine->AddState(new MStateIdle(this));
    m_machine->SetDefaultState(approach);

    g_MessagePump.AddMessageToSystem(MESSAGE_WILL_COLLIDE);
    g_MessagePump.AddMessageToSystem(MESSAGE_NO_ASTEROIDS);
    g_MessagePump.AddMessageToSystem(MESSAGE_NO_POWERUPS);
    g_MessagePump.AddMessageToSystem(MESSAGE_ASTEROID_NEAR);
    g_MessagePump.AddMessageToSystem(MESSAGE_ASTEROID_FAR);
    g_MessagePump.AddMessageToSystem(MESSAGE_POWERUP_NEAR);
    g_MessagePump.AddMessageToSystem(MESSAGE_POWERUP_FAR);
    g_MessagePump.AddMessageToSystem(MESSAGE_CHANGE_STATE);

    g_MessagePump.RegisterForMessage(MESSAGE_CHANGE_STATE,
                        m_ship->m_ID,m_changeStateCallback);
}

//--------------------
void ChangeStateCallback::function(int pid, Message* msg)
{
    ChangeStateMessage* csMsg = (ChangeStateMessage*)msg;
    int newState = *((int*)(csMsg->m_data));
    ((MessAIControl*)Game.m_AIControl)->
                                SetMachineGoalID(newState);
}
```

TeamLRN

```
//--------------------
void MessAIControl::SetMachineGoalID(int state)
{
    m_machine->SetGoalID(state);
}

//--------------------
void MessAIControl::Init()
{
    m_willCollide  = false;
    m_powerupNear  = false;
    m_nearestAsteroid = NULL;
    m_nearestPowerup  = NULL;
    m_safetyRadius    = 15.0f;

    m_target = new Target;
    m_target->m_size = 1;
    Game.PostGameObj(m_target);
}

//--------------------
void MessAIControl::Update(float dt)
{
    if(!m_ship)
    {
        m_machine->Reset();
        return;
    }

    UpdatePerceptions(dt);
    m_machine->UpdateMachine(dt);
}

//--------------------
void MessAIControl::UpdatePerceptions(float dt)
{
    if(m_willCollide)
        m_safetyRadius = 30.0f;
    else
        m_safetyRadius = 15.0f;

    //store closest asteroid and powerup
    m_nearestAsteroid = Game.
           GetClosestGameObj(m_ship,GameObj::OBJ_ASTEROID);
```

```
if(m_ship->GetShotLevel() < MAX_SHOT_LEVEL)
    m_nearestPowerup  = Game.
        GetClosestGameObj(m_ship,GameObj::OBJ_POWERUP);
else
    m_nearestPowerup = NULL;

//reset distance to a large bogus number
m_nearestAsteroidDist = 100000.0f;
m_nearestPowerupDist  = 100000.0f;

//asteroid collision determination
m_willCollide = false;
if(m_nearestAsteroid)
{
    float speed = m_ship->m_velocity.Norm();
    m_nearestAsteroidDist = m_nearestAsteroid->
                m_position.Distance(m_ship->m_position);
    float dotVel;
    Point3f normDelta = m_nearestAsteroid->m_position -
                m_ship->m_position;
    normDelta.Normalize();
    float astSpeed = m_nearestAsteroid->
                                        m_velocity.Norm();
    if(speed > astSpeed)
        dotVel  = DOT(m_ship->UnitVectorVelocity(),
                        normDelta);
    else
    {
        speed = astSpeed;
        dotVel = DOT(m_nearestAsteroid->
                        UnitVectorVelocity(),-normDelta);
    }
    float spdAdj = LERP(speed / AI_MAX_SPEED_TRY, 0.0f,
                        50.0f)*dotVel;
    float adjSafetyRadius = m_safetyRadius + spdAdj +
                            m_nearestAsteroid->m_size;

    //if you're too close, and I'm heading
    //somewhat towards you, flag a collision
    if(m_nearestAsteroidDist <= adjSafetyRadius &&
       dotVel > 0)
    {
        m_willCollide = true;
        Message* msg = new Message(MESSAGE_WILL_COLLIDE);
        g_MessagePump.SendMessage(msg);
```

TeamLRN

```
            }
            else
            {
                Message* msg = new Message(MESSAGE_WONT_COLLIDE);
                g_MessagePump.SendMessage(msg);
            }

        }
        else
        {
            Message* msg = new Message(MESSAGE_NO_ASTEROIDS);
            g_MessagePump.SendMessage(msg);
        }

        //powerup near determination
        m_powerupNear = false;
        if(m_nearestPowerup)
        {
            m_nearestPowerupDist = m_nearestPowerup->m_position.
                                   Distance(m_ship->m_position);
            if(m_nearestPowerupDist <= POWERUP_SCAN_DIST)
                m_powerupNear = true;
        }
        else
        {
            Message* msg = new Message(MESSAGE_NO_POWERUPS);
            g_MessagePump.SendMessage(msg);
        }

        //arbitrate asteroid/powerup near messages
        if(m_powerupNear && m_nearestAsteroidDist >
                            m_nearestPowerupDist)
        {
            Message* msg = new Message(MESSAGE_POWERUP_NEAR);
            g_MessagePump.SendMessage(msg);
        }
        else if(m_nearestAsteroid)
        {
            if(m_nearestAsteroidDist > APPROACH_DIST)
            {
                Message* msg = new Message(MESSAGE_ASTEROID_FAR);
                g_MessagePump.SendMessage(msg);
            }
            else
```

```
            {
                Message* msg =new Message(MESSAGE_ASTEROID_NEAR);
                g_MessagePump.SendMessage(msg);
            }
        }
    }
```

CODING THE STATES

The individual states themselves are barely changed. This book will show one of them, the idle state, to illustrate the differences. Listing 17.9 is the `MStateIdle` header (including all the required callback object declarations), and Listing 17.10 shows the implementation differences from the regular FSM method.

LISTING 17.9 `MStateIdle` Header

```
//callbacks for handling messages
class EvadeCallback : public Callback
{
    void function(int pid, Message* msg);
};
class ApproachCallback : public Callback
{
    void function(int pid, Message* msg);
};
class AttackCallback : public Callback
{
    void function(int pid, Message* msg);
};
class GetPowerupCallback : public Callback
{
    void function(int pid, Message* msg);
};

class MStateIdle : public MessState
{
public:
    //constructor/functions
    MStateIdle(Control* control):
                MessState(FSM_STATE_IDLE,control){}
    void Enter();
    void Exit();
    void Update(float dt);
```

TeamLRN

```
    EvadeCallback       m_evadeCallback;
    ApproachCallback    m_approachCallback;
    AttackCallback      m_attackCallback;
    GetPowerupCallback m_getPowerupCallback;

};
```

LISTING 17.10 MStateIdle Implementation Differences Beyond the Normal FSM Version

```
//--------------------
void MStateIdle::Enter()
{
    g_MessagePump.RegisterForMessage(MESSAGE_WILL_COLLIDE,
                                     m_parentID,m_evadeCallback);
    g_MessagePump.RegisterForMessage(MESSAGE_ASTEROID_FAR,
                                    m_parentID,m_approachCallback);
    g_MessagePump.RegisterForMessage(MESSAGE_ASTEROID_NEAR,
                                    m_parentID,m_attackCallback);
    g_MessagePump.RegisterForMessage(MESSAGE_POWERUP_NEAR,
                                m_parentID,m_getPowerupCallback);
}

//--------------------
void MStateIdle::Exit()
{
    g_MessagePump.UnRegisterForMessage(MESSAGE_WILL_COLLIDE,
                                       m_parentID);
    g_MessagePump.UnRegisterForMessage(MESSAGE_ASTEROID_FAR,
                                       m_parentID);
    g_MessagePump.UnRegisterForMessage(MESSAGE_ASTEROID_NEAR,
                                       m_parentID);
    g_MessagePump.UnRegisterForMessage(MESSAGE_POWERUP_NEAR,
                                       m_parentID);
}

//--------------------
void EvadeCallback::function(int pid, Message* msg)
{
    DataMessage<int>* newMsg = new DataMessage<int>
              (MESSAGE_CHANGE_STATE,MFSM_STATE_EVADE);
    newMsg->m_fromID  = pid;
    g_MessagePump.SendMessage(newMsg);
}
```

```
//--------------------
void ApproachCallback::function(int pid,Message* msg)
{
    DataMessage<int>* newMsg = new DataMessage<int>
                    (MESSAGE_CHANGE_STATE, FSM_STATE_APPROACH);
    newMsg->m_fromID  = pid;
    g_MessagePump.SendMessage(newMsg);
}

//--------------------
void AttackCallback::function(int pid,Message* msg)
{
    DataMessage<int>* newMsg = new DataMessage<int>
                    (MESSAGE_CHANGE_STATE, FSM_STATE_ATTACK);
    newMsg->m_fromID  = pid;
    g_MessagePump.SendMessage(newMsg);
}

//--------------------
void GetPowerupCallback::function(int pid,Message* msg)
{
    DataMessage<int>* newMsg = new DataMessage<int>
                    (MESSAGE_CHANGE_STATE, FSM_STATE_GETPOWERUP);
    newMsg->m_fromID  = pid;
    g_MessagePump.SendMessage(newMsg);
}
```

In Listing 17.10, you can also see the use of the `DataMessage` template, for sending messages with additional data. The `DataMessage` contains the data member `m_dataStorage`, the type of which is passed into the template on instantiation. When this message is delivered, the receiving callback function can cast the incoming message pointer to the correct type of `DataMessage` structure, and gain access to the data. More complex data would use the same system. If a message had to pass multiple data fields through, the message would only have to be created with a `struct` containing the necessary data fields. At send time, the `struct's` fields would be initialized to the relevant values, and the receiver would again just cast the incoming message pointer to the type of the `DataMessage` used.

PERFORMANCE OF THE AI WITH THIS SYSTEM

The game performance of this system is virtually identical to that of the regular FSM implementation. Nothing has changed other than the method by which transitions occur. As previously discussed, however, the actual CPU performance has

TeamLRN

most likely gone down a bit because we are using the messaging system somewhat wastefully; the states are registering and unregistering themselves from specific messages whenever they enter or exit.

One thing that is different from the regular FSM implementation is that this version has much more difficulty evading asteroids. This shows one of the drawbacks of making a state machine use messages to drive the state changes, that of transition priority. The state changes are currently being triggered by callbacks that have been registered for message events, so we no longer have control of the priority of multiple callbacks that could occur within a single game loop. The system simply changes state on the first incoming message. In fact, the system will change state for each incoming message (because the message pump updates before the state machine does), and so the only state change that matters ends up being the *last* state change message in the queue. This *could* be fixed by simply arbitrating the priorities at the perception level, but then you lose the decoupling of logic that the messaging system is giving you in the first place. You end up only sending out the highest priority messages, and essentially the system becomes a very convoluted way of calling a member function. A better way could be an incoming queue of state change requests to the machine, which could then determine priorities. This method isn't very clean either; however, because you will be centralizing logic, that should be at the state level. The real solution for this problem would be to implement a change state queue, as mentioned before, but also assign priority numbers on the messages themselves, and use these numbers for arbitration. Messaging priority will be discussed later in this chapter in the "Extensions" section.

Pros of Messaging Systems

Messaging systems, in effect, optimize the *client-side* code. In our game, this means that the states, which represent the client side, are optimized in that they do not have to worry about transition determinations. The *server side* (the controller class in our game) has to perform the necessary calculations, and the states merely wait until they get the correct signal from the server. We're really not saving anything (the code is moving from the states to the controller), so it might not seem like anything special.

But, the limited test bed implementation is only the beginning of what could be done with the messaging system. We could also have run the collision system with messages, informing game objects that they've collided. Powerups could send a message to the ship when collected, to inform the ship of the results, instead of the switch statement that the ship class has for getting powerups. Individual asteroids could check for proximity to the main ship and could send out messages when within some boundary or when a collision is imminent, which would speed up the

colliding determination in the controller class' `UpdatePerception` method. When all these examples are incorporated, as well as the many other ways the game could take advantage of a general messaging system, then the real power of the technique comes into play. The decoupling of class level communication frees the programmer to use the messaging system to do things that would have required full-class access between game areas before.

Most of the system is event driven, so debugging the system can be aided by logging the message stream. If the system is written for it, the message stream could even be recorded and played back through the system to reproduce bugs or specific game behavior directly.

By using the messaging system as the primary means of communication with other game sections, a class can be written in a completely modular way. This will speed up compile time for the class (because it doesn't need to include files from all over the place to gain access to other classes) and make it easier for internal methods to change over time because the only interface to the outside is through the messages passed in and out. How the messages are determined and sent from each class doesn't matter anymore, it is only important that they are sent.

Cons of Messaging Systems

Messaging systems have very few detracting points because their simplicity and commonsense understandability make them easy to work with, and their very nature (distributed computation with notification instead of centralized, polled computation) makes them less CPU intensive. Event-driven architectures usually have an additional memory footprint, caused by having to keep track of messages (and any attached data). But with simple messaging systems (such as the one implemented in this chapter), there are no insurmountable problems. Even with systems that require huge numbers of incoming and delayed messages and large amounts of passed data, load balancing techniques can be applied to the message pump, which could smooth out processing of the system, as well as address memory concerns. But even without that level of complexity, a simple messaging system can provide a game engine with most advantages of messaging.

There is the notion of too much of a good thing, however. Systems that try to incorporate messaging completely might suffer from specialization. If your game framework is completely event driven, certain parts of your game might find it hard to perform the polling functionality that it requires, or the overhead involved in polling within the messaging system might become cumbersome if the number of polled objects is large. Monitoring CPU usage to notice this would be important, and you (and your game engine) should be flexible enough to allow a module that uses a more optimized approach to getting the data it requires.

TeamLRN

Another downside is the problem discussed earlier in the chapter dealing with performance of this system in the AIsteroids game, that of message priority. Without extending the system by giving messages a level of priority, you lose a lot of control of incoming messages that is implicit in a normal state-based system.

EXTENSIONS TO THE PARADIGM

The messaging system presented in this chapter is very straightforward and simple. It is easy to understand, use, and add to the system. Some things that would advance the system include message priority, message arbitration, and automatic and extended message types.

Message Priority

By attaching a priority to messages, the message pump could sort the messages so that the more important messages get processed first. This is especially useful in games where the message system is given a finite amount of time and has to allocate its time in the most urgent direction. This also brings up the problem, however, of message starvation. Care must be taken that low-priority messages eventually build in priority (the longer they are in the queue) so that they don't sit at the bottom of the queue forever, continually pushed back by a steady stream of higher priority messages. This is OK if you have ambient messages that do not require attention, but are merely fallback status events. But in an online game that is using the messaging system to keep up to date across a network, message starvation might lead to a complete loss of game synchronization.

Message Arbitration

Another common process that is employed is that of arbitration. This is a sort of bookkeeping procedure that is performed by reading through the list of incoming messages, before processing, and looking for things such as message redundancy, message collision (two messages effectively canceling each other out), starving messages, or any other problems that the system sees a need to fix on the fly. The arbiter would then deal with each problem according to some built-in rules. Message collisions might result in both messages being removed from the queue. If the two messages don't *completely* cancel each other out, they might be removed and require a small cleanup code segment to be run. Message redundancy might simply involve removal of the excess messages, unless there is some meaning to the redundancy, such as additional messages (even though the same type of message) having slightly different and more recent data elements. In this case, they should all be thrown out except for the last one. This kind of message preprocessing can be

taken quite far, with many levels of optimization and clean up. However, an arbitration system can also become quite complex (if allowed to be), so care must be taken that a nice clean message-based system doesn't become convoluted because of a messy arbitration phase.

Automatic and Extended Message Types

Message types can streamline the messaging process and make frequent or automated tasks more convenient. Other message types might include special case messages that are handled differently by the system. Several common types of messages that are used include the following:

- *Periodic messages.* These are messages that happen on a constant time schedule, such as every second or twice a minute. Some games that are heavily event driven use an update message that happens automatically every game tick and is what signals all the objects in the game to call their `Update()`method.
- *Debugging messages.* Messages can be embedded into the system that will be perceived by a central debugging system (a very easy system would emulate asserts, but instead of stopping program execution would merely trap the program in a central place for observation or ignoring the asserted code). The debugging system could discern these *hidden* messages (in that the rest of the game doesn't know or care that they exist), as well as normal game variables to provide the programmer with whatever he needs to fix or tune code segments. These messages can be written so that they are compiled by setting a define, so that the final code isn't slowed down by debugging information and processes.
- *Confirmation messages.* Some systems might require *stalling* messages; when game object A sends B a message, A then waits (or stalls) for a response from B before carrying on. Confirmation could be sent automatically to the sender upon receipt of a message, and this behavior could thus be accommodated.
- *Immediate messages.* In the current implementation, messages are stored in a queue and are processed all at once every game loop. *Immediate* messages would skip this convention, and be sent immediately to the other game object for processing. This is necessary when you have overriding messages that require immediate attention, or have some game states that are more important than others but don't have a priority system built into your messaging structure.

OPTIMIZATIONS

Event-driven AI is fairly optimized (compared with the more traditional polling models), but only for systems that contain a more event-friendly environment. Otherwise, messaging could add overhead and complexity.

TeamLRN

As we saw in our AIsteroids test bed, some games don't lend themselves well to a messaging paradigm. Although the implementation of messaging into the test bed (that of having the transitions be event based) may be a questionable example, other parts of the game would benefit from messaging, and later game extensions could then take advantage of an already instantiated event system.

DESIGN CONSIDERATIONS

Messaging systems can find their way into almost any game, given that they are a very straightforward and efficient means of communicating between game objects and code sections. They allow fast, unconnected systems that have a centralized means for passing data and events back and forth without complicated, heavily coupled class structures

Types of Solutions

Messaging usually helps more with higher order types of solutions, meaning strategic versus tactical. This is because strategic thinking is more about coordinating multiple elements (either separate game entities, or even different aspects of a singular game entity) toward a greater goal, which implies a large degree of communication between disparate game elements. Tactical solution types, on the other hand, have more to do with physical actions in response to a given order, or to determine the best way to go about a simple task. Even at the tactical level, however, messaging provides a good means for tactical feedback to the strategic systems for increased strategic response. So, messaging works from the top down (by providing an efficient means for distributing strategic plan information to many elements), but also from the bottom up (by providing the means for many separate game elements to provide feedback into the system).

Agent Reactivity

Event-driven agents are all about reactivity because they are simply waiting for something to happen so they can respond. With the proper perception system, any level of reactivity can be implemented with messaging-based agent communication.

System Realism

Event-driven games need to be watchful that their possible event list isn't too narrow, otherwise the AI-controlled characters will be too predictable and static, behaviorally. This is not to say that event-driven systems themselves are predictable. But some games drive the actual behavior of the characters almost com-

pletely from perception events, and this can dull the richness of the character dramatically. Usually, event-driven characters also need to use other techniques, such as scripting, to respond to key game events with proprietary, rich behavior. This is only one example of how messaging could be combined with another AI technique to provide additional realism or depth of play.

Genre and Platform

Genre and platform are of almost no concern to messaging methods, which excels, however, with games that have either numerous game objects that require communication (such as RTS games with huge armies that are being ordered around and are sending back constant feedback), or very rich interaction between game objects (such as a football game, where two or more players are responding to each other's positions relative to each other, are engaged in complex collision with each other, and are dealing with other factors during a play). Messaging systems do require additional memory, but usually make up for it by simplifying the code base and can be implemented on even the smaller, more restrictive platforms.

Development Limitations

Development limitations might actually push a team *toward* a message-based system, which provides amazing bang for the buck, and can speed up implementation of game features that require cooperation between classes and game objects. Message-based systems (if done properly) are generally quicker to compile and build, because of the class decoupling that the method enforces. Debugging systems can be integrated directly into the messaging stream (or at the object level) and provide many access points for fixing potential problems and logging behaviors for review.

Entertainment Limitations

Tuning difficulty settings, balancing specific behaviors, and other entertainment concerns are generally independent of messaging system use, so are not usually a problem.

SUMMARY

Messaging systems can provide a variety of decision-making paradigms with additional flexibility in communication between game objects, as well as between separate code segments that must act in concert.

TeamLRN

- AI characters are usually reactive, which lends itself well to an event-driven methodology.
- AI systems are usually high level, meaning that they have access to many separate parts of the game engine, so they can perform the kinds of determinations and behaviors necessary. Messaging provides a clean interface for this access to occur without giving global class access.
- The simple messaging system in this chapter was implemented in three parts: the message pump, the message object, and client handlers.
- The client handlers in this method will be coded as callback objects to facilitate greater flexibility and organization over more centralized methods, such as a general `ProcessMessage()` function.
- The test bed will be implemented so that all transitions occur because of events. Although this is not the ideal messaging example, it does show off the power of the technique and provides a clear example of how messaging can be used.
- Message-based systems work best when most of the game takes advantage of the system. Debugging and other secondary systems can also use the messaging engine to ease those tasks as well.
- Messaging does require additional memory to store messages and attached data, but load balancing and arbitration systems can optimize the level of CPU and memory necessary.
- Adding the ability to set message priority, allow message arbitration, and add automatic or extended message types to the system can extend the use of messaging beyond its most simple implementation.

18 Scripting Systems

So far, we have dealt with ways of constructing game logic and behavior in code. This results in custom AI systems that are tied closely to the product and can be optimized to perform well. But, as we have seen, this requires dedicated programmers to implement, debug, and extend the system. Depending on the game, however, programmers may not be the people that are creatively overseeing the product. There might be considerable roadblocks to communicating the creative vision to the available programmers. Even more common, the level of creative content that must be included in the game is simply too great to allow a limited number of programmers to custom code everything at the quality level that the game requires.

SCRIPTING OVERVIEW

A common technique for getting more hands into the guts of the AI system, without having to hire more programmers, is called *scripting*. Scripting means using a simplified programming language (although scripting tools can be made visual as well as written) to build AI elements, logic, or behaviors. The conversation tree in a role-playing game, the cinematic movement and visuals of various characters during story sequences, the specifics of each move in a complex fighting game, or the way that groups of enemies coordinate attacks can and have all been performed using scripting systems in various games.

Scripting languages range from the very simple (sometimes called predicate languages because they usually comprise general "rules" with a keyword, a value, and possibly some kind of expression), to being full-fledged programming languages (*Neverwinter Nights* uses Java as its scripting language).

Making a scripting language for your product is not a trivial task. You are essentially creating another product, with a specific user (in this case game designers or end users who are creating mods for your game), input and output requirements, and a design, implementation, and debug schedule separate from the actual

TeamLRN

game you intend on creating. The design phase of your scripting system requires careful thought to many different technical and creative elements:

- *The kinds of functionality that the language will require.* Does your game need linear triggering of events, or does the script need to include conditional branches? Will the scripters need variables? Essentially you are defining the complexity of the scripting system. Let us consider an example, that of a vertical shooter. In this game, you will fly a ship along a set length of game world, with attacking enemies flying in patterns along the way. A simple scripting system for this game could be one that merely allows the scripter to define the points at which specific enemy types spawned and could even modify starting parameters for each enemy as the scripter saw fit. The specifics of the enemy behavior would be left in code. A more complex system might allow the scripter to define enemy movement patterns within script and allow assignment of these patterns to enemies. An even more involved system might *completely* drive the creation of enemies: a scripter would define each as a set of attributes (body type, speed, duration, armor level, etc.) and behavior types (attack method, movement style and pattern, etc.) from a laundry list of possibilities that the game engine is capable of. Finally, a system could be implemented that would also allow enemies to assess game side values and respond to them within scripted sequences. This would allow the scripters to write reactive systems that could account for the state of the player or other enemies in the game.
- Will the runtime engine interpret the scripts, or will they be precompiled into some kind of bytecode? Interpreted scripts tend to be slower and larger. But they also allow more flexibility (you could enter a new script from an in-game console, for example, and it could then be reinterpreted by the engine on the fly), and are simpler to implement because of the lack of the intermediary compilation step. This determination should be made if your game might not have the CPU power in excess that an interpretive system requires.
- How big will the scripts be in size? Code tends to compile down into small chunks, whereas script code (especially interpreted script) does not. Care must be taken that the platform you are developing for has enough memory (or memory bandwidth, if you will be streaming scripts into the game from disk) to accommodate the sum of the script requirements for the game.
- Who will be using the scripting language? If junior programmers will be using the language, it can be made fairly complex and full featured. Or are the primary users high-level designers who don't have much technical knowledge? For the severely nontechnical, the scripting system needs to be simple enough to use, robust enough to handle errors without crashing, and include a decent amount of debugging hooks to be useful. A very common mistake in creating

a scripting system is to make something that is far too complex and powerful for nontechnical designers. They usually become overwhelmed by the system and end up not using it to its full potential. A better way is to make the initial system easy to use, then gradually add more advanced features as the designers become comfortable with the system. If you are going to expose this scripting language (and its tools, if needed) to the public to make modding your game easier, realize that the average person who might try and use it may have little, if any, programming experience. The learning curve can be simplified if your scripting language follows similar rules to an already established language (the primary reason for the large number of "C-like" scripting languages), or is so simple that a few example scripts can be included with the game to show how to use all the functionality.

EXAMPLE IMPLEMENTATION IN OUR AISTEROIDS TEST BED

This chapter will actually cover two different ways of scripting within a game. First, we will implement a very simple configuration scripting system that will allow you to bind in-game variables or actions to scripting tokens. Second, we will cover a more general and full-featured method, by discussing the Lua language and how it can be embedded within a game application as well as exposing game functions and variables to Lua.

A Configuration Script System

The first scripting system we will cover is very basic and will allow the programmer to set up a very simple grammar using keywords and parameters. There will be no variables, scoping, parameter passing, or any other of the more advanced language constructs, but it will still allow for barebones "languages" that can be used to set up transition rules, triggers, or simple behaviors. Think of this system as a formalized way of configuring or initializing variables from within a script. You expose whatever flags, values, and triggers you want to the scripting language, and then a script can be written that uses the correct tokens to perform the setup tasks that you require.

The implementation we will use for the AIsteroids game is split into four parts: a parser, a list of tokens, in-game token callbacks, and the actual script files.

- The parser is the code that loads in the script file, scans it for applicable tokens, and then executes any tokens it finds.
- Tokens comprise a token name (which is also what is used by the scripter to invoke the token), and an `Execute()` function, which is called by the parser for

TeamLRN

each token it finds. Executing a token involves scanning the file for any additional parameters that the token expects, and then sending the game a message that includes this data.

- The in-game callbacks are the functions that will respond to the messages sent by the token `Execute()` calls. This is where you will actually bind real game variables to incoming script data.
- The scripts themselves can be written in any text editor. The only real grammar that the system uses is that each line must end in a semicolon. The example scripts included in this implementation (see Listing 18.1 for the file test.txt) use the general grammar "`Token= parameter;`" but the = sign is actually part of the token name, so it could be anything you want. The "comment" lines at the top of the file do not require the comment signifier //, they are just there for readability. In fact, if a real token had been included in the comment line, it would have been found and parsed.

LISTING 18.1 `test.txt` Script File

```
//don't need to put anything in a certain order,
//the parser ignores whitespace after the = sign,
//all lines begin with a token and end in a semicolon,
//and tokens are case insensitive
//all values in the script Override the default values
//that are set up in the Init() functions.

PowerupScanDist= 150.0;
SeekPowerups= true;
MaxSpeed= 80.0;
ApproachDist= 180.0;
AttackDist= 260.0;
SafeRadius= 15.0;
```

To initialize the system, you register the parser with all the `Tokens` (Listing 18.2 is the header) you intend on using in your script. As the code shows, each token stores an ID, an internal value `m_matchPos` (used by the parser when scanning for tokens), and the name string that is used to identify the token in a script file. The Get functions are all standard retrieval methods to extract the various kinds of parameter types from a text file. If you had any other widely used parameter types (be they different data types, or more complex data structures) you could add methods for loading them here. The `enum` stores all the ID types for the tokens used in the game.

LISTING 18.2 Token Header Information

```
class Token
{
public:

    enum
    {
        TT_NONE,
        TT_POWERUPDIST,
        TT_POWERUPSEEK,
        TT_APPROACHDIST,
        TT_ATTACKDIST,
        TT_MAXSPEED,
        TT_SAFERADIUS,
};

//constructor/functions
Token(int type = TT_NONE, char* name = "")
     {m_tokenAsStr=new char(MIN(strlen(name),MAX_TOKEN_LENGTH));
      m_tokenAsStr = name;m_tokenID = type;m_matchPos = 0;}
~Token(){}
virtual void Execute(_iobuf* fileName) {}

//Additional data acquisition
float GetFloat(_iobuf* fileName);
char  GetChar(_iobuf* fileName);
int   GetInt(_iobuf* fileName);
bool  GetBool(_iobuf* fileName);
void  GetString(_iobuf* fileName, string& storageStr);

    //data
    int             m_tokenID;
    int             m_matchPos;
    char*           m_tokenAsStr;
};
```

The parser (Listing 18.3 is the header, 18.4 is the important function implementations) is very straightforward. In fact, it's about as simple as a file parser can get. As the header shows, it only stores one data structure: a pointer to the list of tokens, passed in when you instantiate the parser. This facilitates using the parser in a general sense for the whole game; you can change the token list and reparse a file, or parse a different file. You might have different token lists for different states

TeamLRN

of the game, or different levels, or for various unconnected places within the game that you might use this system.

LISTING 18.3 `Parser` Header Information

```
class Parser
{
public:
    Parser(TokenList *tList = NULL):m_tokenList(tList){}
    int         CheckForToken(char currentChar);
    bool        ParseFile(char* fileNameStr);
    void        Reset();
    void        SetTokenList(TokenList *tList){m_tokenList = tList;}

protected:
    TokenList*  m_tokenList;
};
```

The parser works on a "single character at a time" basis. It gets a character out of the file, and then checks that against the list of tokens. If it finds a token with a matching character, it adds to its `m_matchPos` variable. Any time it finds a token that has matched all its characters to the incoming stream it flags it as a token, resets everybody else, and returns. If it is in the middle of a token string, and an incoming character doesn't match, it resets the count because the matches need to be in order. Two things to notice:

- Tokens are scanned for using case insensitivity. This will save you the headache of looking for bugs based on this problem, especially if nontechnical people are going to be the primary users of the system. Notice too that the `GetBool()` method in `Token.cpp` will accept a variety of symbols as true or false so that it makes it easier for nonprogrammers to use.
- Tokens whose names are subsets of other tokens' names might be a problem. If you had two tokens, "Shout=" and "Out=", you will have collision problems because "out=" is in both strings. So, based on the order in which they were registered, only one token will ever be called. You could either extend the system to allow for this (by flagging tokens for execution and then batch executing them after scanning, or whatever means you see fit), or by simply keeping the token names from colliding with each other (you can enforce the convention verbally to the programmers, or make a `RegisterToken()` function (which you would pass a `TokenList` pointer because the parser class itself doesn't store the tokens), and have it flag incoming name collisions as an error.

LISTING 18.4 Parser Implementation

```
//--------------------
int Parser::CheckForToken(char currentChar)
{
    TokenList::iterator tListiterator;
    for (tListiterator = m_tokenList.begin();
         tListiterator != m_tokenList.end(); ++tListiterator)
    {
        Token* pToken = *tListiterator;
        if (tolower(currentChar) ==
            tolower(pToken->m_tokenAsStr[pToken->m_matchPos]))
        {
        // if the currentChar matches the requested
        // character of the current token,...
            // increase the "match-position" counter
        pToken->m_matchPos++;

        if (pToken->m_matchPos == strlen(pToken->m_tokenAsStr))
        {
            // if the counter equals the length of the current
            // token, we found a token. Thus,...
            // ...reset the counters of all the
                // other tokens and...
            Reset();
            // ...return the token found
            return pToken->m_tokenID;
        }
}
else
{
            // if the currentChar does *not* match the requested
            // character of the current token,...
            // reset the corresponding counter
            pToken->m_matchPos = 0;
        }
    }
    return NO_TOKEN;
}

//--------------------
void ZeroPosition(Token* pToken)
{
    pToken->m_matchPos = 0;
}
```

TeamLRN

```
//--------------------
void Parser::Reset()
{
    for_each(m_tokenList.begin(),m_tokenList.end(),ZeroPosition);
}

//--------------------
bool Parser::ParseFile(char* fileNameStr)
{
    FILE* pFile;
    if ((pFile = fopen(fileNameStr,"r")) == NULL)
    {
        return false;
    }

    char buffer;
    Reset();
    while (fread(&buffer, 1, 1, pFile) == 1)
    {
        int currentToken = CheckForToken(buffer);
        if(currentToken == Token::TT_NONE)
            continue;
        else
        {
            TokenList::iterator tListiterator;
            for (tListiterator = m_tokenList.begin();
                 tListiterator!=m_tokenList.end();++tListiterator)
            {
                Token* pToken = *tListiterator;
                if(pToken->m_tokenID == currentToken)
                    pToken->Execute(pFile);
            }
        }
    }

    fclose(pFile);
    return true;
}
```

Once a token is found, its `Execute()` call is performed (Listing 18.5 shows the execute method for a couple of the example tokens). This method is responsible for retrieving any additional parameters from the file, then setting up and sending the correct message to the engine. The method allows for any implementation you

want, so this could be used for any kind of structure you want. The single parameter shown in the examples could be extended to multiple parameters separated by commas, or a token could be a kind of state indicator. An "if" token's execute method could be an entirely new parse phase that looked for any number of additional conditions It could have its own list of conditional tokens and call the parser on the current file with this new token list. A "then" token would then mark the latter phase of this token; it would stop looking for conditions and instead start scanning for actions to perform. This style of complex, nested token might actually read in many lines of the script, all triggered by a single token (if). Again, because of the generic structure of this parsing system, you can implement almost anything you require for simple parsing situations.

LISTING 18.5 Some Execute Method Implementations

```
//--------------------
void TokenSafeRadius::Execute(_iobuf* fileName)
{
    float safeRad = GetFloat(fileName);

    //send out message with data of incoming token
    DataMessage<float>* newMsg = new DataMessage<float>
                    (MESSAGE_TOKEN_SAFERAD,safeRad);
    g_MessagePump.SendMessage(newMsg);

}

//--------------------
void TokenPowerupSeek::Execute(_iobuf* fileName)
{
    bool seekPowerups = GetBool(fileName);

    //send out message with data of incoming token
    DataMessage<bool>* newMsg = new DataMessage<bool>
                    (MESSAGE_TOKEN_POWSEEK,seekPowerups);
    g_MessagePump.SendMessage(newMsg);

}
```

PERFORMANCE OF THE AI WITH THIS SYSTEM

This simple scripting system is built directly on top of the messaging-based system and does nothing except set up initial variables, so it performs exactly like the

TeamLRN

purely message-based method. The small file that is parsed at load time adds negligible time to level start up, and even a rather large configuration file with a decent number of possible tokens would still not be much, if anything, of a concern to the game's performance.

Extensions to the Configuration Script Paradigm

This solution is so open and generic that you could technically code whatever kind of advanced token type you needed for your game. You could construct tokens that require any number of parameters, or use the "//" token to actually signify a comment, and discount parsing the rest of the entire line. A more complex addition would be tokens that effectively put the parser into a special mode, so that it's then looking for different tokens. This would be roughly equivalent to a *block* signifier within a regular programming language. For example, when the C compiler sees an `if` token in the code, it then looks for a left parenthesis (signifying the start of an expression block), or else there's a syntax error. After the expression, the C compiler then looks for a single statement, which may be a curly brace, signifying another block. Your language could technically build any type of block structure that it wanted to give the script advanced organization and structure. Of course, with the simplistic and rough parsing being done in this system, you would probably have a bit of a hurdle dealing with the possible errors that scripters might introduce trying to follow anything but a very basic grammar within this code.

EMBEDDING LUA

In this section, we will embed Lua in a game environment as a substitute for creating our own scripting language. We will cover a very light review of the Lua language, and then move on to the details of integrating the language into your game and passing information back and forth between Lua and a C/C++ program environment.

Lua Overview

Lua is a lightweight programming language developed by a team at the Computer Graphics Technology Group at the Pontifical Catholic University in Brazil. In addition to being used as a stand-alone language, many games are using Lua as a general-purpose scripting and extension system for their games. You might decide to go in this direction for a number of reasons. You might not have the time to write, debug, and maintain your own custom system. You might have many areas in your game that you would like to use scripting with to allow maximum extensibility, and thus, want a very general scripting system to encompass the many areas of your game code you want to affect. You might even have a number of people on

your staff that already know the Lua language because it has been around for a while, and has been used in a number of well-known commercial games (*Baldur's Gate* and *Grim Fandango* are two examples).

Lua is slowly but surely beating out older embedded languages like Python for many reasons:

- It is generally faster, has a smaller memory footprint, and is easier to learn.
- Its syntax is largely procedural and has dynamic typing.
- It can be interpreted from script or compiled bytecode.
- It has automatic memory management with garbage collection facilities.
- It is easy for both programmers and nontechnical people to learn, with a sort of free-form Pascal syntax (at the 2003 Game Developer's Conference, it was discussed that seasoned programmers should pick it up in an hour or so, and nontechnical people would need just a bit longer for simple Lua tasks).

But the biggest reason that Lua is gaining steam is its abilities in the area of integration with other languages. Instead of implementing numerous internal language features, Lua includes an easy-to-use API for exchanging data back and forth between your game and Lua. In this way, Lua can be thought of as a tool for building game-specific languages. You are effectively building a set of functions (in Lua and your code) that allow designers, or whoever is using the scripting system, to write game specific scripts to perform actions within your game. Coupled with the very forgiving syntax of Lua, very usable and human-readable code can be generated quickly and easily. This also keeps the core language small and fast.

Lua Language Fundamentals

This section will give a very brief overview of the Lua language. It is not meant to be exhaustive but, rather, to simply show some of the primary language features. For a much broader look at how Lua is programmed, go to the source: *http://lua-users.org/wiki/TutorialDirectory*, which has a nice selection of topics that are covered quite well. These tutorials can be followed easily because the stand-alone interpreter can be run, and Lua commands can be entered directly into an internal prompt for immediate execution. Lua's syntax is easy to pick up if you have any experience programming more high-level languages. Listing 18.6 shows some example Lua code, which will be used for illustration. Some of the basic language features include the following:

- *Very simple scoping.* All Lua statements are in the global environment. The only way to restrict scoping is to assign a variable local status within a smaller block of code (this block being delimited by a control structure, within a function, etc.).

TeamLRN

- *Dynamic typing.* Lua does not require variable types to be declared. Lua only recognizes seven different types; those being "nil," "boolean" (*nil* counts as false, but the number zero and "" is true), "number" (all numbers in Lua are considered floats), "string," "table," "function," "thread," and "userdata" (a type specifically designed to allow for arbitrary C pointers to be stored, they are essentially void* variables). These types can be intermingled (especially if you're a diehard C++ programmer) to an alarming degree.
- *Tables.* Tables are *the* free-form data construct in Lua. Much like a list in LISP, you can put any combination of types into a table, and tables can contain other tables. Each member of a table is essentially stored like an STL `map`, in that it has a key and a value. Simple tables (like one declared `table = {1,2,3}`) are also called *numerically indexed* because all the keys are implied as array indices. A corresponding non-numerically indexed table would could be declared as `table = {name = "Bob", number = "5551212", hometown = "Somewhere"}`. In this second table, you would access members by key name, for example, `table.name == "Bob"`. Tables can also contain functions, which could be thought of as object-oriented "methods."
- *Control Structures.* Lua provides a number of standard control blocks, including `do` loops, `while..do` loops, `repeat..until`, `if..then..else..elseif` blocks, and `for` loops. Each control structure (except for `repeat..until`) must be delimited with an "`end`" identifier.
- *The Stack.* Lua uses a "stack" to pass values to and from C programs. Even though it is referred to as a *stack*, it really isn't a stack. Usually, stacks are only accessed with push and pop commands. Lua stacks are more like an indexed set of registers that are used during the communication between programs and scripts. Anytime you call a C function from Lua, a new, independent stack is created for passing data back and forth. The default size for these stacks (as defined in `lua.h` as `LUA_MINSTACK` is 20, usually more than enough unless you're pushing a bunch of things to the stack from within the function or passing huge structures), but it can be grown using the function `lua_checkstack()`. In addition to the expected push and pop functions, Lua stacks have commands for inserting, removing, replacing specific elements, as well as recognizing pseudo-indices (by using negative values, you can index relative to the top, and positive values index relative to the bottom) to make random stack access easier. The last chunk of code in Listing 18.6 shows a small block of C code that manipulates stack values for illustration.

LISTING 18.6 Simplistic Lua Syntax Demo

```
--examples declaring different types
```

```
varNumber   = 5
varFloat    = 5.5
varFunction = function(i) return i-1 end
varNumber   = varFunction(56)
varTable    = {1,false,6,8,{12,"string",7.99}}
v1,v2,v3   = 12,"apple",-5.6

--examples of control structures
index = 1
do
  index = 5
  print("Index = "..index)--should print 5
end
print("Index = "..index)--should print 1
--===========
index = 1
while index < 5 do
    print("Been here "..index.." times")
end
--===========
num = 1
repeat
    print(num)
    num = num * 3
until num > 100
--===========
function min(a,b)
    local minimum
    minimum = a
    if b < a then
            minimum = b
    end
    return minimum
end
--===========
if x == 3
    print("X equals 3")
elseif x < 1
    print("X is not 1")
else
    if x > 0
            print("X is positive, and less than 1")
    else
            print("X is negative")
    end
```

TeamLRN

```
end
-==========
for index 1,50,3 do
    print("Loop value ="..index)
end

varTable = {name="marvin",look="monkey",job="ceo"}
for key,value in varTable do
    print(key,value)
end

-examples of table usage
table = { 23,44.5,18, color="blue", name="luxor" }
print(table[1])-will print 23
print(table[color])-will print blue



--------------------

//examples of stack usage, C code
//As an example, if the stack starts as 10 20 30 40 50*
//(from bottom to top; the '*' marks the top
lua_pushnumber(L, 10);// -> 10*
lua_pushnumber(L, 20);// -> 10 20*
lua_pushnumber(L, 30);// -> 10 20 30*
lua_pushnumber(L, 40);// -> 10 20 30 40*
lua_pushnumber(L, 50);// -> 10 20 30 40 50*


lua_pushvalue(L, 3);//  -> 10 20 30 40 50 30*
lua_pushvalue(L, -1);// -> 10 20 30 40 50 30 30*
lua_remove(L, -3);//    -> 10 20 30 40 30 30*
lua_remove(L, 6);//     -> 10 20 30 40 30*
lua_insert(L, 1);//     -> 30 10 20 30 40*
lua_insert(L, -1);//    -> 30 10 20 30 40* (no effect)
lua_replace(L, 2);//    -> 30 40 20 30*
lua_settop(L, -3);//    -> 30 40*
lua_settop(L, 6);//     -> 30 40 nil nil nil nil*
```

Integration

Integrating Lua scripts into your game is simple. You link your game with the Lua libraries, instantiate an instance of the Lua interpreter, and then either perform Lua

commands directly, or by loading and parsing an entire file. Listing 18.7 shows the code necessary to start up the interpreter. The secondary `libopen` functions initialize parts of the interpreter that you might need (input/output, advanced string functions, and math functions, respectively). The `lua_settop()` call clears the stack of any random values that were left there by the library initializations. The last part of the whole process is to expose game side functions and values to Lua, and vice versa. The code in this book will use a very simple extension to Lua, the LuaPlus Call Dispatcher, written by Joshua Jensen, to help with the exposing process. This single header file is a nice compilation of templates that allow very simple registration C++ code and data elements, whether global, members of a class, or even virtual in the case of functions. The reason for using this is that normally any function exposed to Lua from C needs to be a static function of the type `static int Function (lua_state* ls)` , and all arguments and return values being passed on the stack. These are called *glue functions,* in that they provide a middle layer between your real C++ methods and Lua scripts. The LuaPlusCD merely uses some very clever template coding to provide these glue functions for us, as well as handling the stack manipulation necessary to pass the arguments and return values. Listing 18.8 shows examples of exposing variables and code from C++ to Lua, and back.

LISTING 18.7 Simple Lua Interpreter Startup Code

```
#include "luaPlusCD.h"
extern "C"
{
    #include "lua.h"
    #include "lualib.h"
}

//and this code must be in an actual function

m_luaState          = lua_open();
lua_baselibopen(m_luaState);
lua_iolibopen(m_luaState);
lua_strlibopen(m_luaState);
lua_mathlibopen(m_luaState);
lua_settop(m_luaState,0);
```

LISTING 18.8 Examples of Exposing Variables and Data to and from Lua

```
//from C++ to Lua
//--------------------
```

TeamLRN

```
//variable data
int integerVariable   = 42;
char stringVariable[] = "doughnut";

lua_pushnumber(m_luaState,integerVariable);
lua_setglobal(m_luaState,"intVar");
lua_pushstring(m_luaState,stringVariable);
lua_setglobal(m_luaState,"strVar");
////////////////////////////////////////
//static functions using barebones Lua
//function takes a number argument,
//and returns 3*the number and 4*number
static int MyCFunction(lua_state* L)
{
    int numArgs = lua_gettop(L);//should be one
    float arg[numArgs];
    int i;
    for(i=0;i< numArgs;i++)
        arg[i] = lua_isnumber(L,1);
    for(i=0;i<numArgs;i++)
    {
        lua_pushnumber(L,arg1*3.0f);
        lua_pushnumber(L,arg1*4.0f);
    }
    return 2*numArgs;//number of results
}
lua_register(m_luaState,"MyCFunction",MyCFunction);
//Lua script can then say:
// a,b = MyCFunction(25)
//with results: a==75, b==100
//...or...
// a,b,c,d = MyCFunction(4,5)
//with results: a==12,b==16,c==15,d==20
////////////////////////////////////////

//regular functor examples using LuaPlusCD
//(example taken from author's website)
static int LS_LOG(lua_State* L)
{
    printf("In static function\n");
    return 0;
}
```

```
class Logger
{
public:
    int LS_LOGMEMBER(lua_State* L)
    {
        printf("In member function. Message:%s\n",
                                            lua_tostring(L,1));
        return 0;
    }

    virtual int LS_LOGVIRTUAL(lua_State* L)
    {
        printf("In virtual member function\n");
        return 0;
    }
};

lua_pushstring(L, "LOG");
lua_pushfunctorclosure(L, LS_LOG, 0);
lua_settable(L, LUA_GLOBALSINDEX);

Logger logger;
lua_pushstring(L, "LOGMEMBER");
lua_pushfunctorclosure(L, logger, Logger::LS_LOGMEMBER, 0);
lua_settable(L, LUA_GLOBALSINDEX);

lua_pushstring(L, "LOGVIRTUAL");
lua_pushfunctorclosure(L, logger, Logger::LS_LOGVIRTUAL, 0);
lua_settable(L, LUA_GLOBALSINDEX);

//and the package can also set up direct calls, which are much
//more natural to C programmers...
void LOG(const char* message)
{
    printf("In global function: %s\n", message);
}


class Logger
{
public:
    void LOGMEMBER(const char* message)
    {
        printf("In member function: %s\n", message);
    }
```

TeamLRN

```
    virtual void LOGVIRTUAL(const char* message)
    {
        printf("In virtual member function: %s\n", message);
    }
};


lua_pushstring(L, "LOG");
lua_pushdirectclosure(L, LOG, 0);
lua_settable(L, LUA_GLOBALSINDEX);

Logger logger;
lua_pushstring(L, "LOGMEMBER");
lua_pushdirectclosure(L, logger, Logger::LOGMEMBER, 0);
lua_settable(L, LUA_GLOBALSINDEX);

lua_pushstring(L, "LOGVIRTUAL");
lua_pushdirectclosure(L, logger, Logger::LOGVIRTUAL, 0);
lua_settable(L, LUA_GLOBALSINDEX);
////////////////////////////////////////

//from Lua to C++
//--------------------

//variables
int intVar;
char strVar[20];
lua_getglobal(m_luaState,"intVarName");
intVar = lua_tonumber(lua_gettop(m_luaState));
lua_getglobal(m_luaState,"strVarName");
strVar = lua_tostring(lua_gettop(m_luaState));
////////////////////////////////////////

//functions
//Lua function looks like:
//    function multiply(x,y)
//        return x*y
//    end

//C code would require:
float x = 123.0f;
float y =  55.0f;
lua_getglobal(m_luaState,"multiply");
lua_pushnumber(m_luaState,x);
```

```
lua_pushnumber(m_luaState,y);
float result = lua_tonumber(lua_call(m_luaState,2,1),-1);
```

EXAMPLE IMPLEMENTATION IN THE AISTEROIDS TEST BED

To run Lua scripts from the test bed, we need just a few additions. We will be building on the messaging-based system from Chapter 17, "Message-Based Systems." Essentially, we are going to expose the necessary perception data members to our Lua script, which will then perform the necessary logic to determine which state the ship's state machine should be in. Listing 18.9 shows the changes to the code, and Listing 18.10 shows some small sample Lua scripts that can be used to control the AI ship for this demo. Notice that the second script only uses the `Evade` and `Approach` states. It relies on the fact that the `Evade` state shoots the guns if you line up with an asteroid. Not really a great way to play the game, this is just to show contrast between the real script and this one.

LISTING 18.9 `MessAIControl` Changes Needed to Use Lua Scripting

```
#include "luaPlusCD.h"
extern "C"
{
#include "lualib.h"
}

//--------------------
MessAIControl::MessAIControl(Ship* ship):
AIControl(ship)
{
    g_MessagePump.AddMessageToSystem(MESSAGE_SHIP_TOTAL_STOP);
    g_MessagePump.AddMessageToSystem(MESSAGE_CHANGE_STATE);

    //construct the state machine and add the necessary states
    m_machine = new MessMachine(MFSM_MACH_MAINSHIP,this);
    m_machine->AddState(new MStateApproach(this));
    m_machine->AddState(new MStateAttack(this));
    m_machine->AddState(new MStateEvade(this));
    m_machine->AddState(new MStateGetPowerup(this));
    MStateIdle* idle = new MStateIdle(this);
    m_machine->AddState(idle);
    m_machine->SetDefaultState(idle);
    m_machine->Reset();
```

TeamLRN

```
    m_messReceiver = new MessageReceiver;

    //default values
    m_safetyRadius      = SAFETYRADUIS;
    m_powerupScanDist   = POWERUP_SCAN_DIST;
    m_maxSpeed          = MAI_MAX_SPEED_TRY/Game.m_timeScale;
    m_appDist           = MAPPROACH_DIST;
    m_attDist           = MATTACK_DIST;
    m_powerupSeek       = true;

    m_luaState          = lua_open();
    lua_baselibopen(m_luaState);
    lua_settop(m_luaState,0);//clear the stack

    //bind const values to lua variables
    lua_pushnumber(m_luaState,MAX_SHOT_LEVEL);
    lua_setglobal(m_luaState,"gvMaxShotPower");
    lua_pushnumber(m_luaState,MFSM_STATE_APPROACH);
    lua_setglobal(m_luaState,"gsSTATEAPPROACH");
    lua_pushnumber(m_luaState,MFSM_STATE_ATTACK);
    lua_setglobal(m_luaState,"gsSTATEATTACK");
    lua_pushnumber(m_luaState,MFSM_STATE_EVADE);
    lua_setglobal(m_luaState,"gsSTATEEVADE");
    lua_pushnumber(m_luaState,MFSM_STATE_GETPOWERUP);
    lua_setglobal(m_luaState,"gsSTATEGETPOWERUP");
    lua_pushnumber(m_luaState,MFSM_STATE_IDLE);
    lua_setglobal(m_luaState,"gsSTATEIDLE");

    //bind state change function for lua to use
    lua_pushstring(m_luaState,"ChangeState");
    lua_pushdirectclosure(m_luaState,*this,
       &MessAIControl::SetMachineGoalState,0);
    lua_settable(m_luaState,LUA_GLOBALSINDEX);
}
//--------------------
void MessAIControl::Update(float dt)
{
    if(!m_ship)
    {
        m_machine->Reset();
        return;
    }

    UpdatePerceptions(dt);
```

```
//update exposed lua variables
lua_pushnumber(m_luaState,m_nearestPowerupDist);
lua_setglobal(m_luaState,"gvDistPowerup");

lua_pushnumber(m_luaState,m_nearestAsteroidDist);
lua_setglobal(m_luaState,"gvDistAsteroid");

lua_pushboolean(m_luaState,m_willCollide);
lua_setglobal(m_luaState,"gvWillCollide");

lua_pushboolean(m_luaState,m_isPowerup);
lua_setglobal(m_luaState,"gvIsPowerup");

    lua_pushboolean(m_luaState,m_isAsteroid);
    lua_setglobal(m_luaState,"gvIsAsteroid");

    lua_pushnumber(m_luaState,m_ship->GetShotLevel());
    lua_setglobal(m_luaState,"gvShotPower");

    //run lua script, which handles state transitions
    lua_dofile(m_luaState,"script1.lua");

    m_machine->UpdateMachine(dt);
}
```

LISTING 18.10 Sample Lua Scripts to Control the Ship

```
--Lua script for simple asteroids state Logic


if gvWillCollide then
    ChangeState(gsSTATEEVADE)
elseif gvIsPowerup  and gvShotPower < gvMaxShotPower then
    ChangeState(gsSTATEGETPOWERUP)
elseif gvIsAsteroid then
    if gvDistAsteroid < 200 then
        ChangeState(gsSTATEATTACK)
    else
        ChangeState(gsSTATEAPPROACH)
    end
else
    ChangeState(gsSTATEIDLE)
end
```

TeamLRN

```
--Another asteroids Lua script
    if gvWillCollide then
       ChangeState(gsSTATEEVADE)
    else
       ChangeState(gsSTATEAPPROACH)
    end
```

Lua script is being used to handle the state transition logic for the state machine. Because of the way the test bed is written (with the script being executed by calling the `lua_dofile()` function), you don't even have to shut off the game to change the AI behavior. If you edit the `script1.lua` file, and then save it, in the next game tick the file will be loaded and executed. You can change the script in a text editor and see the results every time you hit save. In a real game situation, however, you would probably not want frequent disk access during gameplay. Lua provides for this; you can load a script file into a buffer, and then execute it from memory instead of using the direct file access method. You could still keep fast script change access by providing functionality from within your game to reload this buffer on command. You could also just leave the random file access in for development and switch to the buffered system when shipping your final product.

The Lua script encapsulates all the state transition logic into a single `if . . . then . . . else` block, so it might seem like this is a step back, design wise and organizationally. We could do the exact same construct from within our C++ program, and the program would also run faster. The simple state machine logic necessary to run our test bed is small enough that this method is fine, but in a game of any size or complexity, this would definitely not be true.

In a real game, you would probably want to create a simple, barebones FSM system on the game side, with the logic behind it being completely data driven. The game side state machine would consist of a list of states and a block of perception data (that is exposed to your Lua scripts). The game side code would also include a list of "behaviors," which encapsulated the code necessary to actually perform actions within the game world. The scripts themselves would be organized as separate Lua functions for each game state; each function would consist of behavior calls and the transition logic for just that state.

Each time the AI engine would call the Lua script, it would first update a global variable in Lua that stored the name of the current game state, which could then directly translate into the corresponding Lua function to handle that state. To add another state to the game, the scripters would just make a new function in Lua and put the new function or game state name into a global table of function or game states that is exposed to the game side. When the game loads (or reloads), it would grab this global table of game states and construct a barebones state machine structure at run time. Listing 18.11 shows a simple C++ example, along with the Lua script that would be used.

LISTING 18.11 A Better Way of Handling Lua-controlled FSM Transitions

```
//FSM Game code
LuaPerceptionExport();
UpdatePerceptions();

lua_pushnum(m_luaState,m_currentState);
lua_setglobal(m_luaState,"gCurrentState");
lua_doFile(m_luaState,"transitions.lua");

UpdateMachine();

_____________________
-example.lua

-game state functions
function gsStateStand()
    -start/stop behaviors based on Perception data

    -check for transitions
    -would call a "ChangeState() function, which would
    - change the C++ m_currentState variable
end

function gsStateRun()
    -do run state
end

function gsStateSit()
    -do sit state
end

-global table of functions, C code can
-access this in order to find out the number of
-game states in the system, and their order
funcs = {gsStateStand,gsStateRun,gsStateSit}

-executes the current state function
funcs[gCurrentState]()
```

A system like the one just described would have the scripters themselves declaring all the possible game states and, hence, would not need a programmer to be involved in adding or removing them. The result is that you separate your game content into two "camps": perceptions and behaviors are in the code, and logic and

TeamLRN

configuration parameters (things like attributes or numbers that require tuning and balancing) being covered by scripting. The scripters can arbitrarily set up game states, to facilitate any logic tree they desire, and their only requirement of the programmers would be the list of available game perceptions and behaviors that are implemented in the game code.

PERFORMANCE OF THE AI WITH THIS SYSTEM

Our script is very small, so the performance hit of running through the script file, and interpreting the entire script every frame is negligible. The game runs quite well with this system and is quite easy to tune and tweak given that you can edit the game logic as the game runs.

However, this isn't going to scale well for larger games. What if you had hundreds of character states, as well as hundreds of different characters? You would need to complie huge files repeatedly to traverse large `if` blocks. This is not a system you would want to work on. Performance would be atrocious, debugging would be a nightmare, and extending the system would be hard work indeed. Instead, you would want to split up your system into some kind of modular organization, possibly using the system detailed earlier.

For larger implementations of a Lua system, or for games implemented in a multithreaded environment, you could use the more advanced, but very useful Lua constructs: threads and coroutines. Threads are full separate Lua state environments, whereas coroutines are just re-entrant functions that can be paused and resumed at will. By using a system of coroutines, you could set up many different scripts to run the various AI entities in your game world and, with clever programming, not worry about any one script eating up all your CPU resources.

Pros of Scripting Systems

Scripting within an AI engine provides a means by which less technical staff can create and extend logic, tune systems and behaviors, and even completely change whole AI systems (if the engine is set up to be fully data driven). Some of the things that scripting does well include rapid prototyping, increased accessibility, speed of tuning, user extensibility, and easy scaling:

Rapid Prototyping

Any time you are forced to abstract game perceptions and behaviors to a higher level (as you will have to do when deciding what to expose to your Lua scripts), you distill your game to the concepts that make the most sense. Once these basic concepts are communicated to the scripters, they can immediately begin scripting

advanced logic and strings of behavior, as well as start tuning in-game settings and events. Having fast turnaround when developing scripts (by providing script reloading in-game) accelerates the rate at which decent content can be added to the system.

Increased Accessibility

By using a more focused, higher-level language, you remove the barriers from entry that a real programming language incurs and thus, allow more people feel confident to try to "get under the hood," if you will. Imagine how many more people might try and fix their own cars if they knew that the parts involved were simple, straightforward, and intuitive. By designing an easy-to-use scripting system for your game, you actually *entice* designers (and end users, if you allow it) to dig in and make script changes and additions.

Speed of Tuning

Scripting allows for much faster tuning of AI behavior (than is afforded by code changes, followed by a compile and link, followed by a game restart), as well as making available an open means by which more people can perform the tuning.

Provides a Means for User Extensibility

The same system used to encode the AI and gameplay content for the production game can be included in the release of the product. Some companies include almost the full suite of development tools along with their games. This has especially become the norm in the FTPS genre, where user-created mods have increased the shelf life of games from months or weeks to many years. Some of the scripting languages included in these highly open games rival real programming languages in complexity (like QuakeC, or UnRealScript), but they also allow the end user to control, change, and create almost any effect within the game. Through extensive use of the scripting language, people have created flight sims, racing games, and platformers in FTPS games.

Scales Well

As the number of things under the control of the scripting system increases, the real power of scripting makes itself known. The overhead that scripting brings becomes less and less the more that you can leverage the data-driven paradigm. Even though each individual system might have separate script functionality (you might have a different chunk of functions exposed for use in scripting behaviors, versus the types of game functions that you would want to use to script state logic, or cinematic sequences), but all these game systems can be built on one unified scripting platform, if that platform is open and flexible enough to allow it.

TeamLRN

Cons of Scripted Systems

Scripting does have some negative points, but these can generally be overcome with careful design, forethought, and the heavy processing power of today's game machines and PCs. Some of the things to consider when designing your scripting system include speed of execution, debugging difficulty, scripting, power, and number of systems needing maintenance:

Speed of Execution

Any interpreted system is going to be slower than programs that have been compiled into native machine code. Lua scripts can be precompiled into bytecode, which gives it a small speed boost (and has the added effect of being unreadable, in case you don't want users looking into the particulars of your scripts), but not much. This is the reason behind the big push to make any given scripting system easily integrate into one of the more commonly used, general-purpose languages such as C/C++. This way, you create the functions that require speed in the compiled language, while using script to develop the parts of the game that aren't as performance sensitive. Note, however, that Lua has been shown to run many times faster than most scripting languages because of its small size and clean coding.

Debugging Can Be Difficult

The primary beef with scripting systems has always been in the realm of debugging. Two separate issues are the main problems. First, scripting environments (especially custom-developed scripting languages) don't have the level of debugging tools (dedicated debuggers, profilers, asserts and internal error checking, syntax checking during compilation, etc.) that the big languages have unless you specifically code them yourself. Second, the people who usually write the scripts are not as technically minded as dedicated programmers (that was the point, right?), so they tend to lack some "common sense debugging techniques." These techniques would include binary bug hunting (useful for some scripts where a bug makes it do nothing at all; run *half* the script. If there is no error, run other half. Error? Then split that chunk in half and keep going), using print statements to watch script variables change while the game runs, putting error checking into the scripts, or even simple logic gymnastics (this AND that OR not this OtherThing AND these; now invert the whole thing) is sometimes beyond the grasp of some purely creative types. Usually, people learn techniques over time (just like most programmers did), and languages that have a decent learning curve *scale* are best for this. Lua, Python, and some of the other lightweight, yet full-featured scripting languages offer this. You write very simple scripts quickly, yet learn advanced features of the languages, and do some powerful things, as you progress through the learning curve.

Looking Scripted

For a while, games used scripting languages just like Hollywood used scripts, to define large chunks of game that was essentially being performed in front of you. As you entered a room, control would be taken away from the player, the camera would swing dramatically in to reveal three monsters emerging from some portal. They would then take their places, a leader type would approach you, yell some invective your way, and tell you all about how he was going to kill you. Meanwhile, the room would begin filling with fog, and the weird hat you found in the last room starts glowing with some special purpose. This is all very nice, at least the first couple of times you see it. But if you have to retry this battle many times because it is a difficult encounter, or many times during the game this kind of scenario ensues, it starts to grate on the player, and he feels like he's not really playing the game but, rather is being led around by his nose to witness the "next great vision" that will be presented to him. Some games do this very well, and people love it. But many more games do this badly, and it becomes a kind of torture to have to sit through countless lengthy sequences of noninteractive silliness. But looking scripted doesn't have to include long cinematic sequences. It could just be that every time you talk to the drunk at the bar, he burps, then falls down, then says, "Leave me alone!" and then gets back up and reseats himself. If the player talks to the drunk three times in a row, he's going to know there's a very simple script that controls the drunk's behavior. Even though it may actually be a small piece of game code, this kind of thing is usually branded as scripting nowadays. The richer you make your scripted responses, the better they will look the first time. The next time, all the illusion will be destroyed, and your script will be discovered. The way to fight this is, of course, to not do it. Granted, some games definitely want the exact kind of game situations described, and this is fine as long as they don't overdo it, or do it in parts of the game where suddenly a great scripted reaction sequence is going to become a tedious ordeal for a player who is already having trouble with a section of your game (and thus is having to replay the area with the script).

The Question of How Much Power

One problem with data-driven AI systems in general is the question of *when to stop*. You've made the state transitions script based. But in doing so, you've noticed that with a little more extension to the system, you can offload the state machine definitions to the scripters. A little more code, and they could *technically* define the vast majority of actual AI behaviors. A touch more functionality to the scripting language, and they could also define perceptions, including the actual equations for calculation, the update frequency, and on and on. The problem is this: if you make too little script functionality, you've added overhead to the game for very little payoff, but if you add too much functionality, you run the risk of overwhelming the scripters with the entire job of coding a game, except in a scripting environment

TeamLRN

that is only half as user friendly as a real programming language and that runs at three-quarters the speed. Plus, you run the risk of giving the scripters so much power over how things happen in the game that they can't help but constantly break (or be on the edge of breaking) the project by continually having small errors or combinations of factors acting against each other. Where to stop is a serious question that needs serious attention when designing a scripted system. The answer lies in the type of game you are working on, the level of functionality you need in your scripts to perform the things you need, and the level of organization and control you want of the content in your game.

You're Now Maintaining Two Systems

When you sign up to write a script-based AI system, you're really signing up to provide another completely separate product in addition to your game. You now have two jobs. In the first one, you are producing a game, with a target audience that has specific needs and assumptions about how the game should look, feel, and play. In your second job, you are producing a light programming language, with a completely different target audience (although in some cases the scripting system will be given to the end-product users), who also has very specific needs and desires about how the scripting system should work. When deciding to incorporate a scripting engine into your AI system, you are going to have to become somewhat of a people person because the difference between a good scripting language and a bad one lies in how easy it is for your scripters to use it; this initially means "How good are you at teaching your scripting language to others." Be prepared to minimally write many example scripts (and they'd better be decent quality because they will most likely be cut and pasted directly into the game with minor changes for quite a while, as scripters learn), but also to field questions regarding everything from basic debugging to simple programming methods. Things like basic logic, organization methods, and different styles of using loops and data. In addition, the feeling of power that scripting gives to people who were previously nonprogrammers is both contagious and snowballing. The more things you allow scripters to be able to change in the game, the more things they're going to *want* to be able to change, and therefore the more features they're going to want you to add to the scripting language. You must allow for flexibility in the system for future extensions. But remember the last point in the list of cons; too much functionality in the scripting system can be a problem as well.

EXTENSIONS TO THE SCRIPTING PARADIGM

Scripting systems open up to a whirlwind of possibilities, by their very nature. So extensions to a scripted environment are many and really only restricted by you and

your scripter's needs. Some advanced features for scripting languages include custom languages, built-in debugging tools, smart IDE for writing scripts, automatic integration the game, and self-modifying scripts:

Completely Custom Languages

Another common means of creating a scripting system for your game is to go the way of the classic "Lex & Yacc" route. These are tools specifically for creating compilers, but they can also be used to streamline the creation of custom scripting languages. The process is fairly straightforward: first, you generate a *grammar* file, which details the lexicon for the language you're developing, with the help of Lex. It allows you to set up all the rules for your language, by means of special rules called *context-free grammars,* which is another way of saying that these grammatical rules can have wildcards, or nested definitions. You then run Yacc (which stands for Yet Another Compiler Compiler, aptly enough) on this file, and it generates the C code necessary to parse a file using the specified grammar, both into bytecodes (if we desire) and for syntactic correctness. Games that use these tools tend to keep Yacc around as the in-game interpreter, a process that is sometimes called just in time (JIT) compling, which refers to the fact that the script is compiled "just in time" for the game to use it. By using these tools, you can generate completely custom languages that still allow the flexibility and powerful parsing ability shown by more commonly used compilers, without having to code your system completely from scratch. Your scripting language can still use complex block structures, a variety of different operators and keywords, with whatever syntax works best for your game.

Built-In Debugging Tools

Of paramount importance if your system is going to be large or fairly complex is a means for determining why things aren't working. Coding debugging functionality directly into the scripting system, or with an embedded system, giving the scripters immediate game side functions to call that allow them to track down bugs and their causes is a great way to build a scripting system in the first place. Simple tools like "watch" functionality (allowing a variable's value to be constantly visible), break statements (which would stop the game if a specific line of script code is reached) and the ability to single step through a script (running the script one line at a time while being able to see variables and such) will accelerate debugging scripts just as they do regular code. Also, visual debugging functionality that would show up *in game* is very useful. The scripters should be able to set up things within a script to write out text to the game screen, or icons, lines, and anything else they might need to depict what's going on inside the script, as well as allow scripters to

TeamLRN

test out their code. These elements could be ignored once a game goes to production, by simply setting flags within the script parser to ignore them or only compiling the functionality into debug projects and not released code.

A Smart IDE for Writing Scripts

Stupid bugs in code are sometimes the hardest to find. This is even more true for nonprogrammers, who might stare at the line `heroHitPoints = 0` for days and never realize that they needed to use a double equal sign (==) instead of a single, or simply have a function call in all lower case letters instead of having the correct capitalization. By providing an IDE, or simply some kind of dedicated editor for your scripting system, you could provide uses with some level of on-the-fly syntax checking (by interpreting the script, but only for syntactical issues), as well as provide things like keyword finishing (you would type the first three letters of a keyword, and then pressing tab would finish it, as well as correct the case), finding things that are misspelled or missing, and the like. These features will help to erase simple bugs before they ever enter the game and will make the system easier to learn and use overall.

Automatic Integration with the Game

Scripting systems that allow designers to add content to the game (like defining game states, behavior, etc.) usually suffer from having to give *something* back to the game, to be compiled into code so that the game can use the new elements. But, as stated earlier in the chapter, you could have your tool also define data that would be used by both the script and the game engine itself when defining key in-game data structures. The game could read this data, as well as the scripts, and rebuild entire system's sets of objects or properties such that nothing extra would need to be added to the game for header file information. Another method, if there simply must be some programmer work for any new addition, is to simply use the scripting system as a kind of feedback tool for the programmers, as sort of a "request" system. Any time a new Widget is requested (through whatever means you use; be it a special data file that is edited, or just the inclusion of a `RequestedWidget` command from within a script itself), the system would just add it to a special list that programmers would have access to, to find out what the scripters require for new functionality. A useful addition with this system is to allow the scripters to request a new feature and give a short description of it as well, so that the request is not just an empty shout in the dark. The programmers would then actually implement the requested widget, add it to the game side code base, and the next time the scripts get loaded, the system could recognize that the requested widget has been completed, and move on.

Self-Modifying Scripts

Scripting systems, being data based, open the door for a rarely used technique in the realm of code based AI: self-modifying behavior. There's no reason why a script-based system could not keep track of specific behaviors that work (and don't work), and bias them up or down accordingly by actually writing data to their script files. The system could append additional rules onto its script (or remove some) as the consequence of some event during a game. Behavior like this could be thought of as a kind of learning, or it could go further. In fact, there is a field of AI called genetic programming that deals with this phenomenon. Unlike genetic algorithms, which strive to use genetically derived methods to perfect *algorithms* for finding solutions to problems, genetic programming deals with using genetic methods to actually write whole *programs* to solve problems. In effect, the system is searching for the ideal script in which to perform its job, instead of the ideal parameters to use within that script. The problem with using real code in genetic programming is that most genetically created code segments are garbage, and wouldn't even compile, much less run. But with a more abstract, higher level scripting system, you can begin to see the possibilities for a system that could generate scripts, test them, and genetically find superior programs that the AI could run to perform tasks. This is obviously a very different and ambitious path to take, but to the victor go the spoils.

OPTIMIZATIONS

The performance of any given scripting system really relies both on the level of its use within the game and on the functionality of your language and how structured your overall system is. The simple configuration language implemented early in the chapter is barely more than a file parser and could be run with little to no impact on a game (especially because it is primarily designed for setting variables and flags during level load time, rather than during gameplay). If you were to use it as a base to write a serious scripting system (which isn't advised because it has nothing of even the simplest scripting language niceties), you might have to perform considerable file-based gymnastics to get everything working. Lua is a fairly lightweight language, and as such runs pretty fast for an interpreted system, but you still wouldn't want to write your AI system's pathfinder in a Lua function that required a lot of tight search loops and tons of data being passed back and forth.

For simple scripts that are just declaring behaviors in reaction to some event, a scripted AI system might actually make your game AI *faster* than if it were trying to calculate all the things necessary to get the AI to procedurally perform all the actions, one after another, that a script might encapsulate, because the AI is simply being given a list of things to do, without recalculating anything midscript.

TeamLRN

When dealing with large scripts, however, filled with heavy amounts of logic and game side function calls, you might run into performance issues. Thread or coroutine based scripting languages can help if you find that your AI scripts are taking too much time, but only if the problems that are taxing your CPU budget are re-entrant, meaning they can be incrementally solved through a few game loops.

DESIGN CONSIDERATIONS

Scripting systems have a home in many different styles of games, in both old-school games that relied on many patterns to portray behavior and modern games that increasingly depend on large quantities of richly designed and detailed content. Scripting systems allow nonprogrammers to define game content in a fast, generally game-safe way, without the bottleneck of having to rely on dedicated coders to help. These systems also allow fast tuning and the like because they allow changing game behavior without actually changing the game code itself and requiring a recompile.

Types of Solutions

Scripting systems work equally well in the venues of low-level solutions (where scripts might be used to describe behaviors, animation selections, or simply set up game variables with tuned settings that radically change game actions), as well as high-level AI solutions (including strategic decisions involving many game elements, or scripted events that require many behaviors to be linked sequentially). They are a bit better suited toward the high-level work because they usually have an even higher abstracted view of the game world than the game code side of the AI system, although specialized parts of a game's scripting system might involve itself with a very low-level part of the game, such as selecting specific animations for a particular behavior as listed before.

Agent Reactivity

Scripted systems can give any level of reactivity you require, although it does depend on the kinds of scripts your game allows. If you can only use scripted behavioral responses that are somewhat limited or lengthy, your system might be seen as being *too* scripted, and as such, not very reactive. But if you use your scripting system to merely encode logic for a state-based AI system, then you can expect the reactivity of your agents to be commensurate with the update frequency of your system.

System Realism

With the question of realism, scripted systems have something of the largest ability to perform realistic, unique behavior in response to a given game event among many of the various game AI methodologies. The problem, is that the richer and more scripted the behavior is, the more proprietary it is. It can only be used in one limited part of the game. Thus, if you have a completely state-driven, reactive game, and suddenly have one part of your game with massive proprietary scripting giving every reaction realistic or content rich behaviors, it's either going to seem out of place, or it's going to make the rest of your game seem boring. Thus, rich content somewhat forces more rich content to be built into your game. In addition, overly scripted behavior behaves very unrealistically if it can be caused to repeat too much by the player because it will make the game characters seem very robotic and unresponsive. But without the issue of overly scripting a game (in chains of behaviors given to game characters), scripting systems allow increased realism without overly robotic behaviors simply because they allow a greater amount of behavioral content to be included in the game.

Genre and platform are of almost no concern to scripting systems. Games with a large number of AI entities or differing game situations are ideal for scripting systems because the main reason for using them in the first place is that you require more content or tuning than can be easily handled directly (or indirectly) by the programmers on your project. If you are making a game with one main character, one enemy, and three behaviors apiece, you could most likely safely write everything in code and quite easily tune the entire system from the game side. You still might want to use a configuration script system to set game specific variables and properties from data, so that you can set tuning values and not have to recompile the game, but this would depend on you. Scripting systems do require additional memory (in overhead for the parser, interpreter, and data size), but usually make up for it by simplifying the code base so they can be implemented on even the smaller, more restrictive platforms. Care must be taken because, unless specifically designed otherwise, scripted code will always run somewhat slower then compiled code does, and this kind of performance hit must be considered.

Development Limitations

Development limitations are one of the most important things to consider when deciding on using a scripting system in your AI engine. You have to determine if you have the time to implement the language, the time to teach it to your scripters, and the extra effort needed to debug the scripts and the game code. You will save time when it comes to tuning the final product, but this may not offset the costs incurred by all these hurdles.

TeamLRN

Entertainment Limitations

Tuning difficulty settings, balancing specific behaviors, and other entertainment concerns are generally the reasons why a team chooses to implement a scripting system because these issues are either very important to the product, or the level of this kind of work is incredibly large. Scripting, using either a basic or a full system, will aid in all these endeavors.

SUMMARY

Scripting systems provide game makers with a means to get more content into the game without needing more dedicated programmers. Scripting systems use a simplified programming system to build AI elements, logic, and behaviors. Scripting languages can be anything from very simple representational token-based systems, to full-fledged programming languages themselves.

- When designing your scripting system, consider the kinds of functionality that the language will require, how the runtime engine will interpret the scripts, the overall size of the scripts, and the potential users of the language.
- A configuration script system is a simple, text parsing system employing user-defined tokens.
- The simple configuration scripting code included in this chapter is broken down into three parts, `Token`, `Parser`, and a number of callbacks, each one associated with a particular `Token`.
- Because of the generic nature of the simple scripting system given here, extensions to the method are unlimited. However, the lack of any real features found in more robust scripting systems might keep you from investing any time and effort with such a rudimentary foundation.
- Embedding Lua as a scripting language is becoming a popular shortcut to rolling your own language. Lua is small, fast, easy to learn, and integrates well with C/C++.
- Lua features include dynamic typing, garbage collection, LISP-like associative arrays called tables, and a random access stack structure for passing values back and forth between scripts and the host language.
- To integrate Lua into a game environment, you compile the libraries into your game, include the header files, instantiate a `lua_state` interpreter, register any host language functions with the interpreter, and finally pass the interpreter data, direct commands, or whole files to execute. When the game is done, you close the interpreter.

- Scripts can be compiled into bytecode, which makes them execute a bit faster, but also encrypts them so they cannot be read.
- The test-bed implementation places all the state transition logic into a Lua script. Given a large-scale game, a more robust implementation would be to put the entire state machine into Lua, and just pass perception data and register behaviors with the scripting system. Each state could have its own script function for modularity and maintainability.
- Lua supports threads and coroutines for use in large scripts that must be completed across game loops, in sequentially written scripts that have pauses, or in multithreaded game environments.
- The pros of scripting involve rapid prototyping, increased accessibility, speed of tuning, and easy scaling.
- The cons of game scripting, which can in most cases be overcome, involve speed of execution, debugging difficulty, becoming "scripted looking," balancing the power versus ease of use of the script system, and that you now must produce two products (the game and the scripting language).
- Extensions to base scripting systems commonly are built-in debugging tools, smart environments for script writing, automatic integration with game side code, and self-modifying script code.
- Optimizing a scripting system involves the specific game in question and the specific scripting system in question. Some scripting might actually improve game performance because the AI character would be just "following a script," instead of calculating a response. For re-entrant problems, threading or coroutine based scripts might help with performance problems.

TeamLRN

19 Location-Based Information Systems

This chapter, like Chapter 17, "Message-Based Systems," will cover a family of AI techniques that are secondary to an overall game AI decision-making system. However, unlike messaging, which is more of a communication technique, location-based information systems (LBI) are helper routines that augments decision making by providing additional information to the intelligence engine. This extra information is brought forward in the form of a centralized bank of data that is tied to the game world itself in some fashion. LBI could be thought of as very specialized perception data, which might also include embedded logic or lower-level intelligence.

LOCATION-BASED INFORMATION SYSTEMS OVERVIEW

The discussion of LBI will be broken into three common (but not exclusive) categories in this book: influence maps, smart terrain, and terrain analysis. The first part of this chapter will give a brief discussion of each of these categories, and then later, we will implement some simple influence mapping techniques. The other two techniques are considerably more complex and much more game specific (and hence, a full implementation would go beyond the scope of the method explanation), so this chapter will not fully implement these techniques, but will broadly discuss implementations of smart terrain and terrain analysis, both for test application and for real world games.

Influence Maps (IMs)

Influence mapping is slowly becoming one of the most commonly used secondary AI techniques in games. Its generic structure and open-ended usage make it second only to finite-state machines (FSMs) in ease of implementation and adaptability to different game AI problems. The term *influence map* refers to using a simple array of data, where each element represents data about a specific world position. IMs are

TeamLRN

usually conceptually thought of as a 2D grid overlaid on the world. The resolution of this grid (and thus the number of elements in your influence array) depends on the minimum size of game space that you need to tag with information. Where your game can compromise between data size and influence accuracy will also determine this resolution. So, if you absolutely need specific information for every square inch of a large game world, your IM will have a very high resolution indeed (and take up a lot of memory in doing so). Many games will employ multiple IMs, to either help with memory and searching costs, or to provide different levels of game space resolution to the various AI systems. Thus, a real-time strategy (RTS) game might have an IM with very low resolution (say, each element is an entire game screen) that reflects the amount of each resource within. The game uses this for high-level planning when building the town and determining which direction in which to expand the town. You would want to expand your base (and thus your main defenses) toward more resourceful areas to facilitate future expansion. Our example game also employs another IM, with a much higher resolution (each element is now approximately equal to four of the smallest units standing together) that keeps track of the number of units that have been killed in each grid square. This map is used to affect the pathfinding engine so that units will not continue to take paths into areas that have a high mortality score.

In games with worlds that are heavily 3D (that is, comprising many vertical layers built on top of each other), a more complex data structure is necessary; these can be represented with correspondingly layered IMs, or by building the influence data into the navigation mesh used for pathfinding. Another technique might be to only use IMs where you need them, or *local* IMs. For example, a battle between forces in an RTS game might start anywhere on the map. You might want to have a heavily detailed IM during battles to coordinate forces, but not want to use the memory to have a global IM for the game world that would provide the level of information you require. Instead, you could implement a heavily detailed, but local IM. The system would detect a battle and set the coordinates of the local battle planning IM to be some distance out from the "center" of the battle. The center of the battle could be determined using a different, lower-resolution IM that keeps track of population data or fighting locations. Thus, the global IM is still constrained, but local information can become quite detailed.

Many games still use what could be called a *high-level* approach to having game characters interact with each other. Each character will consult some large list of game characters, and determine whether interactions are required or necessary. Using this approach is somewhat akin to collision detection, but with the AI system being more interested in perception distances rather than collision distances. In this way, it's like each character is high above the game world, looking down at all the information within, picking and choosing the items of interest. This is not how people do things in real life, however. People have limited senses and can only react

to things in their immediate areas. Keeping track of player locations within an IM allows for a more *low-level* approach to game character interaction. Limiting the information that any game character has access to from a global view to a more localized perception base is much more realistic. The IM also provides a central place for game entities to search for interaction data, provides a platform for other types of information to be stored, and helps decouple individual characters from having to include global access to code and other game entities.

The nature of IM's generic data structure can be used to construe location-specific data in an infinite number of different and useful ways, limited only by your imagination and the relevance to your game in particular. In fact, this basic system can be (and usually is) the central repository for the other two LBI techniques to communicate with the rest of the game.

Smart Terrain

Made popular by the *Sims* games (in fact, the term was coined by Will Wright, the *Sims* creator), this technique places logic and behavior data for how to use various world features and game objects into the objects themselves. In the *Sims,* characters in the game were motivated by needs that could be satisfied by interfacing with the various objects in the game. The *Sims*'s programmers could apply different attributes to each object, which corresponded to particular needs that the game characters required. Thus, a microwave oven satisfied the Food need, and a bed fulfilled the Sleep need. The character then navigates around the world, trying to gratify his unmet needs at any given time, doing so by listening for broadcast messages from any nearby smart objects. These messages would communicate to the character the need categories each object satisfied, and he would then be free to use any world object to satiate himself. Of course, this is a simplified view of the workings of the *Sims* game, but you get the idea.

The technique also bundles all the interaction animations, sounds, and any other special data that a character would need to use the object into the object, thus making the object completely self-contained. In this way, new objects could be added whenever the developers wanted and would require only two things: that the new object contained all the data that the game requires to be a fully functional game object, and that the new object assuaged one or more of their basic needs so that the characters would actually seek out and use the new object.

Terrain Analysis (TA)

Tracking various attributes and statistical data within an IM is only part of the problem. You must also make use of this data. TA is a family of methods that have made use of IM techniques to provide AI systems with increased strategic information about maps, especially randomly generated maps that haven't had the benefit

TeamLRN

of ever being touched by a level designer. Even with custom created maps that have had preprocessed TA, most games have some element of dynamic change within the game world, which again calls for additional, on-the-fly analysis of the map. TA methods are best described as specialized pattern recognition within an IM data array. You search through the IM array, looking for motifs that can be exploited or that need to deal with strategic and tactical decision making. Any TA done on even a medium-resolution IM array can be very computationally expensive because of the large amount of searching required for many pattern-matching algorithms. To counter this, some games use TA systems employing nonbrute force methods to determine patterns, such as neural nets or fuzzy logic systems, that can algorithmically find patterns within the IM array.

HOW THESE TECHNIQUES ARE USED

IMs are becoming very common in game AI, especially RTS games (although other genres, like role-playing games [RPGs] and action and adventure games are following suit). Some examples of how games have used IMs include occupance data, ground control, pathfinding, danger signification, rough battlefield planning, simple terrain analysis, and advanced terrain analysis:

Occupance Data

Occupance data means tracking various populations within the game. An easy use of an IM is to keep track of the number of specific game objects within a certain area. You might want to keep track of all combat units, specialized resource locations, important quest items, or any other in-game object. Simple occupance data can be used to help with obstacle avoidance (overriding the pathfinding system with local detours around occupied terrain), give rough estimates of various game perceptions (army size, town density, the direction to the most powerups, etc.), or any other task that requires quick access to localized population data.

A very common usage extension of the occupance IM is the familiar *fog of war* line-of-sight system that almost all RTS games have. Initially, this fog covers the map completely, so that you are forced to explore the map to find resources and enemy towns. Exploring the terrain removes the fog and allows you to see the physical details of the land, but you must have a unit within line of sight of any given location to see the *current* activity within it. The game uses occupance within the map to uncover areas of the map that are visually within line of sight of any of your units.

Ground Control

Ground Control is finding *actual* influence of game ground. Although the term *influence map* is used in game AI as a loosely defined data structure, the phrase historically refers to techniques derived from the field of thermodynamics (in determining heat transfer) and field analysis in general (such as electromagnetic fields). These same equations can be used in a game setting and can quickly determine which player has control over which part of the game map.

The algorithm for this is simple: First, zero out the entire map. Then, assign each grid square a value based on its team-specific control (the magnitude represents the degree of control, the sign differentiates teams; a positive value for player A's units, a negative value for player B's units, with more complex schemes for more than two players). Then go over the map again, and for each map square, add up the values of the squares surrounding it, scale by some amount (to prevent value overflow) and add that to the square's value. Repeat a few times to disperse the influence out until you achieve a stable state. Player A controls the squares that have positive values, and player B controls negatively valued squares. This technique will provide you with a way of measuring *global,* as well as local, control. In areas where no one has direct positional control, the influence numbers will have propagated from the nearest units that do have direct control. You can then determine large regions of contiguous control, which carve the game world into areas of player affect. You could then sum all the areas to determine who has the *most* control over the map, for king of the hill scenarios.

Pathfinding System Helper Data

When provided with additional information about a specific area, the pathfinder can help smooth the solution through a tricky part of the map by giving a shortcut, or allowing the AI-controlled character to use a specific map feature such as a teleporter or ladder. The pathfinder data may include things like passability, relative to terrain features (such as hills or cliffs) and to terrain type (land versus sea) or even which player currently controls the areas of the map you want to traverse. Some games use a simplistic potential fieldlike technique for augmenting a path node–based system, implemented through designer-placed influence data throughout the map, or to be procedurally determined. Usually, this would be used to help (or force) steering of AI characters away from hazards or places the game developer just doesn't want a bunch of roving monsters to congregate. A potential field IM might also change and adapt to game conditions over time, giving characters the appearance of learning as the game progresses. An in-game example of this might be in a sports setting (for example, a hockey game), where a small offset IM field is used in conjunction with a formation system. When the game starts, the offsets are

TeamLRN

all zeroed out, so the CPU team uses the standard formation for positioning. But as the human player starts to play against the CPU team, the offset IM is used to perturb the formation of the computer players toward places that the human often uses for travel or for passing the puck. In this way, the human is forced to change his game to continue to score effectively because the computer team is continually fine-tuning its formations to the human's style of play.

Danger Signification

Another useful implementation of IMs is to keep track of areas where bad things have happened over a period of time. This data can then perturb the regular AI behavior, so that the AI doesn't continue to perform the same actions that result in the continuation of this harm. An example mentioned in the RTS chapter would be the human placing an attacking tower in the midst of a pathfinding route that the AI uses regularly. The tower then proceeds to kill the single line of units that continually trickle by the tower. If the AI had a danger signification system in place, the units would eventually stop walking by the tower (because the AI would use the danger data to influence the pathfinding cost of traveling through that area), and better yet, would send out an attack group to investigate what is causing the danger in the area. Another example would be an AI deathmatch bot in a FTPS game remembering areas where he was ambushed or sniped, so that he could avoid those areas, or just approach them differently.

Rough Battlefield Planning

The ground control method detailed can be used to quickly point out areas of interest in midst of battles. By looking for thin areas of zero (or near zero) value, you can easily find where armies have met, and are fighting for control of an area. This will tell you where the conflicts are, and where the front line of any given battle is. Large areas of near zero values are probably places where no one is in control. By knowing where the brunt of your force is, in relation to the main body of the other army (as well as the relative size of each army), you can focus attack direction, determine chances of winning to initiate additional charges or retreats, send reinforcements more intelligently, and coordinate attacks on multiple fronts more cleanly.

Simple Terrain Analysis

This includes somewhat more mathematical determinations such as *cover* (how much a given position is open to attack from any given angle), *visibility* (in some ways, the opposite of cover but also considers lighting concerns and line-of-sight

issues), and *height factors* (many games allow greater missile weapon range from higher ground and better visibility). The best areas of cover become sniping spots. Areas with low visibility might become sneaky back doors to other map areas. An area with high visibility might work as the target of an ambush if it is surrounded on some sides by high-cover terrain.

Advanced Terrain Analysis

RTS games routinely desire much more advanced TA methods to appear even remotely intelligent when playing against a human opponent.

Finding good choke points in a map, or places where movement or visibility are severely restricted, is a common way to use IMs. By scanning maps for this feature, the AI can set up ambushes if a choke point exists between two or more major map zones, especially if it's a *perfect* choke point (meaning there is no other way to travel between the two zones, and the other player has to travel through to win the game). Walls can be built into choke points to minimize the amount of walls that need to be built to close off a map area.

Another key usage of IM information is in determining the best way to build a town, defensive walls, or other structures. Towns should be built with some preplanning, to keep crowding under control (for pathfinding as well as for protection because buildings that are too close together can be splash damaged by large artillery type weapons *en masse*), to maximize future growth (by growing toward additional resources and minding routes of travel from older buildings), and yet allow maintenance (watch so that the town doesn't have too many flanks, and spread out the town's defenses to prevent weak flanks). The AI might want to use impassable terrain to its advantage by building the base of a town against it, therefore removing a line of attack. Humans build walls to slow down or redirect AI enemies that are pathfinding through an area (Figure 19.1 shows a human-made kill zone constructed of a maze of walls). These same sorts of tricks could be employed by an AI system to trip up players who don't micromanage their forces. But setting up these measures would require the AI system to scout for good places in which to do so, or else the behavior of the AI would look very silly indeed.

Determining important map areas (such as maps with severely limited resources or key strategic positions) is something that humans are extremely good at. Given a snapshot of the map layout (see Figure 19.2), a good human player will quickly tell you that you have to control area A because it contains most of the powerups and is fairly defensible. AI systems are usually quite bad at this type of determination, but an IM tracking this type of information (powerup location density, cover information, and choke points) would be quick to help this shortcoming.

TeamLRN

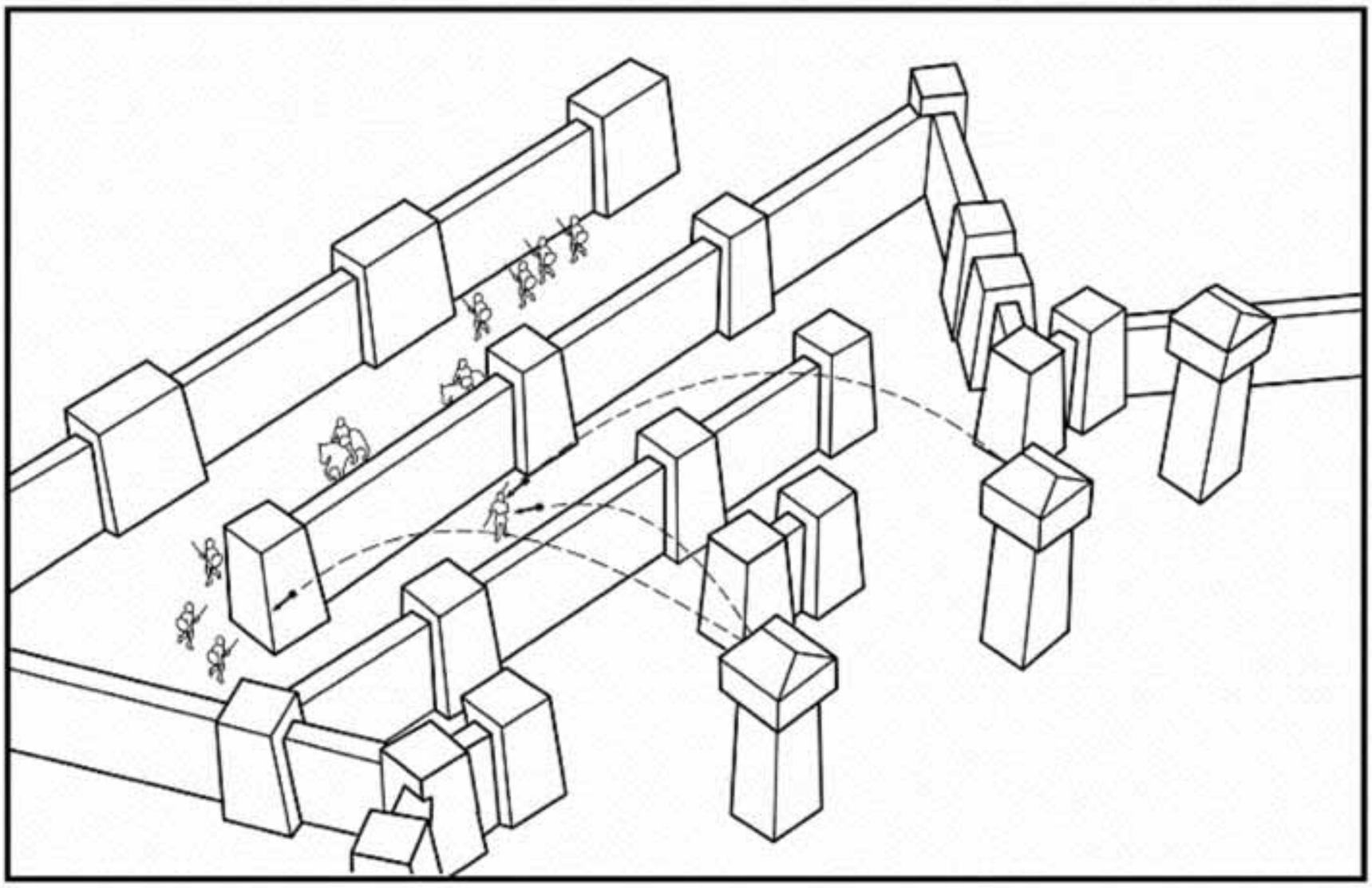


FIGURE 19.1 Walls built in a maze configuration to hold back AI attackers.

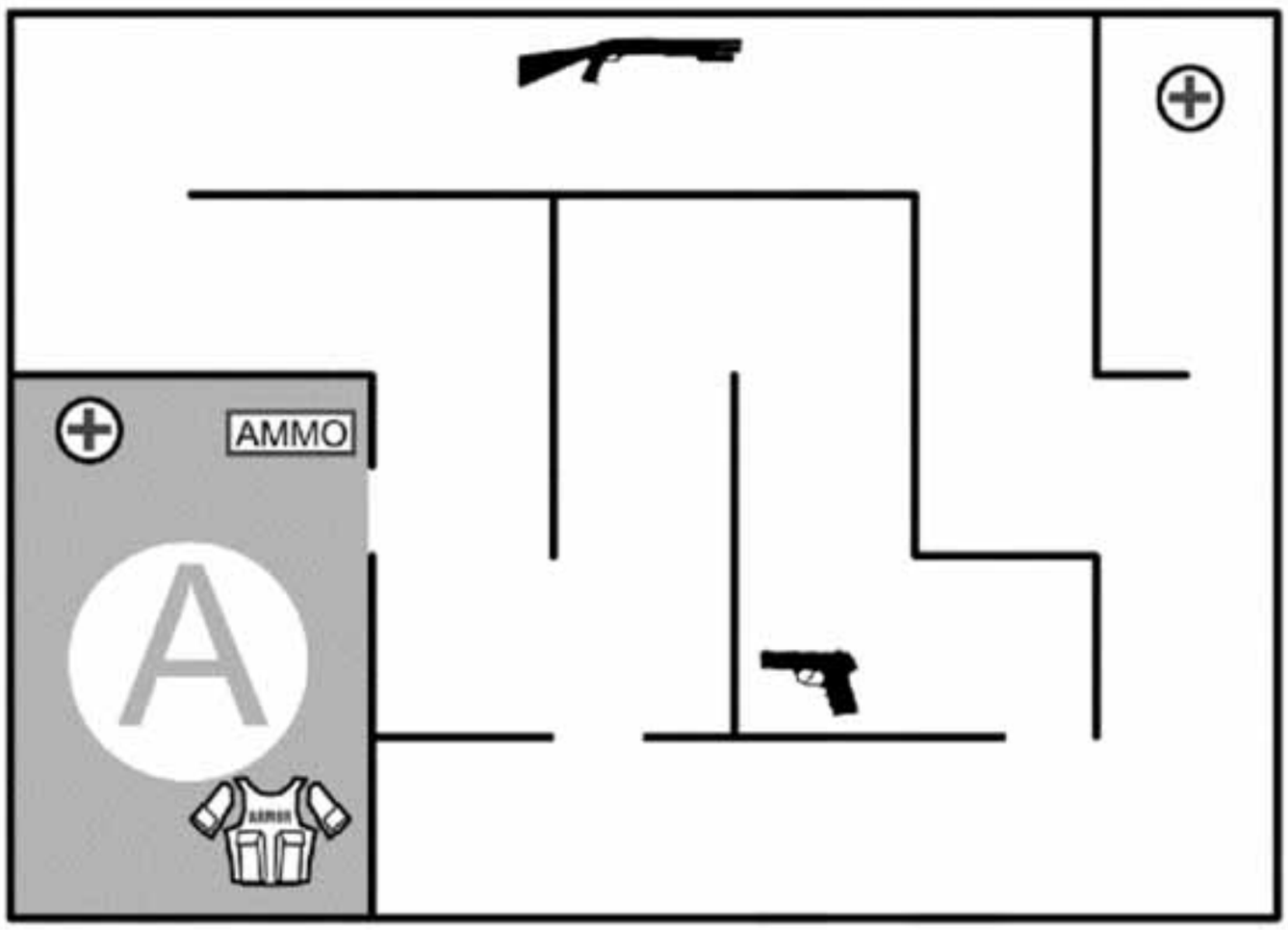


FIGURE 19.2 Example map with several strategic elements.

INFLUENCE MAPPING SKELETAL CODE AND TEST BED IMPLEMENTATION

In this section, we will implement a few different kinds of basic IMs into the test bed application. The implementation will be for illustrative purposes and will not affect the decision-making process of the AI subsystem within the test bed. Rather, it will show how easy it is to gather information and centralize it within an IM, and will display this information visually by means of a debugging system that allows both the grid and the cell contents to be drawn during the game. After each implementation, a discussion will follow about how the particular method could have been used by the test bed to improve performance.

Three simple types of IMs will be implemented, to show different ways to use them. These are the following:

- Occupance-based IM, which tracks where a given game object is in the world.
- Control-based IM, which uses a gradient to show an area of control around each game object and uses the notion of player sides.
- Bitwise IM, which splits the IM element's value into bitwise data components.

Each IM type inherits from the basic IM class, `InfluenceMap` (see Listing 19.1 for the header, 19.2 for the function implementations). As you can see, the first thing of interest in the base class is the IM array, `m_map`, which is an array of `int` (an unsigned 16-bit field). If you needed more or less storage within your IM array, you could change this to whatever you needed. You could even make a custom `struct` that the array would be composed of, but then you would have to change the class to accommodate this.

LISTING 19.1 `InfluenceMap` Header Information

```
struct RegObj
{
    GameObj*    m_pObject;
    int         m_objSizeX;
    int         m_objSizeY;
    int         m_objType;
    Point3f     m_lastPosition;
    bool        m_stamped;
};

typedef std::list<RegObj*> RegObjectList;
```

TeamLRN

```
class InfluenceMap
{
public:
    //constructor/functions
    InfluenceMap(int type):m_influenceType(type)
                        {m_drawGrid = false;m_drawInfluence = false;}
    ~InfluenceMap();
    virtual void Update(float dt) {}
    virtual void Draw();
    virtual void DrawTheGrid();
    virtual void DrawTheInfluence();
    virtual void Init(int sizeX, int sizeY, int wSizeX, int wSizeY);
    virtual void Reset();
    virtual void RegisterGameObj(GameObj* object);
    virtual void RemoveGameObj(GameObj* object);
    virtual void StampInfluenceShape(int* pMap,Point3f& location,
                                     int sizeX,int sizeY, int value);
    virtual void StampInfluenceGradient(int* pMap,Point3f& location,
                                        int initValue);
    int  SumInfluenceShape(int* pMap,Point3f& location,
                           int sizeX,int sizeY);
    int  GetInfluenceValue(int* pMap,Point3f& location);
    void SetType(int type) {m_influenceType = type;}
    void DrawGrid(bool on = true){m_drawGrid = on;}
    void DrawInfluence(bool on = true){m_drawInfluence = on;}
    int  GetSizeX(){return m_dataSizeX;}
    int  GetSizeY(){return m_dataSizeY;}

    //influence map types
    enum
    {
        IM_NONE,
        IM_OCCUPANCE,
        IM_CONTROL,
        IM_BITWISE
    };

protected:
    //data members
    int*    m_map;
    RegObjectList m_registeredObjects;

    int     m_dataSizeX;
    int     m_dataSizeY;
```

```
    int     m_numCels;
    int     m_worldSizeX;
    int     m_worldSizeY;
    float   m_celResX;
    float   m_celResY;
    int     m_influenceType;

    bool    m_drawGrid;
    bool    m_drawInfluence;
};
```

The influence system works by maintaining a list of registered game objects within its `m_registeredObjects` list. Game objects are thus freed of having to worry about updating themselves within the IM, because the system keeps its own list, but must remember to remove themselves from the IM system when they die in general.

Two functions, `StampInfluence()` and `StampInfluenceGradient()`, are used to actually write values to the IM. The plain version merely writes a value to a chunk of the array that is passed in size and position. The gradient version writes a decreasing value square gradient into the array starting at a certain position. These are generic enough that they are in the basic class, but they can be overridden by any subclass you create to perform custom writes to the IM array.

`GetInfluenceValue()` is an accessor for the map, and `SumInfluence()` is a generic function that merely sums the influence values at a position in a specific shape. Notice that all the functions dealing with the IM element map take an array pointer as a parameter. This is to facilitate custom IM types that may require additional scratch maps to perform multiple pass actions on the overall IM array.

LISTING 19.2 `InfluenceMap` Implementation of Important Functions

```
//--------------------
InfluenceMap::~InfluenceMap()
{
    if(m_registeredObjects.size() == 0)
        return;
    RegObjectList::iterator listObj;
    for(listObj=m_registeredObjects.begin();
        listObj!=m_registeredObjects.end();++listObj)
    {
        delete (*listObj);
    }
    m_registeredObjects.clear();
```

TeamLRN

```
}
//--------------------
void InfluenceMap::Init(int sizeX, int sizeY, int wSizeX, int wSizeY)
{
    m_dataSizeX     = sizeX;
    m_dataSizeY     = sizeY;
    m_numCels       = m_dataSizeX*m_dataSizeY;
    m_map           = new int[m_numCels];

    //clear out the map
    memset(m_map,0,m_numCels*sizeof(int));

    m_worldSizeX    = wSizeX;
    m_worldSizeY    = wSizeY;
    m_celResX       = m_worldSizeX / m_dataSizeX;
    m_celResY       = m_worldSizeY / m_dataSizeY;
}

//--------------------
void RemoveAll(RegObj* object)
{
    delete object;
}

//--------------------
void InfluenceMap::Reset()
{
    //clear out the map
    memset(m_map,0,m_numCels*sizeof(int));

    //get rid off all the objects
    if(m_registeredObjects.size() == 0)
        return;
    for_each(m_registeredObjects.begin(),
             m_registeredObjects.end(),RemoveAll);
    m_registeredObjects.clear();
}

//--------------------
void InfluenceMap::RegisterGameObj(GameObj* object)
{
    int sizeY,sizeX;
    sizeX = sizeY = 1;
```

```
    RegObj* temp;
    temp = new RegObj;
    temp->m_pObject      = object;
    temp->m_objSizeX     = sizeX;
    temp->m_objSizeY     = sizeY;
    temp->m_lastPosition = object->m_position;
    temp->m_stamped      = false;
    m_registeredObjects.push_back(temp);
}

//--------------------
void InfluenceMap::RemoveGameObj(GameObj* object)
{
    if(m_registeredObjects.size() == 0)
        return;

    RegObjectList::iterator listObj;
    for(listObj=m_registeredObjects.begin();
        listObj!=m_registeredObjects.end();++listObj)
    {
        RegObj* temp = *listObj;
        if((*listObj)->m_pObject == object)
        {
            m_registeredObjects.erase(listObj);
            delete temp;
            return;
        }
    }

}

//--------------------
void InfluenceMap::StampInfluenceShape(int* pMap,Point3f& location,int
sizeX, int sizeY, int value)
{
    int gridX = location.x()/ m_celResX;
    int gridY = location.y()/ m_celResY;

    int startX = gridX - sizeX/2;
    if(startX < 0) startX += m_dataSizeX;
    int startY = gridY - sizeY/2;
    if(startY < 0) startY += m_dataSizeY;
```

TeamLRN

```
        for(int y = startY;y<startY + sizeY;y++)
        {
            for(int x = startX;x<startX + sizeX;x++)
            {
               pMap[(y%m_dataSizeY)*m_dataSizeY+(x%m_dataSizeX)]+=value;
            }
        }
    }

    //--------------------
    int InfluenceMap::GetInfluenceValue(int* pMap,Point3f& location)
    {
        int gridX = location.x()/ m_celResX;
        int gridY = location.y()/ m_celResY;
        return pMap[gridX,gridY];
    }

    //--------------------
    int InfluenceMap::SumInfluenceShape(int* pMap,Point3f& location,
                                        int sizeX,int sizeY)
    {
        int sum = 0;
        int gridX = location.x()/ m_celResX;
        int gridY = location.y()/ m_celResY;

        int startX = gridX - sizeX/2;
        if(startX < 0) startX += m_dataSizeX;
        int startY = gridY - sizeY/2;
        if(startY < 0) startY += m_dataSizeY;

        for(int y = startY;y<startY + sizeY;y++)
        {
            for(int x = startX;x<startX + sizeX;x++)
            {
                sum+=pMap[(y%m_dataSizeY)*m_dataSizeY+(x%m_dataSizeX)];
            }
        }
        return sum;
    }

    //--------------------
    void InfluenceMap::StampInfluenceGradient(int* pMap,Point3f&
                                              location, int initValue)
```

```
{
    int gridX = location.x()/ m_celResX;
    int gridY = location.y()/ m_celResY;

    int stopDist = fabsf(initValue)*0.75f;//*(m_dataSizeX/32);
    int halfStopDist = stopDist / 2;
    int startX = gridX - halfStopDist;
    if(startX < 0) startX += m_dataSizeX;
    int startY = gridY - halfStopDist;
    if(startY < 0) startY += m_dataSizeY;

    for(int y = startY;y<startY + stopDist;y++)
    {
        for(int x = startX;x<startX + stopDist;x++)
        {
            int value;

            int distX = fabs(x - (startX + halfStopDist));
            int distY = fabs(y - (startY + halfStopDist));

            value = initValue*( halfStopDist -
                        MAX(distX,distY))/halfStopDist;
            pMap[(y%m_dataSizeY)*m_dataSizeY +
                    (x%m_dataSizeX)] += value;
        }
    }
}
```

The `OccupanceInfluenceMap`

Now that you have been given the basic system, we can go forward to the specific implementations. Listing 19.3 and 19.4 give the header and implementation of the class `OccupanceInfluenceMap`, which is a simple IM that tracks population data within the IM of the different game objects.

LISTING 19.3 `OccupanceInfluenceMap` Header

```
class OccupanceInfluenceMap:public InfluenceMap
{
public:
    //constructor/functions
    OccupanceInfluenceMap():InfluenceMap(IM_OCCUPANCE){}
    ~OccupanceInfluenceMap();
```

TeamLRN

```
        virtual void Update(float dt);
        virtual void RegisterGameObj(GameObj* object);
        virtual void RemoveGameObj(GameObj* object);
        virtual void DrawTheInfluence();
    };
```

As Listing 19.4 shows, the `Update()` function is where most of the work is being done. Here, too, you can see two different ways that IM data is handled. In the update method, you can see a call to `memset` that is commented out, above a small chunk of code that *unstamps* the old locations before the new locations are *stamped.* Having the function unstamp the old locations before continuing is akin to a *dirty rectangles* scheme of graphics drawing, in which you only redraw the elements that need it, rather than the entire scene. But you can cancel out this unstamping step with the simple act of commenting out the unstamping block, undoing the movement check in the stamping loop, and uncommenting the memset call. If your game world is small and you have many objects to write, like our test bed, it's much more reasonable to just wipe the IM array and start over. But in the midst of a very large game world, and not very many game objects, or a game with static IM data (such as terrain features or specialized IM flags), you could instead use the dirty rectangles method, so that you don't upset the buffer. Notice also that because this class was written using the unstamp process, the `RemoveGameObj()` method for this class must unstamp the removed object, so no artifacts are left behind.

`RegisterGameObj()` also sets the size of the influence for the object. This probably could have been a more algorithmic process, but for converting the mostly round game objects into square IM shadows for the test bed, this proved to be a quick solution. In the `DrawInfluence()` function, notice that we're drawing a grayscale polygon for each IM array element and that it reaches maximum value at 10 objects within the cell.

LISTING 19.4 `OccupanceInfluenceMap` Implementation of Important Functions

```
//--------------------
void OccupanceInfluenceMap::Update(float dt)
{
    //bail out if nobody to update
    if(m_registeredObjects.size() == 0)
        return;

    //clear out map
//    memset(m_map,0,m_numCels*sizeof(int));
```

```
RegObjectList::iterator listObj;
//unstamp old locations
for(listObj=m_registeredObjects.begin();
    listObj!=m_registeredObjects.end();++listObj)
{
    if((*listObj)->m_pObject->m_position ==
       (*listObj)->m_lastPosition)
        continue;
    if((*listObj)->m_stamped)
        StampInfluenceShape(m_map,(*listObj)->
                            m_lastPosition,(*listObj)->
                            m_objSizeX,(*listObj)->m_objSizeY, -1);
}
    }

    //stamp new locations
    for(listObj=m_registeredObjects.begin();
        listObj!=m_registeredObjects.end();++listObj)
    {
        if((*listObj)->m_pObject->m_position ==
           (*listObj)->m_lastPosition)
            continue;
        StampInfluenceShape(m_map,(*listObj)->m_pObject->
                            m_position,(*listObj)-
>m_objSizeX,(*listObj)->
                            m_objSizeY, 1);
        (*listObj)->m_stamped = true;
        (*listObj)->m_lastPosition = (*listObj)->m_pObject->m_position;
    }
}

//--------------------
void OccupanceInfluenceMap::RemoveGameObj(GameObj* object)
{
    if(m_registeredObjects.size() == 0)
        return;

    RegObjectList::iterator listObj;
    for(listObj=m_registeredObjects.begin();
        listObj!=m_registeredObjects.end();++listObj)
    {
        RegObj* temp = *listObj;
        if((*listObj)->m_pObject == object)
        {
```

TeamLRN

```
                if((*listObj)->m_stamped)
                    StampInfluenceShape(m_map,(*listObj)->
                                        m_lastPosition,(*listObj)->
                                        m_objSizeX,(*listObj)->m_objSizeY, -
1);
                m_registeredObjects.erase(listObj);
                delete temp;
                return;
            }
        }

}

//--------------------
void OccupanceInfluenceMap::RegisterGameObj(GameObj* object)
{
    int sizeX,sizeY;
    if(object->m_size <4)
    {
        sizeX = m_dataSizeX/16;
        sizeY = m_dataSizeY/16;
    }
    else if(object->m_size<11)
    {
        sizeX = m_dataSizeX/10;
        sizeY = m_dataSizeY/10;
    }
    else if(object->m_size<33)
    {
        sizeX = m_dataSizeX/8;
        sizeY = m_dataSizeY/8;
    }
    else if(object->m_size <49)
    {
        sizeX = m_dataSizeX/5;
        sizeY = m_dataSizeX/5;
    }
    else if(object->m_size <65)
    {
        sizeX = m_dataSizeX/4;
        sizeY = m_dataSizeX/4;
    }
    else
    {
        sizeX = m_dataSizeX/3;
```

```
    sizeY = m_dataSizeX/3;
}

//set minimum size of 1 in each direction
sizeX = MAX(1,sizeX);
sizeY = MAX(1,sizeY);

RegObj* temp;
temp = new RegObj;
temp->m_pObject      = object;
temp->m_objSizeX     = sizeX;
temp->m_objSizeY     = sizeY;
temp->m_lastPosition = object->m_position;
temp->m_stamped      = false;
m_registeredObjects.push_back(temp);
}

//--------------------
void OccupanceInfluenceMap::DrawTheInfluence()
{
    glPushMatrix();
    glDisable(GL_LIGHTING);
    glTranslatef(0,0,0);
    glEnable(GL_BLEND);
    glBlendFunc(GL_ONE, GL_ONE);
    for(int i=0;i<m_numCels;i++)
    {
        if(m_map[i])
        {
            int y = i / m_dataSizeY;
            int x = i - y*m_dataSizeY;
            float grayscale = m_map[i]/10.0f;
            glColor3f(grayscale,grayscale,grayscale);
            glBegin(GL_POLYGON);
            glVertex3f(x*m_celResX,          y*m_celResY,          0);
            glVertex3f(x*m_celResX,          y*m_celResY+m_celResY,0);
            glVertex3f(x*m_celResX+m_celResX,y*m_celResY+m_celResY,0);
            glVertex3f(x*m_celResX+m_celResX,y*m_celResY,          0);
            glEnd();
        }
    }
    glDisable(GL_BLEND);
    glEnable(GL_LIGHTING);
    glPopMatrix();
}
```

TeamLRN

When using the system, you instantiate the map, and then initialize it with the grid resolution and the size of the game world you want. You then register any objects you want tracked with the IM. In our test bed, all game objects are being registered with the IM, and another data member of the class `GameObj` has been added, the Boolean `m_influence`, so you can prevent particular game objects from affecting the influence system.

Uses within the Test Bed for an Occupance IM

Figure 19.3 shows a screenshot of the test bed with this system engaged and being drawn for debugging purposes. Use of an occupance IM system within the AIsteroids test bed could help to improve the `Evade` state to steer the player away from heavily congested areas. You could even put a static occupance ring around the extents of the world, and the `Evade` state would then try and keep the ship from staying too near the edges of the world, which tends to get the ship killed by fast moving asteroids that world wrap and catch the ship off guard.

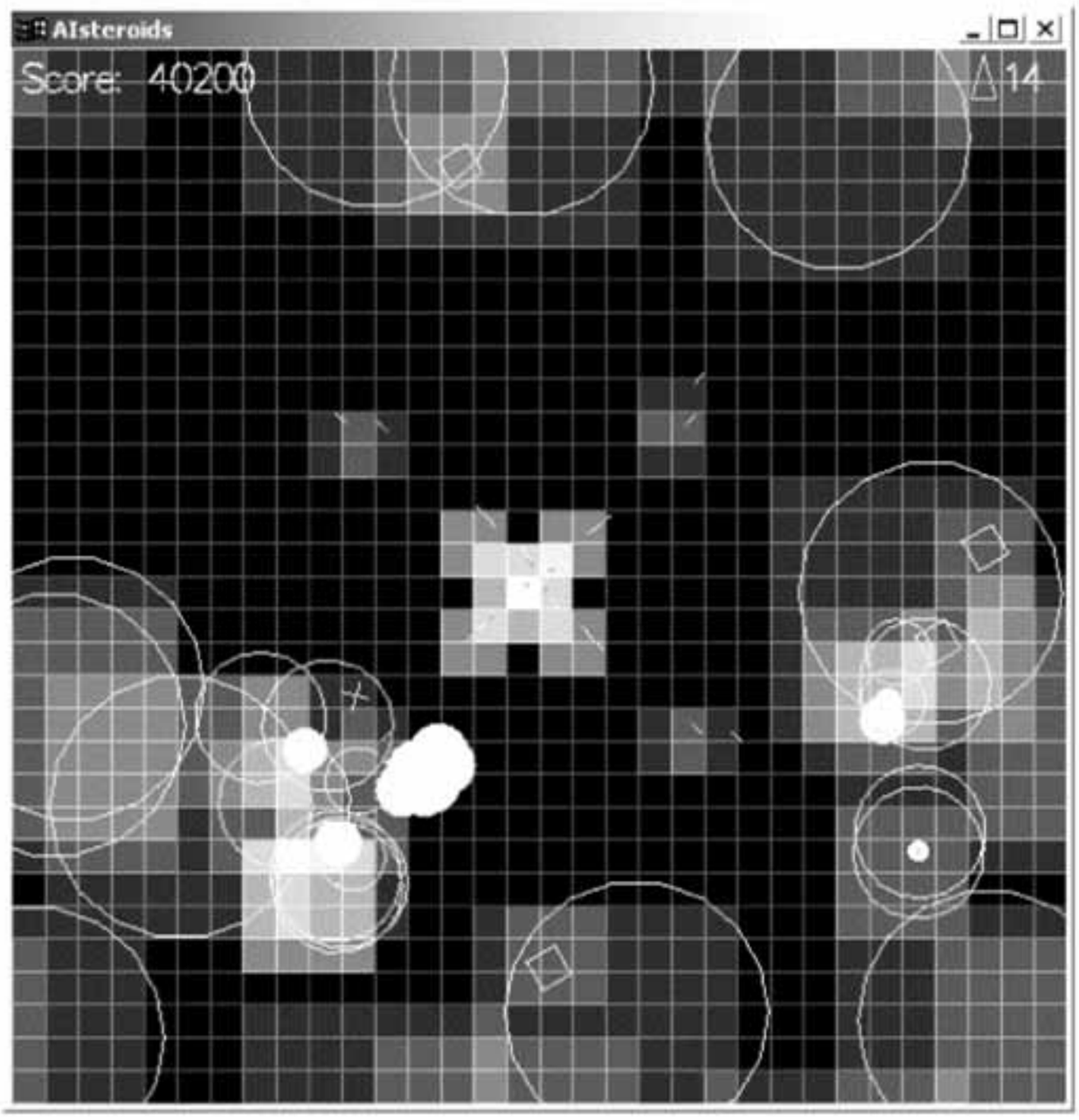


FIGURE 19.3 AIsteroids test bed with an occupance IM engaged.

The `Attack` state could also check the occupance of the map and send that many bullets toward it, so that multiple asteroids at the same location would all be attacked.

The `ControlInfluenceMap`

The next system we will cover tracks control of game areas. Game objects will have a gradient of control written to the map, the magnitude of which is determined by the overall size of the object (except for some special objects, which are given control magnitudes directly). It also assigns the ship and bullets to have positive influence values, and the asteroids to use negative values. All object influence values are added to the map, so the more positive an influence value an IM element contains, the more ships or bullets are inhabiting that element, and the converse holds for negative values and asteroids. Listings 19.5 and 19.6 give the header and implementation of the `ControlInfluenceMap` class.

LISTING 19.5 `ControlInfluenceMap` Header

```
class ControlInfluenceMap:public InfluenceMap
{
public:
    //constructor/functions
    ControlInfluenceMap():InfluenceMap(IM_CONTROL){}
    ~ControlInfluenceMap();
    virtual void Update(float dt);
    virtual void RegisterGameObj(GameObj* object);
    virtual void DrawTheInfluence();

};
```

Notice that within this class, there is no unstamping of influence values, we simply wipe the IM array clean every update. The `ControlInfluenceMap` class uses a type for its registered objects, of which only `OT_FRIENDLY` or `OT_ENEMY` are counted for updates (`OT_BULLET` is just a special type of `OT_FRIENDLY`). It then uses this type to determine whether to write positive or negative control values to the map.

In the `DrawInfluence()` function, we're now drawing a colored polygon for each IM array element, based on the magnitude of control at each location and the sign of that control.

```
//--------------------
```

TeamLRN

LISTING 19.6 ControlInfluenceMap Implementation of Important Functions

```
void ControlInfluenceMap::Update(float dt)
{
    //bail out if nobody to update
    if(m_registeredObjects.size() == 0)
        return;

    //clear out map
    memset(m_map,0,m_numCels*sizeof(int));

    //stamp obj locations
    RegObjectList::iterator listObj;
    for(listObj=m_registeredObjects.begin();
        listObj!=m_registeredObjects.end();++listObj)
    {
        //only care about "control" objects, not miscellaneous
        if((*listObj)->m_objType == OT_MISC)
            continue;

        if((*listObj)->m_objType == OT_FRIENDLY)
            StampInfluenceGradient(m_map,(*listObj)->
                                   m_pObject->m_position, 16);
        else if((*listObj)->m_objType == OT_BULLET)
            StampInfluenceGradient(m_map,(*listObj)->
                                   m_pObject->m_position, 8);
        else
            StampInfluenceGradient(m_map,(*listObj)->m_pObject->
                    m_position, -(((*listObj)->m_pObject->
                                   m_size)/2));
        (*listObj)->m_lastPosition = (*listObj)->m_pObject->
                                                    m_position;
    }
}

//--------------------
void ControlInfluenceMap::RegisterGameObj(GameObj* object)
{
    int sizeX,sizeY;
    sizeX = sizeY = 1;

    RegObj* temp;
    temp = new RegObj;
    temp->m_pObject      = object;
```

```
temp->m_objSizeX     = sizeX;
temp->m_objSizeY     = sizeY;
temp->m_lastPosition = object->m_position;
temp->m_stamped      = false;
if(object->m_type == GameObj::OBJ_SHIP ||
   object->m_type == GameObj::OBJ_SAUCER)
    temp->m_objType = OT_FRIENDLY;
else if(object->m_type == GameObj::OBJ_BULLET)
    temp->m_objType = OT_BULLET;
else if(object->m_type == GameObj::OBJ_ASTEROID)
    temp->m_objType = OT_ENEMY;
else
    temp->m_objType = OT_MISC;
    m_registeredObjects.push_back(temp);
}

//--------------------
void ControlInfluenceMap::DrawTheInfluence()
{
    glPushMatrix();
    glDisable(GL_LIGHTING);
    glTranslatef(0,0,0);
    glEnable(GL_BLEND);
    glBlendFunc(GL_ONE, GL_ONE);
    for(int i=0;i<m_numCels;i++)
    {
        if(m_map[i])
        {
            int y = i / m_dataSizeY;
            int x = i - y*m_dataSizeY;
            float color = m_map[i]/16.0f;
            if(color > 0)
                glColor3f(0,0,color);
            else
                glColor3f(-color,0,0);
            glBegin(GL_POLYGON);
            glVertex3f(x*m_celResX,y*m_celResY,0);
            glVertex3f(x*m_celResX, y*m_celResY+m_celResY,0);
            glVertex3f(x*m_celResX+m_celResX,
                       y*m_celResY+m_celResY,0);
            glVertex3f(x*m_celResX+m_celResX,
                       y*m_celResY, 0);
            glEnd();
        }
}
```

TeamLRN

```
        glDisable(GL_BLEND);
        glEnable(GL_LIGHTING);
        glPopMatrix();
    }
```

Uses within the Test Bed for a Control-Based IM

Figure 19.4 shows a screenshot of the test bed with the control-based system engaged and being drawn for debugging purposes. Tracking control within the AIsteroids test bed allows many improvements.

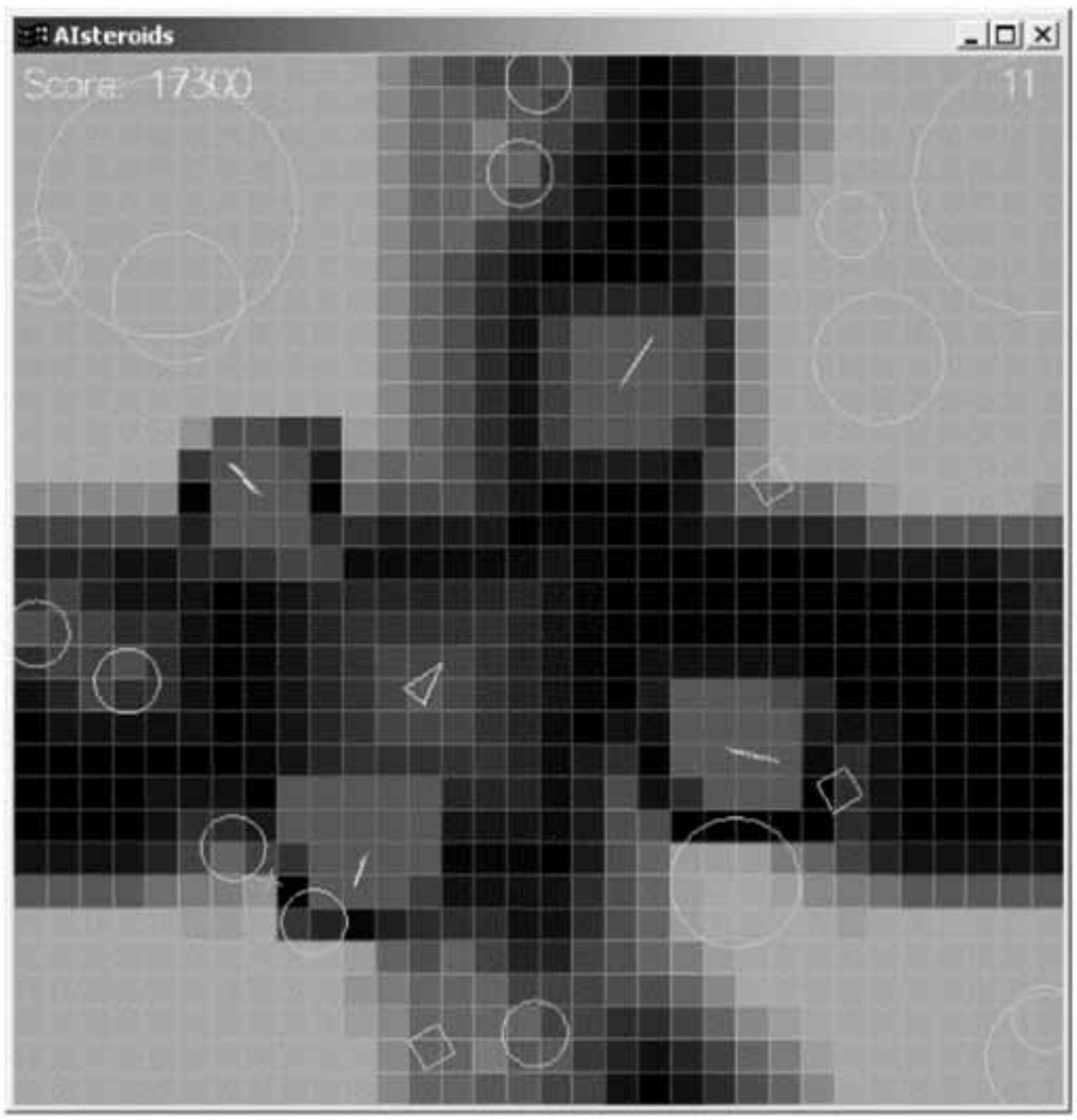


FIGURE 19.4 AIsteroids test bed with an occupance IM engaged.

The `Evade` state could be made much more intelligent by staying within areas of control if possible (providing much more active evasion, rather than the reactive evading that the game currently uses), as well as providing a platform for simple pathfinding to be performed on the IM array to find clear lanes of travel. Evasion

could be improved even more if the control positioning considered velocity, either by perturbing the shape of the control gradient in the direction of travel, or by computing a future position for the object and using that as the position sent to the stamping function. Like the occupance IM, you could put a static ring of "asteroid control" around the extents of the game world, so that the `Evade` state would try to avoid getting near the edges. This type of IM data would give more of a fuzzy effect—the ring of static control could be a smooth gradient, making the avoidance become stronger the closer the ship was to the edge.

The `GetPowerup` state could increase its priority if the closest powerup is within the area of the ship's control. It could also sum the total control of the asteroids and, when low enough, make filling up on powerups a total priority (so that when there are very few asteroids remaining, the ship will max out its shot power and vastly increase its chances of surviving the next wave).

The `BitwiseInfluenceMap`

The last simple IM design we will cover shows how you can use each bit in an array element as a separate Boolean value. This very generic usage of an IM array allows you to custom tailor the information that you are tracking within your IM system. In our test bed application, we will be tracking two main things: object type, and direction of travel. The bottom 8 bits of each array element correspond to the type of object, and bits 9–12 are set if the object is moving in any of the cardinal directions. This is a somewhat arbitrary usage of the system, but it is just for illustration of the method and not an example of what should be done. Listings 19.7 and 19.8 give the header and implementation of the `BitwiseInfluenceMap` class.

LISTING 19.7 `BitwiseInfluenceMap` Header

```
class BitwiseInfluenceMap:public InfluenceMap
{
public:
    //constructor/functions
    BitwiseInfluenceMap():InfluenceMap(IM_BITWISE){}
    ~BitwiseInfluenceMap();
    virtual void Update(float dt);
    virtual void RegisterGameObj(GameObj* object);
    virtual void DrawTheInfluence();
    virtual void StampInfluenceShape(int* pMap,Point3f& location,
                    int sizeX,int sizeY, int value, bool undo = false);
    int GetVelocityDirectionMask(GameObj* object);
    int  GetInfluenceType(int* pMap,Point3f& location);
    int  GetInfluenceDirection(int* pMap,Point3f& location);
 };
```

TeamLRN

This class is much like the others, with the small changes necessary to handle bitwise array access. The stamp function use logical operators, and even though the current implementation doesn't require the use of the unstamping (because the map is zeroed out each update), the stamp function does have the ability to undo stampings.

The debug draw function is a little different for this setup because it draws three polygons for each IM element. The bottom half of the square is the type of object inhabiting the square. The top left quarter is colored if the object is moving up or down, and the top right quarter is colored if the object is moving right or left. A good debugging system for a real game would use more explanatory visual debugging aids than simple colors (such as small status icons or text display, for example), but this will be fine for a test application.

LISTING 19.8 BitwiseInfluenceMap Implementation of Important Functions

```
//--------------------
void BitwiseInfluenceMap::Update(float dt)
{
    //bail out if nobody to update
    if(m_registeredObjects.size() == 0)
        return;

    //clear out map
    memset(m_map,0,m_numCels*sizeof(int));

    //stamp new data
    RegObjectList::iterator listObj;
    for(listObj=m_registeredObjects.begin();
        listObj!=m_registeredObjects.end();++listObj)
    {
        RegObj* temp = *listObj;
        //have to update the bits, since you can
        //change direction continuously
        temp->m_objType = (char)temp->m_objType |
                          GetVelocityDirectionMask(temp->m_pObject);
        StampInfluenceShape(m_map,(*listObj)->m_pObject->
                    m_position,(*listObj)->m_objSizeX,(*listObj)->
                    m_objSizeY,(*listObj)->m_objType);
        (*listObj)->m_stamped = true;
        (*listObj)->m_lastPosition = (*listObj)->m_pObject->
                                                    m_position;
    }
}
```

```
//--------------------
void BitwiseInfluenceMap::RegisterGameObj(GameObj* object)
{
    int sizeX,sizeY;
    if(object->m_size <4)
    {
        sizeX = m_dataSizeX/16;
        sizeY = m_dataSizeY/16;
    }
    else if(object->m_size<11)
    {
        sizeX = m_dataSizeX/10;
        sizeY = m_dataSizeY/10;
    }
    else if(object->m_size<33)
    {
        sizeX = m_dataSizeX/8;
        sizeY = m_dataSizeY/8;
    }
    else if(object->m_size <49)
    {
        sizeX = m_dataSizeX/5;
        sizeY = m_dataSizeX/5;
    }
    else if(object->m_size <65)
    {
        sizeX = m_dataSizeX/4;
        sizeY = m_dataSizeX/4;
    }
    else
    {
        sizeX = m_dataSizeX/3;
        sizeY = m_dataSizeX/3;
    }

    //set minimum size of 1 in each direction
    sizeX = MAX(1,sizeX);
    sizeY = MAX(1,sizeY);


    RegObj* temp;
    temp = new RegObj;
    temp->m_objType = object->m_type;
    temp->m_objType |= GetVelocityDirectionMask(object);
```

TeamLRN

```
        temp->m_pObject      = object;
        temp->m_objSizeX     = sizeX;
        temp->m_objSizeY     = sizeY;
        temp->m_lastPosition = object->m_position;
        temp->m_stamped      = false;
        m_registeredObjects.push_back(temp);
    }

    //--------------------
    int BitwiseInfluenceMap::GetVelocityDirectionMask(GameObj* object)
    {
        int velDir = 0;
        if(object->m_velocity.x() > 0)
        velDir |= DIR_RIGHT;
        else if (object->m_velocity.x() < 0)
        velDir |= DIR_LEFT;
        if(object->m_velocity.y() > 0)
        velDir |= DIR_UP;
        else if (object->m_velocity.y() < 0)
        velDir |= DIR_DOWN;
        return velDir<<8;
    }

    //--------------------
    void BitwiseInfluenceMap::DrawTheInfluence()
    {
        glPushMatrix();
        glDisable(GL_LIGHTING);
        glTranslatef(0,0,0);
        glEnable(GL_BLEND);
        glBlendFunc(GL_ONE, GL_ONE);
        for(int i=0;i<m_numCels;i++)
        {
            if(m_map[i])
            {
                int y = i / m_dataSizeY;
                int x = i - y*m_dataSizeY;
                //determine color for type
                Point3f color(0,0,0);
                for(int index = 0;index<8;index++)
                {
                    int bitset = (m_map[i] & (1<<index));
                    if(bitset)
                        color += colorArray[index];
```

```
}
glColor3f(color.x(),color.y(),color.z());
glBegin(GL_POLYGON);
glVertex3f(x*m_celResX,y*m_celResY,0);
glVertex3f(x*m_celResX,y*m_celResY+m_celResY*0.5f,0);
glVertex3f(x*m_celResX+m_celResX,y*m_celResY+
                                    m_celResY*0.5f,0);
glVertex3f(x*m_celResX+m_celResX,y*m_celResY,0);
glEnd();

color = Point3f(0,0,0);
//get colors for direction
int direction = m_map[i]>>8;
if(direction & DIR_LEFT)
    color = colorArray[COLOR_SILVER];//left
if(direction & DIR_RIGHT)
    color = colorArray[COLOR_PURPLE];//right
glColor3f(color.x(),color.y(),color.z());
glBegin(GL_POLYGON);
glVertex3f(x*m_celResX,y*m_celResY+m_celResY*0.5f,0);
glVertex3f(x*m_celResX,y*m_celResY+m_celResY,0);
glVertex3f(x*m_celResX+m_celResX*0.5f,y*m_celResY+
                                        m_celResY,0);
glVertex3f(x*m_celResX+m_celResX*0.5f,y*m_celResY+
                                    m_celResY*0.5f,0);
glEnd();

color = Point3f(0,0,0);
if(direction & DIR_UP)
    color = colorArray[COLOR_OLIVE];//up
if(direction & DIR_DOWN)
    color = colorArray[COLOR_TEAL];//down

    glColor3f(color.x(),color.y(),color.z());
    glBegin(GL_POLYGON);
    glVertex3f(x*m_celResX+m_celResX*0.5f,y*m_celResY+
                                    m_celResY*0.5f,0);
    glVertex3f(x*m_celResX+m_celResX*0.5f,y*m_celResY+
                                        m_celResY,0);
    glVertex3f(x*m_celResX+m_celResX,y*m_celResY+
                                        m_celResY,0);
    glVertex3f(x*m_celResX+m_celResX,y*m_celResY+
                                    m_celResY*0.5f,0);
    glEnd();
}
```

TeamLRN

```
        }
        glDisable(GL_BLEND);
        glEnable(GL_LIGHTING);
        glPopMatrix();
    }

    //--------------------
    void BitwiseInfluenceMap::StampInfluenceShape(int* pMap,Point3f&
                        location,int sizeX, int sizeY, int value, bool
    undo)
    {
        int gridX = location.x()/ m_celResX;
        int gridY = location.y()/ m_celResY;

        int startX = gridX - sizeX/2;
        if(startX < 0) startX += m_dataSizeX;
        int startY = gridY - sizeY/2;
        if(startY < 0) startY += m_dataSizeY;

        for(int y = startY;y<startY + sizeY;y++)
        {
            for(int x = startX;x<startX + sizeX;x++)
            {
                if(undo)
                    pMap[(y%m_dataSizeY)*m_dataSizeY + (x%m_dataSizeX)]
                        &= ~value;
                else
                    pMap[(y%m_dataSizeY)*m_dataSizeY + (x%m_dataSizeX)]
                        |= value;
            }
        }
    }
    //--------------------
    int BitwiseInfluenceMap::GetInfluenceType(int* pMap,
                                              Point3f& location)
    {
        int gridX = location.x()/ m_celResX;
        int gridY = location.y()/ m_celResY;
        return pMap[gridX,gridY] & 0x0f;
    }

    //--------------------
    int BitwiseInfluenceMap::GetInfluenceDirection(int* pMap,
                                                   Point3f& location)
    {
```

```
        int gridX = location.x()/ m_celResX;
        int gridY = location.y()/ m_celResY;
        return pMap[gridX,gridY] >> 8;
    }
```

Uses within the Test Bed for a Bitwise IM

Figure 19.5 shows the bitwise system up and running in the test bed game. Even using the somewhat arbitrary variables that we tracked in the example, the AI ship would benefit. By checking the IM under approaching asteroids, the ship could use the general direction flags to steer his evasion in better directions. The direction of travel could also help him with asteroids that are soon to wrap, because his evade state could watch for asteroids moving away from him against the opposite edge of the game. Both evading and approaching asteroids could use the general direction of travel logged into the IM as a way to either steer clear efficiently, or to proactively approach along a parallel direction, which is more like how humans play asteroids. Humans rarely fly directly at asteroids, knowing that they will shoot them. Rather, they usually approach from a safe side path of travel, and then turn and shoot. If other variables had been tracked within the bitwise system, any number of different behaviors could be forced from the system.

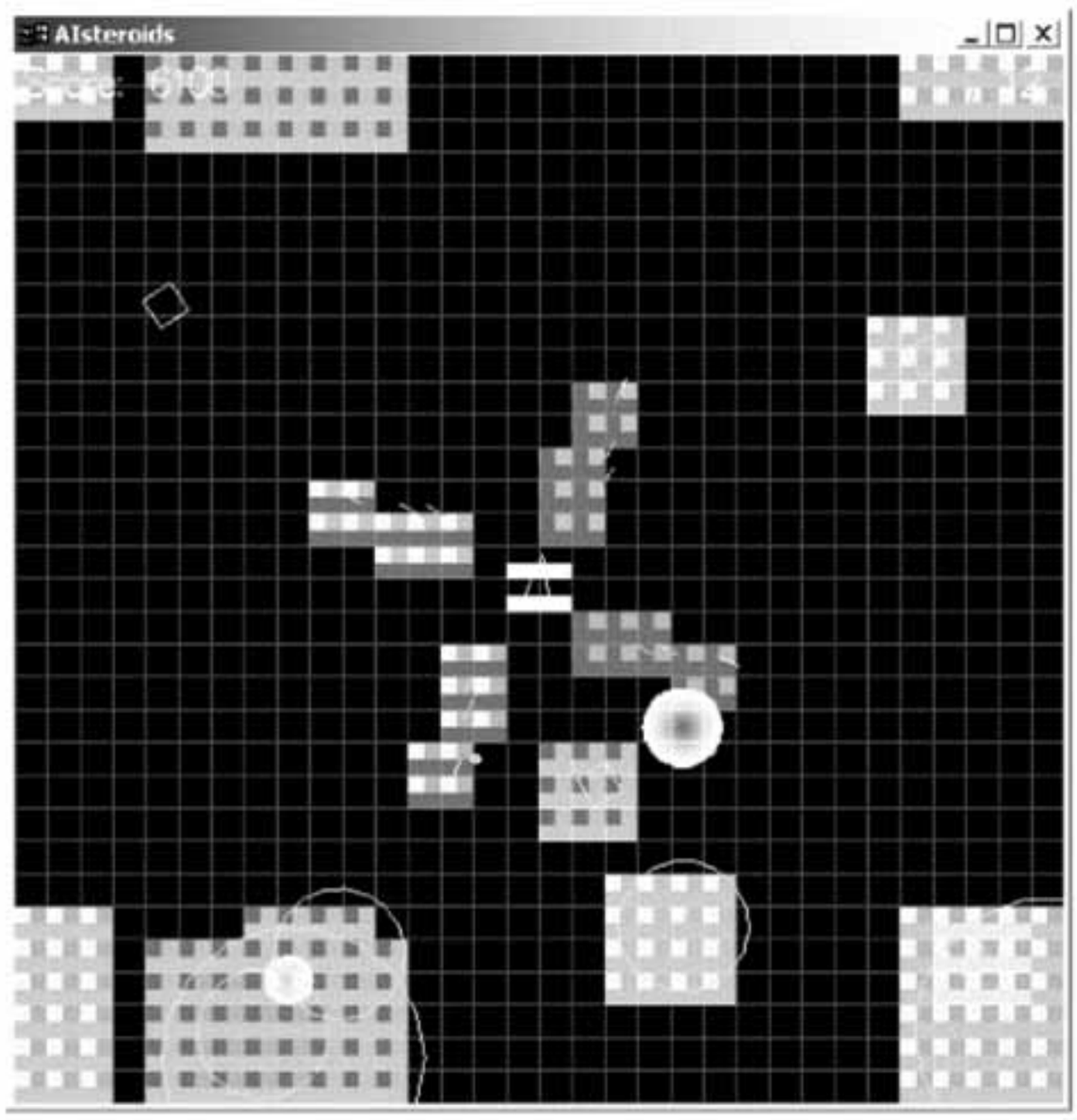


FIGURE 19.5 Alsteroids test bed with a bitwise IM engaged.

TeamLRN

Other Implementations

All these example IM implementations are just a sampling of what can be accomplished with the basic influence paradigm. Some other examples within our test bed might be the following:

- A *danger* measurement (small, slow moving asteroids headed away from us get low numbers; fast, large asteroids on collision courses with the ship get the highest) tracked within the IM would enable the ship to evade much more effectively. This would be similar to the control type of map, but more specialized for evasion.
- If the game had additional powerups, enemy ships, or environmental objects (such as static planetoids or black holes, for example) several more complex states would be needed to handle them within the AI system. An IM would help by further specializing the different pathfinding tasks, providing "control" information for objects that require more complex interactions, and being a platform for terrain analysis.
- If the game world were much larger, or oddly shaped, the IM array would be a good place to do game object searches because a pathfinder could find better targets than the simple "Closest asteroid or powerup" system currently being used. The IM could be tagged with connectivity data so that object searches take the wrapping borders into account, and irregularly shaped worlds would wrap as usual.
- The IM array could keep a short duration (10 seconds might be long enough) record of the occupance data, and maybe the movement direction of the occupance. The ship could use this information to try to keep out of highly traveled areas of the map, or line up next to one of these "routes," and the next time the asteroid comes along, the ship could attack then. This system might only be turned on when there are only a few asteroids left, and only if there are fast-moving asteroids, so that the ship doesn't do a lot of unnecessary movement to chase down a straggler asteroid.
- Using smart terrain within the test bed would require four things:

  1. An extension to the `GameObj` class (`SmartObj`, namely) that includes a new `Update()` function, where it would broadcast a message about the type of object it is, and possibly some other information (position or distance from the main ship, and some kind of priority value). Each `SmartObj` would also need an `Interact()` function that the ship would call to properly *use* each object. Using an object would be context sensitive, so using an asteroid might mean shooting or dodging it, but using a powerup would mean collecting it. Notice that objects do not have to move and could be static structures within the game world, but they still must be represented by an object at some level.

2. A new decision system for the ship—a simple FSM would suffice—that listens to the incoming messages, and is based on the state of the ship, would decide on a primary object to interact with. Based on other factors, the ship might also interact with several others (calling the `Interact()` function on six asteroids might only uncover one asteroid that causes the ship to fire; the other five additively thrust the ship to perform avoidance or lead the ship toward an alternate path), so the ship would keep a list of the objects that it cares about and calls each object's `Interact()` method. If the interaction is fairly complex, then the smart object's `Interact()` call would likely be better off written as an FSM or a script of some sort.
3. To be fully smart, each object would also need to include all the additional code and data necessary for the ship to interact with it. So, if you make a new powerup for the game that requires the ship to dock with it by playing a special animation, the object would have to also have that animation data. Other special case things would include sound effects, powerup effects, any necessary code to incorporate it into the IM system the game is using (if any), and so on.
4. Game code that dealt with some object interactions would need to be removed. So, the `GetPowerup()` code needs to be removed from the ship class, and most of the behavior code will be moved from the ship's states and into the smart object's interaction functions. This might seem a bit backward, but notice that when this process is finished, adding new powerups, weird space anomalies, or enemies would become a process of just setting up the smart object, and letting it loose into the game.

PROS OF LOCATION-BASED INFORMATION SYSTEMS

LBI systems are a generic interface for games, and as such, almost any specialized location data requirement can be built within the system. LBIs are intuitive and easy to program and scale well to large and small problems. Debugging LBI systems is generally very simple; employing visual feedback is straightforward (as the demonstration implementations show).

Generally, IM systems tend to simplify the perception search space by lowering the resolution of the data that the AI needs to consider when making decisions. IM systems also represent a kind of shared stored knowledge base about the world that AI characters can use to act more intelligently. Thus, even though the ship hasn't personally made every little calculation about the asteroids in the map, it can consult the IM for a wealth of info about each asteroid and make far smarter decisions in less time.

TeamLRN

CONS OF LOCATION-BASED INFORMATION SYSTEMS

LBI systems do have trade-offs, however. IM arrays tend to be size expensive, especially in games with large data requirements, large world sizes, and high array resolutions. You need to be smart in implementing your IM system, using multiple levels of resolution to limit data size and using local, relocatable higher resolution IMs for more detailed work instead of an all-encompassing IM.

Terrain analysis can be computationally very expensive because of the many searches through the array that need to be performed to glean all the necessary information. Almost any pattern-matching algorithm is going to be costly and error prone—there is a reason why computer handwriting recognition is still in its infancy.

EXTENSIONS TO THE PARADIGM

The LBI systems implemented in this chapter are very rudimentary. The only limits to the different ways that you can use these basic principles in your game are the type of game you are working on, the data size you are allotted for your IM, and the CPU time you can spend searching the map for useful patterns that you can exploit. Almost any genre can find a suitable use for these techniques. FTPS games could use them for king-of-the-hill-style matches, to track control. RTS games are the biggest potential users of these methods, with the possibility for many different areas of the game using a shared IM for a variety of tasks. Even genres like classic adventure games could use LBI methods; you could keep track of where the user is clicking with the mouse, and if he seems to be clicking everywhere, or the same places over and over again, he's probably stuck or doesn't understand some puzzle element and could use some contextual help.

OPTIMIZATIONS

IMs deal with a large contiguous block of memory, so writing and reading from IMs becomes a problem similar to using early software graphics engines. You are almost "blitting" influence data to the array, and reading values back out again. Thus, many of the optimizations that people used for early graphics also apply to IM optimizing, which is where the *dirty rectangles* analogy came from earlier when we were discussing the occupance IM implementation. Instead of drawing every element into the array every update, you draw the small areas where objects moved. Other similar optimizations might include finding out the size of the data bus on the machine you're developing for and ensuring that the size of the usual data

element you write or read out of it fits within the bus, to ensure fast data transfer, as well as better cache usage.

The other optimizations talked about during the rest of the chapter, such as levels of IMs with increasing resolutions (level of detail IM arrays), as well as local IMs that use much more detail, will save you both memory and computation time because you only apply as many CPU resources to IM tasks as you need.

TA functionality needs to be optimized on a per case basis because TA tasks differ so greatly in terms of many factors: what they are trying to accomplish, the scope of their search within the IM array, the kinds of patterns they are seeking to find, and the frequency that the given TA task must be updated.

DESIGN CONSIDERATIONS

LBI systems are usually found in the more AI heavy games, such as RTS, deathmatch FTPS, and RPGs, because they require a level of intelligence that the more action-oriented genres do not need from their AI opponents. LBI have an open architecture for location specific information and allow proven search methods to be applied to this data in a central location.

Types of Solutions

LBI can be used to solve both tactical and strategic types of AI problems. Tactically, an IM can help guide pathfinding and dynamic obstacle avoidance. It can provide a character with secondary behavior cues so that he looks more engaged in the world. At the strategic level, TA can provide an AI system with pattern matching necessary to really use the terrain, and plan large-scale battles or building whole towns. Objects within a smart terrain system are more tactical because they don't usually add much to the strategic intelligence of an AI character. You don't see *Sims* characters planning very far ahead to satisfy a need. They are mostly just roaming around, toward the next object that will help them out. Yes, the characters go to work to get money, but that's more of a game-state mechanism rather than a plan in which a *Sims* character thinks about wanting something and then goes off to earn the money to buy it. This is because each object is an island onto itself and doesn't know about any of the other objects in the world except for the thing it has been programmed to interact with.

Agent Reactivity

LBI is a secondary system, so reactivity is more a question of what primary AI technique is being used. LBI systems can help to make a character much more proactive in his reactions, however, so that should be considered.

TeamLRN

System Realism

IM-enabled games are not necessarily more realistic (people keep a large local memory to themselves, instead of tagging locations with memory or perception data, and the microwave certainly doesn't broadcast that it will feed you), but with the right level of IMs and TA, a game could make much more realistic, humanlike decisions. Using a central map of information is much more like how humans approach these kinds of problems, rather than just knowing everything like a computer opponent and cheating. Smart terrain objects aren't really a realistic way of modeling things, but they do allow a much richer environment because new objects can be added so readily, and they do abstract objects and environmental elements into categories, which is a realistic human behavior.

Genre and Platform

The genres listed earlier—RTS, deathmatch FTPS, and RPGs—are the usual suspects for IM systems and TA. Smart terrain has only found its way into a small number of titles so far, but it is much more general, relative to genre. Any game that needs a no-nonsense level of interaction between arbitrary objects and the environment could benefit from a smart objects system—both for ease of writing the primary AI decision system and from an expandability point of view. The only real platform concern with LBI is the memory requirements of an IM array, but with proper forethought and optimizations, this can be overcome.

Development Limitations

Development limitations are not really a concern for LBI systems. LBI information actually might help debugging of a game, so that isn't an issue. IMs are another way to decouple AI characters from the rest of the code (by providing a central data location for them to search in, instead of making gamewide code calls) and help the AI system become more modular. Smart terrain objects, by their nature, allow modular implementation, so they tend to be debuggable and scaleable.

Entertainment Limitations

Tuning difficulty settings, balancing specific behaviors, and other entertainment concerns are generally independent of LBI system use, so are not usually a problem.

SUMMARY

Location-based information systems can provide a variety of decision-making paradigms with additional flexibility in dealing with location-specific data, as well

as decoupling the AI characters from the rest of the game by providing them with a central data location or encapsulating logic and data for interactions.

- The three main categories of LBI systems covered were influence maps, smart terrain, and terrain analysis.
- IMs are a generic data structure usually represented as a 2D grid of data elements laid over the game world. The data contained within, or even the structure of the data within, is completely arbitrary to the method.
- Smart terrain is a technique whereby logic and behavior data showing how to use various world features and game objects are stored in the objects themselves. This provides a modular and expandable world for the game characters to live in, but limits the amount of interaction that can be done with any one element.
- Terrain analysis is a family of methods that can be performed on terrain data to search for usable patterns that can lead to better strategic decisions.
- The demonstration implementation was done in four parts: the basic IM, and the three various versions: occupance, control, and bitwise IMs.
- LBI methods within a game can be implemented in many other ways than were demonstrated in this chapter.
- The pros of LBI systems include ease of implementation and debugging, generic interface, and centralizing AI data.
- The cons of LBI systems include large memory requirements for IM arrays, and possible high computation costs for heavy TA.
- IM functions can sometimes be optimized along the lines of early graphics routines because you are writing and reading data from large contiguous arrays.

TeamLRN

Part IV Advanced AI Engine Techniques

Whereas Part III covered the more common techniques used within commercial game AI programming, Part IV will delve into the more exotic methods that are starting to show up in games, fresh from the academic world.

First, we delve into two specific techniques: genetic algorithms and neural nets. As in Part III, they will be discussed, and skeletal code will be implemented within the AIsteroids test bed, with a full discussion of the technique, its pros and cons, extensions, and optimizations.

After that, Chapter 22, "Other Techniques of Note," describes many other techniques, but does not implement them within the test bed. These are the fringe methods that are just beginning to find roots growing into game AI. However, these other methods *are* discussed fully, including the pros and cons of each method. There are also plenty of resources on these techniques on the CD-ROM in the form of Web links, source and demonstration code, and references to books and papers.

TeamLRN

20 Genetic Algorithms

Some times, we come up against AI problems that defy solving, either because of computational difficulty, or simply because there isn't enough time. There are too many possible responses, or too many incoming variables to consider. It's always *possible* that a solution could be found, but only after many, perhaps hundreds or thousands, of programming iterations involving manually trying different avenues in a hunt for the best algorithm. As an example, consider having to tune the performance parameters for the physics simulation used by each car in *Gran Turismo 4*. With more than 500 vehicles and dozens of tweakable settings for each intricate piece of a car's handling system being simulated, this would truly be a daunting task for any company to accomplish (at least, within any reasonable time and monetary budget), especially if the goal was to accurately depict the real-life performance of each car.

GENETIC ALGORITHMS OVERVIEW

In this chapter, we will cover an AI technique called genetic algorithms (or GAs) that take lessons learned from evolutionary science to try and find novel solutions to these kinds of problems. We will cover the basic method by discussing the natural model, and then show how the model can be applied to our game problems. A basic, general case GA class will then be put forth and implemented into the AIsteroids test bed for illustration.

Evolution in Nature

GA techniques try to use the principle of evolution, normally found in natural systems, to search for solutions to algorithmic problems. The process in nature works roughly like this:

TeamLRN

- To survive as a species, all living creatures need to be able to reproduce. Reproduction is (heavily simplified) simply executing the encoded rules necessary to build an organism. These rules are stored in strings of DNA (made up of proteins) called *chromosomes,* which are found in every cell that makes up a living being.
- Chromosomes are in turn made up of small, modular sequences called *genes,* which are various permutations of the four basic proteins: thymine, adenine, cytosine, and guanine (or T, A, C, and G, respectively). Each gene holds information about the "settings," or *alleles,* of a particular trait (or number of traits because each gene is usually linked to more than one trait within a body).
- When two parents reproduce, their DNA is split, in that half the DNA of the child comes from one parent, and half from the other. This is called *crossover* or *genetic recombination.*
- Genetic crossover results in a new mixture of genetic traits that are passed on to the offspring. If this new mixture of traits is good, the child will have a full life and be able to reproduce as well, again passing on at least half of its traits to future generations. If, however, the child inherits weak or even bad traits, it may not survive long, or even be able to reproduce at all (either because of biological reasons, such as infertility, or social reasons, in that it is not a desirable mate). Over many generations, the trend of organisms with a better mix of traits being more likely to reproduce, and creatures with bad gene mixes being starved out of the overall pool (and thus removing their genes), leads societies of creatures to evolve toward the genetically superior version of their species. The quality measure of any one creature's mix of traits is called its *fitness,* and the higher the fitness value, the better that creature is at applying its traits to the world, both in performance and reproductively. In human terms, a highly successful man might still lose out in the race because his all-encompassing drive to perform well in the working world might make him unavailable to have children and pass on his genes. Thus, both areas (performance and reproduction) need to be expressed for the traits to move forward.
- Occasionally, a flaw happens in this system (although this is up for debate whether this "flaw" process might be an integral component built into the system). A gene within a child organism somehow changed so that it is completely new and cannot be traced to one of its parents. The gene is replicated wrongly, a chemical imbalance occurs during fertilization, or any number of things; we do not know all the causes at this time. When this happens, it is called a *mutation,* and the results are that the allele of the particular gene are now random, with random effects in the organism. In most cases, this results in negative traits. A bird is born with wings that are too short to fly, a tree sloth is born with a large brown spot on its head (and thus no other sloths will mate with it), or a

monkey can hear very high wavelength sounds and goes insane from all the nighttime chatter.

- But sometimes, this mutation results in the child having traits that make it *better* at performing within its environment, or in some way make it more likely to reproduce. When this happens, this mutated gene is (we hope) passed on to other generations through reproduction, and on and on.
- Thus, the "survival of the fittest" paradigm gradually changes the set of traits (called a *genome*) that the species contains on average toward the ideal set, which represents the most adapted genes for the particular creature in its current environment conditions.

Evolution in Games

So what does all this give our game AI? It turns out that this evolutionary algorithm can be implemented within the confines of our game worlds and be used to tune behaviors, parameters, and the like for areas of gameplay that would take far too long to iteratively hand tune. The process can be thought of as "evolutionary search," in that we are still searching across the field of all the possible solutions to a given game problem, but we are going to use a method of searching that is likened to the process of evolution through genetic fitness.

The method is split into two, very unequal halves: evolving a solution, and using the solution. Typically, using the information gleaned from a genetic algorithm in the final game is a black box operation. This is a magic box that makes the particular problem behavior act in the best way possible (or in the best way possible you were able to find). Evolving the solution is thus almost all the work, and this process is usually performed during the production of the game. Very few games actually go out the door with the evolving parts of their genetic algorithms still turned on. Some notable exceptions to this, where learning was one of the central tenets of the game, include the *Creatures* series, and *Black & White.* But the learning components of even these games are very closely monitored; the traits and behaviors that are being influenced by the learning elements are constructed in such a way that they constrain the learning or genetic elements, so that the learning or evolution is as tightly controlled as can be. The nature of these games also fosters unpredictability in the AI character's actions, so some degree of leeway is granted for stupid or inappropriate behavior.

GAs have tended to be computationally expensive (this process can take a good deal of time because you are forced to run many, many generations on a large population of possibilities), which made using them a costly choice in the past, which is one of the reasons that the evolutionary work is mostly done offline. Increasing performance of the average computer has allowed these methods to become more mainstream.

TeamLRN

GAs belong to the class of *stochastic search* methods (others in this family include simulated annealing and threshold acceptance), which means they rely on an element of random chance or probability for directing the search. Hence, numerous search iterations are required (because you never know if the random element of the search has led you astray from the best solution path), but you are much less likely to get stuck around a particular "solution" (some applications may have many genomes that provide good results, but might not be the best possible; this is referred to as finding *local maximums* rather than *global maximums*, and having an element of randomness tends to "jump" the search out of the trap of local plateaus).

Unlike some searching systems, GAs separate their algorithm from the problem representation (the algorithm works for vastly different data structures), which allows them to easily find solutions in systems of mixed variable types (having both discrete as well as continuous values). Although the most common technique for data representation is a string of bits, any data structure you want (including arrays, trees, etc.) can be used as long as each individual can encode a complete solution, and genetic operators can be constructed for your data structure.

One thing to consider about GAs is that they do not guarantee either performance, or success. In fact, a GA can, and sometimes will, perform in the worst possible ways. Such is the price for throwing a randomness element into your algorithm. You will possibly need to tweak the structure of your genes, or even the GA's settings and operators if the system doesn't deliver the kind of behaviors that you were looking for. Even then, you might discover that GA methods just aren't suitable for your particular problem.

BASIC GENETIC METHOD

The basic algorithm for using GAs to find a solution to an AI problem can be broken down into three steps: initialize, evaluate, and generate.

Initialize a Starting Population of Individuals

These beings can be generated randomly, or seeded with promising individuals given some specific knowledge concerning the problem domain. The size of the initial population is somewhat arbitrary and mostly depends on experimentation and resources, how much time you have to devote to the process, and what seems to work.

Evaluate Each Individual's Success within the Problem Space

Each individual is then subjected to evaluation, by running a special *fitness* function, which returns a number value (or possibly a vector) representing the overall performance of this individual. GAs are so expensive because of the time necessary to calculate fitness. If you can look at an individual genome, and algorithmically

calculate its fitness, then this process is quite fast. But especially in the world of gaming, each being in the population must be run through the game loop for some period to determine its fitness score. A fitness test that requires each individual to play the game for 5 minutes, given a population of 100, would thus require 8.33 hours per generation, and a typical GA can take thousands, if not tens of thousands of generations to find any really useful solutions. Making your game simulation time scaleable can obviously speed this process up. But a sped-up world is not the same as the real game (for example, physics checks might miss collisions because of large deltas in time between frames), so your GA might learn things specific to a sped-up world, and not be as effective real time. Another technique involves using a GA to equate a specific *part* of the game decision-making process and have this particular part use a more straightforward algorithmic or time scalable fitness function.

Generate New Individuals Using Reproduction

Once all members of the population have had their fitness calculated, a number of these individuals are selected for breeding. *Selection* is another important part of the genetic process. If you only select the very best performers, you may converge on a local maximum too quickly because you have excluded too many genes from the pool. If you select too randomly, you may never find a good solution because you can cause too much random jumping. Several methods of selection will be covered later in this chapter. Once the parents have all been selected, they are bred to create the next group of possible candidates. The next generation is spawned using three common methods: *crossover* (or sexual reproduction), *mutation* (or genetic variation), and what is called *elitism* (which is just taking the most fit, or elite, beings from the last generation and carrying them over into the next—which isn't exactly breeding, it's more like cloning). The specific mix of these three methods, as well as the exact operator to perform each method with, is again up to experimentation and domain-specific knowledge. Many of these differing operators will be discussed in this chapter.

REPRESENTING THE PROBLEM

GAs are commonly written in the language of the thing being copied, evolution. We will design a way of representing our problem in a genetically compatible way and create iterative operators for dealing with this abstract representation.

The Gene and Genome

First, determine the structure of the genes inherent in your problem. What, and how many specific traits are you seeking values for? Are the alleles of these traits

TeamLRN

binary (on/off), or are they real numbers, and if so what are their ranges? Do any of these traits depend on each other?

The importance of these determinations cannot be overstressed. When you create a gene or genome structure for your GA problem, you are essentially defining the state space that the GA will search in, as well as formalizing the language in how you will receive your answer. Answers will be at the same resolution as the genes themselves. So, if you only encode the four cardinal directions as alleles for direction, your character will only move in those four directions in the solution. But fear not, GAs work just as well with analog allele states as they do with discrete values, just at the cost of larger search spaces.

Herein lies the tradeoff. The larger the search space, the more likely your GA is going to find good, possibly surprisingly good, behavior. The larger search space creates two disadvantages, however: it is going to take longer to find that solution, and the there is a greater chance that the GA might find an exploit in your logic. This means that you've set up your parameters in such a way that the GA finds a solution that maximizes your problem with behavior that doesn't follow the spirit of the game, or are just nonhuman enough that they are unwanted. A somewhat unconnected example is the animation system used by the animators in the *Lord of the Rings* movies. Because of the massive battle scenes, which would have been impossible if each combatant was hand animated, they used an AI system to control the warriors. However, because of the common sense default settings of the AI, when creating the massive battle between the humans and elves and the overwhelming forces of the orcs, all the AI-controlled humans and elves simply turned tail and ran into the woods when the battle started. The default settings needed a bit of tweaking to reflect the unwavering morale and sense of duty that these soldiers felt in the book. GAs are notorious for finding loopholes in your calculations, and can find novel, yet unusable solutions to problems. Giving the GA too large of a search space to look within sometimes can exacerbate these kinds of issues.

One of the most common ways of encoding genes is as strings of bits. Consider a game world for *Pac-Man*. The main character has only four choices for his actions: Move Up, Move Down, Move Right, and Move Left. He can do any one action per game loop. He knows that the path in front of him in the direction he's currently moving has some "state" (defined as the containment of the path; does it contains regular ghosts, blue ghosts, dots to clear, or a wall). So, we could evolve a GA that would get *Pac-Man* to do the right thing when confronted with all the different permutations of world state that he encounters. Each gene would be two bits (representing the four actions he can perform). His genome would then be a string of genes that corresponded as proper responses to the different world states.

Another type of gene that comes up often is an order or content-sensitive gene. A commonly used example of this is the classic Traveling Salesman Problem (TSP).

Given a number of cities that the salesman has on his route (Figure 20.1 shows a typical setup), in which order should he visit them so that he goes to each city only once and also travels the shortest distance? The genome structure for this problem is obviously a list of cities. But unlike the string of actions in the earlier *Pac-Man* example, each gene would have to be unique for the genome to be valid because you can only visit each city once. We could technically define our *Pac-Man* genetic solution using a TSP-style structure, solving each board for optimal travel so that *Pac-Man* would clear the dots in the shortest distance. There's a problem, though. The ghosts are going to be in our way at some point, and *Pac-Man* might have to take a path (to escape) that he's already cleared. So, maybe we should leave *Pac-Man* with the first implementation type. In fact, you should know that *Pac-Man* is really only being used in these examples because of the simplicity of the game world, as well as the almost universal familiarity that people have with the game. In reality, an AI system designed to play *Pac-Man* should probably use a more standard method, rather than GAs. In general, GAs are good for problem arenas in which you really just can't formulate a good way of determining solutions. The number of calculations that would need to be done to simulate the richness of the GA search method would leave a heuristic system either choking on the sheer number of computations, or mired in a sea of tunable values that require programmer time in which to balance. We will cover this more in the section on cons of the system, later in the chapter.

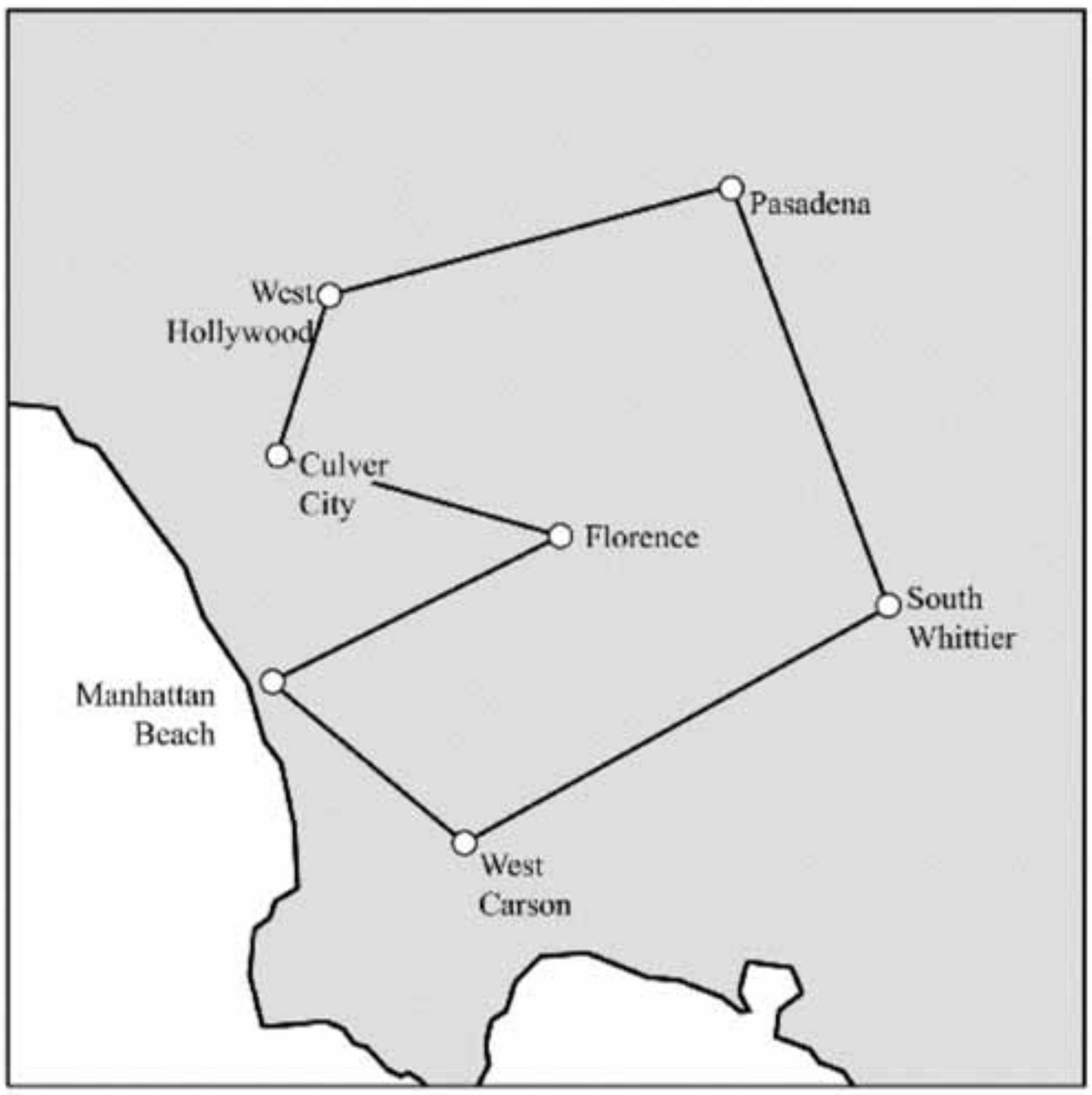


FIGURE 20.1 The traveling salesman problem.

TeamLRN

Other types of structures that have been used as genomes within the general GA method include arrays and trees, both of which can easily be handled with the system, and both have had standard genetic operators written for them. All these examples can use genomes with both a fixed number of genes, and with a variable number of genes. The only real requirements of any genome representation are that it has the ability to encode a successful set of rules for the problem and that genetic operators can be written to manipulate the structure without breaking the system, or mangling the meaning of the genome, any individual gene, or allele.

The Fitness Function

Once we have determined the format of our genes and genomes, we then have to figure out how we're going to score each genome for performance within the world. The fitness function is domain specific. In *Pac-Man,* a fitness function might consider total score, speed of clearing the level, length of survival time, and number of blue ghosts chomped. The function could take these elements into account with different bias coefficients (for example, so that survival is most important, then score, and then speed of level). The addition of "number of blue ghosts chomped" might be double dipping because you are already valuing score (which is what you get when you eat blue ghosts), as well as survival (chomping blue ghosts clears your way for easier survival). How you determine these coefficients would fundamentally mold the type of *Pac-Man* player you are trying to evolve: to what extent does your player value the best score, the fastest time, and sheer survival.

Your fitness function is really the essence of what your GA is trying to optimize its solution for, so you must give careful consideration in its design. Your fitness function is the heuristic that you use to direct evolution within your search space. Too many parameters, and the behavior of your GA is going to be diluted, as well as require many more generations of genetic manipulation to find a good solution. But too few parameters, and your GA is going to discard too much "unnecessary" genetic material from its population in favor of only those genes that maximize its limited fitness model.

Once a basic fitness equation has been designed, and we run our function on all the members of the current population, the fitness data must then usually be scaled, to prevent *premature convergence (PMC)* and *stagnation.* PMC occurs when exceptional individuals are born in an early generation, in a GA with a relatively small population, causing these early supermen's genes to quickly spread to a large portion of the population as they dominate selection. Stagnation occurs more toward the end of the process, when many individuals have similar, high numerical fitness. In this case, the differences between individuals are minimized (at least to the selection process, which will be discussed during the "Reproduction" section later in this chapter), which is not what we want because there is no longer very high

selection pressure. In effect, scaling the fitness values brings out the various (and extremely small) advantages caused by the combinations of genes within the game entities. Some of the common ways of scaling the data include the following:

Sigma truncation. The scaled fitness value (`F`) is equated `F` = `F` – (`F^` – `c*sigma`). `F^` is the average fitness, sigma is the population standard deviation, and c is a reasonable multiplier (usually between 1 and 3). Negative results are set to 0. You are basically scaling everyone's fitness using the standard deviation of the entire group, which means that there is more scaling when the group is more wildly different (hence, the beginning of the simulation) and will gradually take less effect as convergence gets under way and fitness scores start to become similar.

Rank scaling. Rank scaling replaces the fitness score with its position in the sorted order of the fitness scores. So, whoever had the lowest score now becomes 1, the second lowest score becomes 2, and the highest fitness scorer is the size of the population. Easily eliminating the chance of PC, rank scaling does the opposite: it makes a GA take much longer to converge.

Sharing scaling. A method that tries to encourage genetic variation, this scales down individual fitness scores that are very similar to each other. Essentially, the number of genes that different genomes share is recorded, along with how many genes they share. Genomes are then grouped by how many shared genes they have (e.g., all those with five shared genes are in group 5). Finally, the fitness score of each genome is scaled by the number of other genomes in its sharing group.

Reproduction

We have a population of individuals, and they have been evaluated by the fitness function. Now we must build an offspring generation, using the knowledge we've gained from this run. Two main types of reproductive cycles are common. The first is *generational* reproduction, referring to the process of using the last generation as a tool to create the next, either by copying directly or through genetic crossover and mutation, completely replacing the original generation. The second is called *steady state* reproduction, wherein a few new individuals that are created through crossover or mutation replace specific individuals each generation, but the main body of the population remains unchanged. Who is replaced in steady state implementations is another question (most common is to replace the worst, but other schemes involve replacing randomly, most similar, or parents).

If we directly copy individuals from this generation to the next, this is called *elitism,* and it helps ensure that whatever selection routine we use doesn't accidentally miss the best beings in any given population. Elitism does the opposite of

TeamLRN

fitness scaling: it lessens genetic diversity and speeds up convergence, so care must be taken in its use. Too much elitism, and you will find local maximum solutions instead of global ones. Steady state implementations do not require additional elitism because the method is defined by genetic carryover.

Other selection functions include the following:

Roulette wheel selection. The random chance of a genome being selected is proportional to its fitness score. If you have the highest score, you also have the largest chance of being selected (you'll have the largest slice of the roulette wheel). Notice that this selection doesn't take you out of the pool, so a high fitness individual may be selected multiple times. Notice, too, that it's still a random chance. Thus, the fittest individual is not *guaranteed* a place in the next generation, and this is the reason that elitism is a common practice in GA genome selection.

Stochastic universal selection. This is a long-winded name for roulette wheel selection with a twist. The same roulette wheel is constructed, but now you don't have to spin. You take the number of individuals you want to select (n), and select the owner of the roulette slice at 1/n increments along the wheel. So, for 10 individuals, you would select the owner of the roulette slice pointed to each 1/10th of the wheel. This has the advantage over regular roulette wheel selection by keeping the spread of the fitness values chosen low and, thus, keeping genetic diversity high.

Tournament selection. In this technique, a number of individuals are randomly drawn from the pool, and the highest scorer makes it to the next generation. Everybody goes back into the pool, and this is repeated for however many you need to make a new generation.

As individuals are being selected, you check each incoming pair for *crossover*, or sexual gene blending. The simulation keeps a crossover rate number, which is usually around 0.7f (your number may vary as you see fit;). Generate a random number between 0 and 1; if the number is less than the crossover rate, you apply a crossover operator to the two individuals, creating two offspring. Otherwise, they become unaffected offspring. Which crossover operator you use depends on a number of things: the type of variables and structure your genomes are using, and a healthy dose of experimentation.

Some of the common binary variable crossover operators are the following (see Figure 20.2 for visual descriptions):

Single point crossover. A position is randomly chosen somewhere along the length of the genome. Swapping all the genes after this position among the parents forms the offspring.

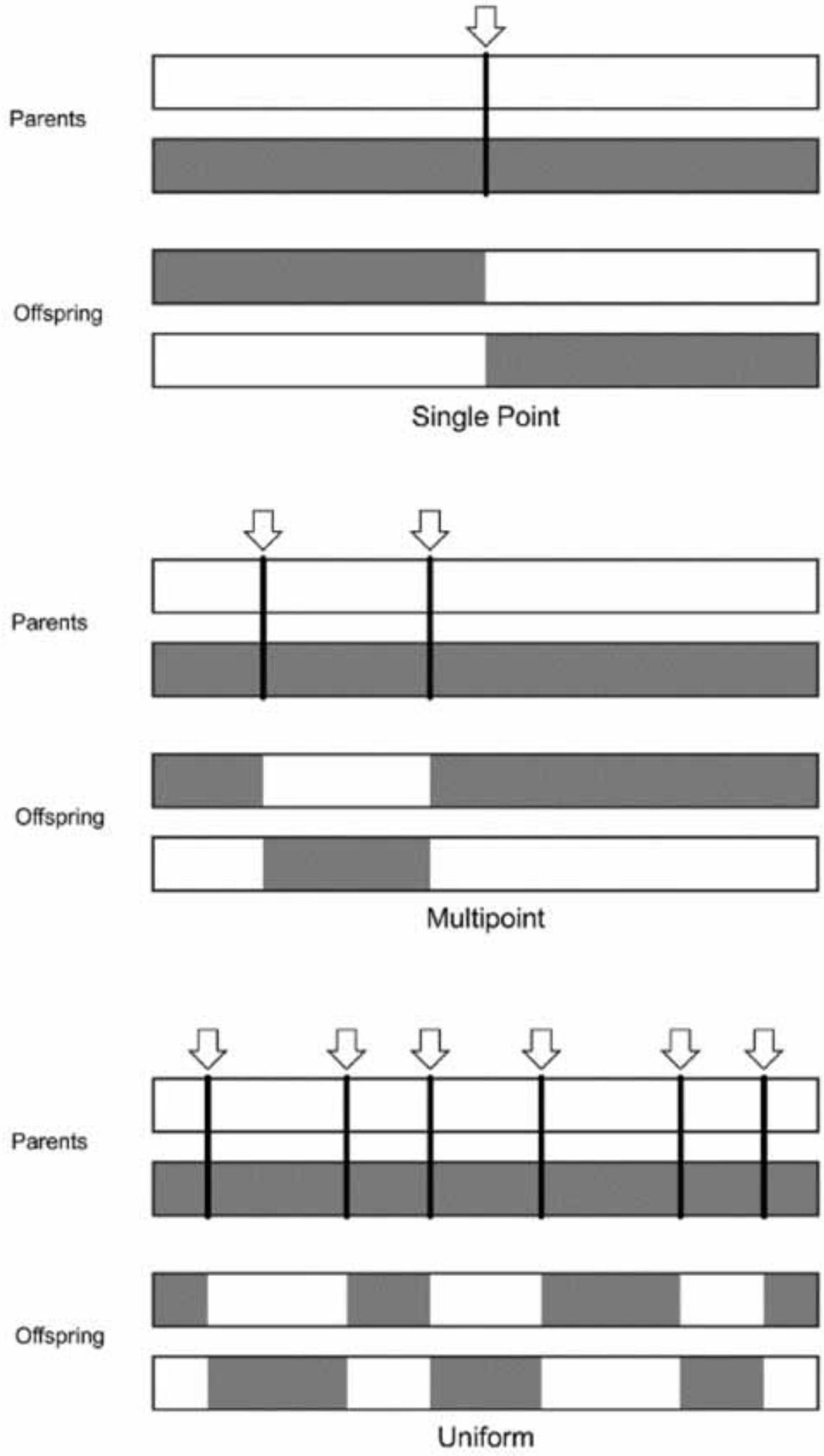


FIGURE 20.2 Binary variable crossover operators.

Multipoint crossover. Same as single point, except that two points are selected, and all the genes between the two points are swapped.

Uniform crossover. What could be called "every point" crossover; this method performs the mutation check with every gene, and swaps it with the other parent if it passes.

TeamLRN

Common continuous value variable crossover operators are the following (see Figure 20.3):

Discrete crossover. Swaps the variable values between individuals.

Intermediate crossover. Determines the offspring's variable values as being around and between the parent's values. The offspring formula is `offspring = parent1val + Scale*(parent2 val – parent1 val)`, where `Scale` is a scaling factor chosen randomly for each value over the interval `(-d,1+d)`. Normal intermediate crossover uses `d=0`, but if you want to extend the children outside the area of their parents, you can use a `d>0`.

Line crossover. The same as intermediate crossover, but all variables use the same `Scale` scaling factor.

FIGURE 20.3 Continuous variable crossover operators.

Order-specific operators include the following (see Figure 20.4):

Partially mapped crossover. Sometimes called PMX, this operator selects two positions randomly within the Parent1 genome, defining a substring. For each gene in the substring, it is noted which gene corresponds positionally (or is *mapped*), in Parent2. Then, build the offspring by taking each parent, and copy the genes into the child, but every time you reach a mapped gene, swap the values.

Order-based crossover. Choose several random genes from Parent1. Impose the same order they are found in the same genes within Parent2, by swapping values as needed.

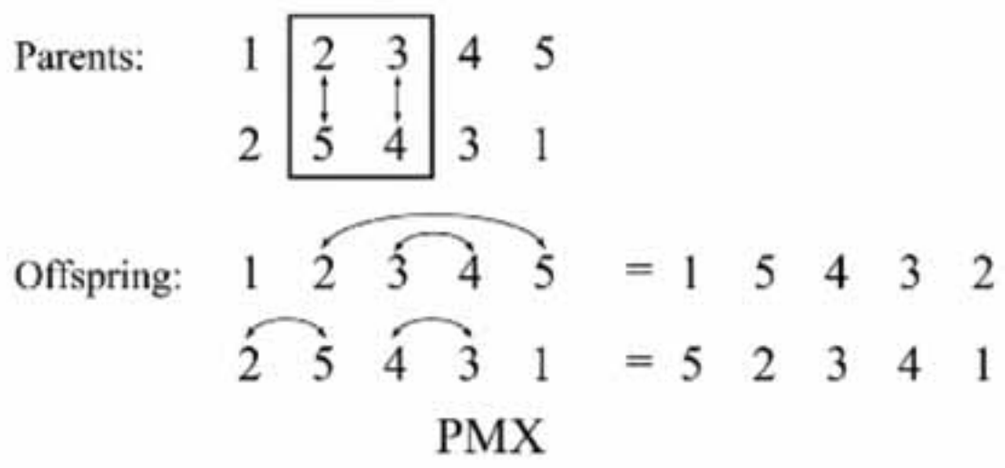


FIGURE 20.4 Order-specific crossover operators.

TeamLRN

Position-based crossover. This is like order-based crossover, except we impose the position the randomly selected genes from Parent1 are found in to the Parent2 genome, and vice versa. Select some random genes in Parent1. Then, put these values into a new genome, in the same positions as you found them. Then, fill the rest of the new genome with the values of Parent2, making sure not to use a gene that is already present in the array.

After everyone has been either copied or crossed over into the offspring population, the last step in reproduction can occur: mutation. Again, this is simply applying an operator to the genes in each offspring genome. The rate of mutation, or the chance that any one gene will be mutated, can vary wildly depending on the problem; various academic papers ([Bäck93], [MSV93]) have reported that 1/(number of variables in your GA) produced good results for a wide range of test functions. A typical number used in bit string style-genomes is around `0.0001f`, whereas the rate when dealing with real numbers is usually much higher, more in the range of `0.05f` to `0.2f`. The specific type of mutation operator that you need to apply is related to the specific structure of the genomes you are using in your GA.

For order-specific genomes, common mutation operators include the following (see Figure 20.5):

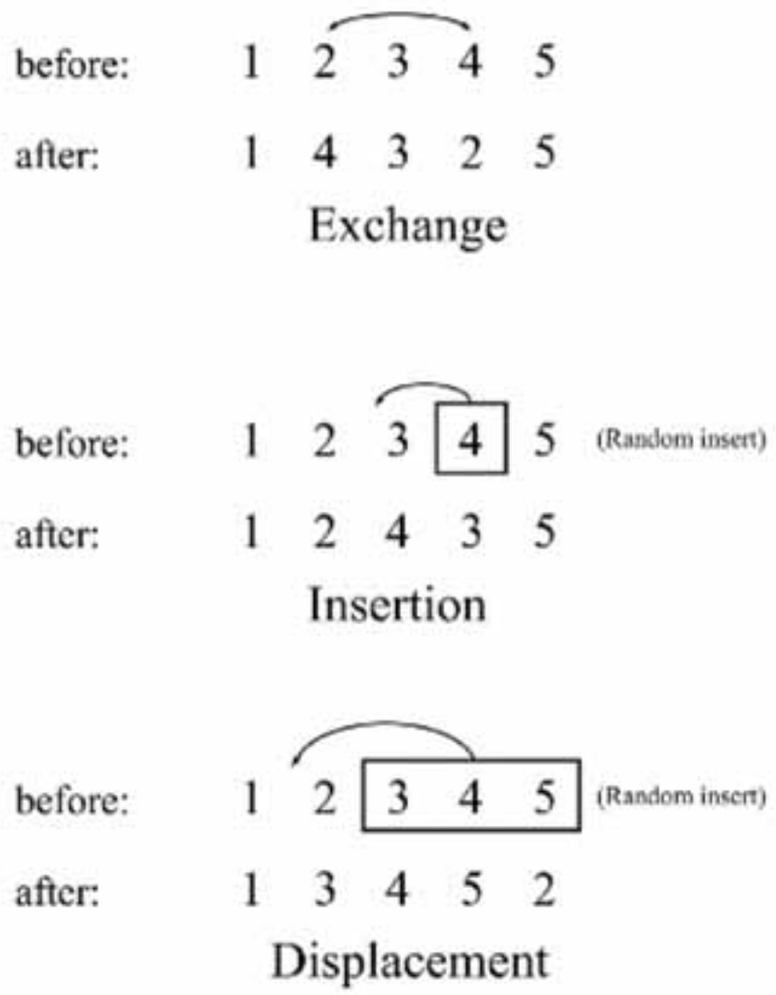


FIGURE 20.5 Order-specific mutation operators.

Exchange mutation. Swap two genes within the genome.

Displacement mutation. Select two random positions within the genome, defining a substring. Then, remove this substring, and re-insert it into a random position.

Insertion mutation. The same as displacement, except that the substring is only one gene. Tests have shown that for order specific GAs, this operator performs consistently better than others do. Your results may vary, however.

Otherwise, non-order-specific operators include the following (see Figure 20.6):

Binary mutation. Merely flip the bit with the genome.

Real-value mutation. Offset the value of a gene by some delta. The size of the delta is somewhat difficult to choose; small steps are often successful, but may take much longer.

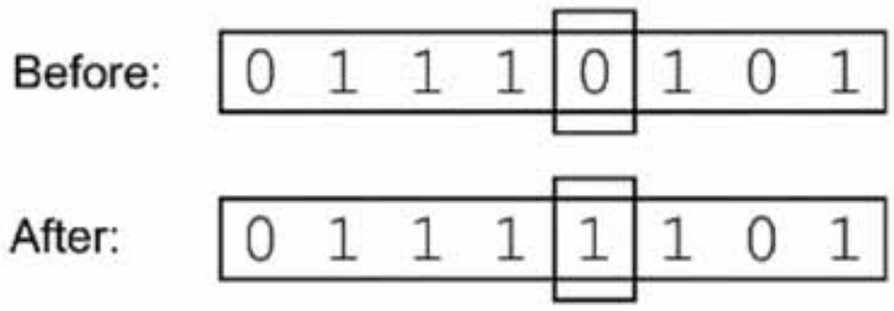


FIGURE 20.6 Non-order-specific mutation operators.

TeamLRN

IMPLEMENTING A GENETIC ALGORITHM SYSTEM INTO THE AISTEROIDS TEST BED

Straightforward *Asteroids* (meaning without large extensions to the core gameplay) has very few problems that would actually require GA techniques to solve. Most of the determinations can be broken into simple math, with some breakup of individual states that might influence behavior. But, for illustration, one aspect of the AI's current behavior stands out as needing some help: the `Evade` state. For our sample implementation, we will design a GA solution to improve the evasion capabilities of our AI ship.

The gene design we'll be using is very simple, mostly because the ship's movement capabilities are so simple; at any given time, the ship can essentially only thrust or turn. So, a gene will be defined as a two `chars`, the first value meaning thrust type (forward, reverse, or no thrust), and the second representing an unsigned integer between 0 and 17 signifying the *sector* that the AI wants the ship to point to. A sector is defined as 20 degrees in our demonstration, meaning that there are 18 possible sectors. Given the range of these variables, this data could be compressed for size with no loss of resolution, if required.

The way that we'll use this gene is as a solution to the question, "Given a certain game state, which direction should I turn to, and how should I use my thrusters?" We will then create a means for defining the current state of the game, such that the GA can then solve a genome that stores all the solutions for each game state. We're only dealing with evasion of asteroids, so the only perceptions we need to encode are those that will help us in that endeavor. A simplified evasion-specific game state should consider three things (all concerning the ship, and the nearest asteroid):

- How fast am I moving toward you? First, we'll determine a normalized *delta* vector, defined as the sum of the normalized vectors of the ship's movement and the asteroid's. The speed at which the two are moving together is calculated by multiplying the ship's speed by the magnitude of this delta vector projected on its movement vector plus the same calculation for the asteroid (see Figure 20.7). The higher this number is, the faster we're moving together. We want to quantize this value into a manageable range, so we'll scale it to the range 0–9, giving us 10 possible collision states.
- What direction are we moving in? Again, we'll quantize this value to limit the number of game states. Calculating this value will use the same normalized movement delta vector as the collision state. We will also simply calculate the angle that vector points in, and then scale it into our 0–17 sector values, for 18 possible direction states.

- What is the separation distance? Last, we need to know how far apart we are. We really only care when the two are pretty close, so we'll again quantize this value down into a few base distances that we care about. The way we'll do this is by metering the distance between the two objects in the units of "asteroid radii," referring to the asteroid we're trying to evade. Thus, if the asteroid is within one radius distance from the ship, its separation distance is one. Two radii is a separation distance of two, and so on, until four (or more) radii distance, when we'll cap the separation distance being considered, giving us four distance states.

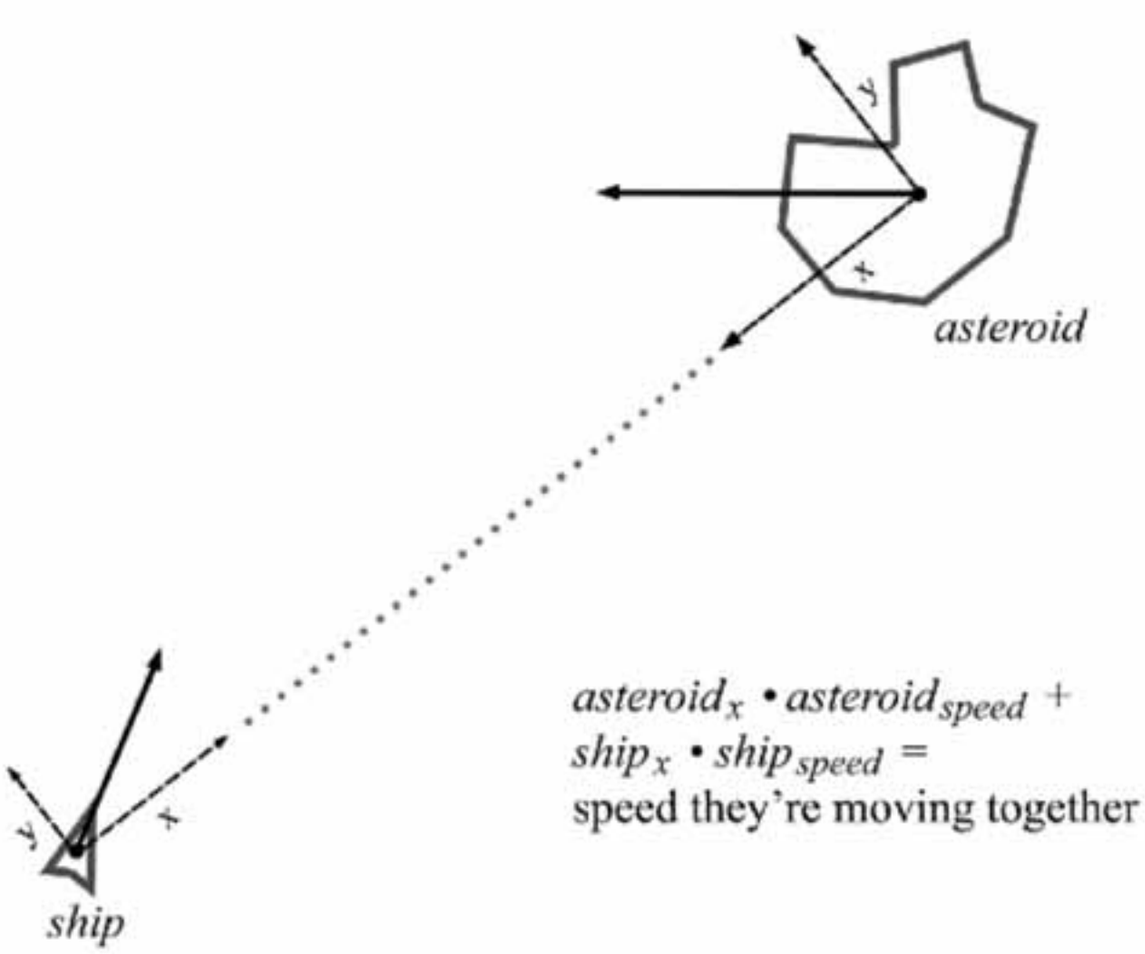


FIGURE 20.7 Diagram of the collision state calculation.

All told, we have defined a theoretical set of rules, given the collision state, direction state, and distance state of the ship and the nearest asteroid—520 given distinct evasion scenarios (10*18*4). If the ship knew how to react to all of these 520 possible game states, it would be pretty good at evading. Again, this implementation could definitely be done more simply using mathematical constructs; this will just be a demonstration of the technique.

So, this is the quest that we will set the GA to solving. We'll define our genome as the collection of rules that will most successfully solve all the necessary evasion rules, meaning our genome will have 520 genes. Listing 20.1 shows the header for the gene and genome.

TeamLRN

LISTING 20.1 Gene and Genome Header Information

```
class Gene
{
public:
    //methods
    Gene() {m_thrust = randint(0,2);m_sector = randint(0,NUM_SECTORS-
1);}
    Gene(int a, int d):m_thrust(a), m_sector(d){}
    bool operator==(const Gene &rhs) const {return (m_thrust ==
                    rhs.m_thrust) && (m_sector == rhs.m_sector);}
    bool operator!=(const Gene &rhs) const {return (m_thrust !=
                    rhs.m_thrust) || (m_sector != rhs.m_sector);}

    enum
    {
        THRUST_OFF,
        THRUST_FORWARD,
        THRUST_REVERSE
    };

    //data
    char m_thrust;
    char m_sector;
};

class Genome
{
public:
    //methods
    Genome():m_fitness(0){}
    Genome(const int num_genes):m_fitness(0)
                    {for(int i=0; i<num_genes;++i)
                        m_genes.push_back(Gene());}

    bool operator<(const Genome& rhs){return (m_fitness <
                                              rhs.m_fitness);}

    //data
    vector<Gene> m_genes;
    float        m_fitness;
};
```

Both classes are very simplistic. `Gene` stores the information that will be genetically modified, whereas a `Genome` is a collection of genes, as well as a fitness score. In our AIsteroids example, we will be storing a "plan" for surviving a collision event, in the form of a thrust setting and a direction sector to turn to. When used for a different game (or use within the same game), this is where you would define the set of genetic material you have to work with, whether it is bit strings, real variables like in our example, or complex data structures like trees. The `Genome` class can be used generically, but the `Gene` class is so basic to the particular implementation that you really have to address its design on a per use basis.

We will implement the *evolution application* (EA), which is where we will first build a suitable rule system that the real game code can use. The parts of this application will be a different `GameSession` class (called `TestSession`), a different set of keyboard controls (called `HumanTestControl`), a new `AIControl` class, `GAAIControl`, and most important, the `GAMachine` class, which houses the bulk of the GA functionality.

`TestSession` and `HumanTestControl` are mostly just the "game side" support code for the EA, meaning that they handle the inputs, drawing code, main game update loop, and so forth. The only controls that the tester application includes are the standard speed up and slow down buttons (. and , respectively), the "step" functionality still operates, and there is a Reset button (the *r* key). As for the session, it's basically the same application, except that it spawns a number of asteroids and ships, and when they collide, it just deactivates them, instead of killing them off. Then, at the end of a generation (when all the ships are deactivated), they are reset, reactivated, and start another round.

The real headquarters for the genetic algorithm code is in the `GAMachine` class. Listing 20.2 shows its header. We're going to split the function implementation into Listings 20.3–2.11, with a brief discussion of each one in turn.

The first thing you should notice about the header is that there are more functions than are necessary for our test application. There are two types of selection, six crossover operators, and four mutation operators. These are implemented to provide you with additional tools to use in your AI programs, as well as to give you some things to tweak within the test bed, to see its affect on the quality of the evolution. Remember, GAs are all about experimentation, and finding out what operators to use, as well as how to tweak key values (the crossover rate, mutation rate, amount of elitism, etc.) is most of the difficulty in using the GA method.

LISTING 20.2 GAMachine Header

```
class GAMachine
{
public:
    GAMachine(GAAIControl* parent):m_parent(parent){}
    void SetupNextGeneration();
```

TeamLRN

```
        void CreateStartPopulation();
        void Update(float dt);
        void UpdateFitness(int index);
        void Init();
        void Reset();
        void ApplyBehaviorRule(int index);
        bool WriteSolution();
        bool ReadSolution();

        //selection operators
        Genome& SelectRouletteWheel();
        Genome& SelectTournament();
        Genome& SelectRank();

        //crossover operators
        void CrossUniform(const vector<Gene> &parent1,
                          const vector<Gene> &parent2,
                          vector<Gene>&offspring1,
                          vector<Gene>&offspring2);
        void CrossSinglePoint(const vector<Gene> &parent1,
                              const vector<Gene> &parent2,
                              vector<Gene>&offspring1,
                              vector<Gene>&offspring2);
        void CrossMultiPoint(const vector<Gene> &parent1,
                             const vector<Gene> &parent2,
                             vector<Gene>&offspring1,
                             vector<Gene>&offspring2);
        //crossover operators - order based genes
        void CrossPMX(const vector<Gene> &parent1,
                      const vector<Gene> &parent2,
                      vector<Gene>&offspring1,
                      vector<Gene>&offspring2);
        void CrossOrderBased(const vector<Gene> &parent1,
                             const vector<Gene> &parent2,
                             vector<Gene>&offspring1,
                             vector<Gene>&offspring2);;
        void CrossPositionBased(const vector<Gene> &parent1,
                                const vector<Gene> &parent2,
                                vector<Gene>&offspring1,
                                vector<Gene>&offspring2);

        //mutation operators
        void MutateOffset(vector<Gene> &genes);
        //mutation operators - order based genes
```

```
    void MutateExchange(vector<Gene> &genes);
    void MutateDisplacement(vector<Gene> &genes);
    void MutateInsertion(vector<Gene> &genes);

    //elitism
    void CopyEliteInto(vector<Genome>&destination);

protected:
    GAAIControl*    m_parent;
    //genetic data
    vector<Genome>  m_genomes;
    int             m_rankIndexLast;
    Genome          m_bestGenome;
    int             m_generation;
    float           m_crossoverRate;
    float           m_mutationRate;
    float           m_offsetSize;
    float           m_bestFitness;
    float           m_totalFitness;
    int             m_liveCount;
};
```

LISTING 20.3 GAMachine::Update() Implementation

```
//--------------------
void GAMachine::Update(float dt)
{
    //find best out of the maximum tries, then start over
    if(m_generation > NUM_MAX_GENERATIONS)
    {
        WriteSolution();
        //reset
        CreateStartPopulation();
        Reset();
    }

    m_liveCount = 0;
    for (int shpNum=0; shpNum<POPULATION_SIZE; ++shpNum)
    {
        if(!Game.m_ships[shpNum]->m_active)
            continue;
        m_liveCount++;
        m_parent->UpdatePerceptions(dt,shpNum);
        ApplyBehaviorRule(shpNum);
```

TeamLRN

```
            UpdateFitness(shpNum);
        }

        //if the generation is over...
        if(!m_liveCount)
            SetupNextGeneration();
    }
```

The `Update()` function is the main loop of the GA. This function first checks to see if you've run some maximum number of generations (thereby running an entire *simulation*), and then writes out the best genome and starts it over. If you want, you could write out the top 10 genomes, or the whole list. The reason for this is that you will most likely be running this program overnight, or at the very least unsupervised, to give it the time it needs to evolve fully into a working solution. You could even set up the system to use slightly different GA parameters, or even different genetic operators, for the different runs, and after an overnight session, you would have a few different solutions to compare and contrast.

If the simulation isn't over, it updates each ship's perception values (take notice that the `GAMachine` is calling the `GAAIControl::UpdatePerceptions()` function; usually the controller updates himself, but within our GA teaching program, more than one ship is being controlled, so the GA machine has to update them separately), then applies the evasion rule for that ship (given those current perceptions), and then scores the ship based on how well it is performing. If there are no active ships left, it calls `SetupNextGeneration()`, the evolution function.

LISTING 20.4 `GAMachine::ApplyBehaviorRule()` Implementations

```
//--------------------
void GAMachine::ApplyBehaviorRule(int index)
{
    if(index < 0 || index > POPULATION_SIZE)
        return;

    Ship* ship = (Ship*)Game.m_ships[index];

    //not going to collide, just idle...
    if(m_parent->m_currentEvasionSituation == -1)
    {
        ship->ThrustOff();
        ship->StopTurn();
        return;
    }
```

```
//thrust
int thrustTp = m_genomes[index].
          m_genes[m_parent->m_currentEvasionSituation].m_thrust;
ship->StopTurn();
if(thrustTp == Gene::THRUST_FORWARD)
    ship->ThrustOn();
else if(thrustTp == Gene::THRUST_REVERSE)
    ship->ThrustReverse();
else
    ship->ThrustOff();

//turn
//-10 puts you in the middle of the sector
float newDir = m_genomes[index].
               m_genes[m_parent->m_currentEvasionSituation].
                                                  m_sector*20 -10;
float angDelta = CLAMPDIR180(ship->m_angle - newDir);
if(fabsf(angDelta)<=90)
{
    if(angDelta >0)
        ship->TurnRight();
    else
        ship->TurnLeft();
}
else
{
    if(angDelta<0)
        ship->TurnRight();
    else
        ship->TurnLeft();
}
}
```

`ApplyBehaviorRule()` takes the particular ship's current evasion state, `m_currentEvasionSituation`, and applies the correct rule coded within the ship's genome by setting the thrusters, and also possibly turning the ship toward some new goal direction. If the ship isn't currently in danger of a collision, the evasion state gets passed in as –1. When the ship registers this, it stops turning and thrusting.

LISTING 20.5 `GAMachine::UpdateFitness` Implementation

```
//--------------------
void GAMachine::UpdateFitness(int index)
```

TeamLRN

```
{
    Ship* ship = (Ship*)Game.m_ships[index];
    if(ship && ship->m_active)
    {
        //if I'm currently surviving a collision situation,
            //incr fitness
        if(m_currentEvasionSituation != -1)
            m_genomes[index].m_fitness++;
        m_liveCount++;
    }
}
```

Fitness, for our test bed, is being based on how often each ship was in an evasion situation and didn't die. The function does this by checking some perceptions (being active, and that the ship currently has something to evade), and if they are true, it increments the ship's fitness value. For our test bed, we aren't using any fitness scaling. Scaling the fitness scores would probably be done after all the individual genomes have been run through their update and fitness calculations. A very simple way to introduce scaling into the test bed is to implement rank scaling. Given that we already sort the genome list (for elitism, keeping fitness statistics, and roulette wheel selection), you could just make a post-sort pass through the list, changing each genome's fitness to be his position within the genome list. If you perform this exercise on the test bed, it should help keep the program from converging on a local maximum to early.

LISTING 20.6 GAMachine::SetupNextGeneration() Implementation

```
//--------------------
void GAMachine::SetupNextGeneration()
{
    //next Generation
    vector<Genome> offspring;

    //sort the population (for scaling and elitism)
    sort(m_genomes.begin(), m_genomes.end());
    m_rankIndexLast = POPULATION_SIZE-1;

    //statistics
    m_totalFitness = 0.0f;
    for (int i=0; i<POPULATION_SIZE; ++i)
        m_totalFitness  += m_genomes[i].m_fitness;
    m_bestFitness   = m_genomes[POPULATION_SIZE - 1].m_fitness;

    CopyEliteInto(offspring);
```

```
while (offspring.size() < POPULATION_SIZE)
{
    //selection operator
    Genome parent1 = SelectRouletteWheel();
    Genome parent2 = SelectRouletteWheel();

    //crossover operator
    Genome offspring1, offspring2;
    CrossSinglePoint(parent1.m_genes,
        parent2.m_genes,
        offspring1.m_genes,
        offspring2.m_genes);

    //mutation operator
    MutateOffset(offspring1.m_genes);
    MutateOffset(offspring2.m_genes);

    //add to new population
    offspring.push_back(offspring1);
    offspring.push_back(offspring2);
}

//replace old generation with new
m_genomes = offspring;

for(i = 0;i<POPULATION_SIZE;i++)
    m_genomes[i].m_fitness = 0.0f;

++m_generation;

//reactivate the ships
for (int shpNum=0; shpNum<POPULATION_SIZE; ++shpNum)
{
    //reset test ships to startup state
    Ship* ship = (Ship*)Game.m_ships[shpNum];
    ship->m_active = true;
    ship->m_velocity.x() = 0;
    ship->m_velocity.y() = 0;
    ship->m_velocity.z() = 0;
    ship->MakeInvincible(3.0f);
}


}
```

TeamLRN

The `SetupNextGeneration()` function is where all the real genetic work happens. It sorts the genomes, tallies the statistics for the generation, uses the elitism function, and then creates the rest of the next generation by using roulette wheel selection, the single point crossover operator, and the offset mutation operator. It also resets the ships for the next generation restart.

LISTING 20.7 `GAMachine::CopyEliteInto()` Implementations

```
//--------------------
#define NUM_ELITE              4
#define NUM_COPIES_ELITE       2
void GAMachine::CopyEliteInto(vector<Genome>&destination)
{
    int numberOfElite = NUM_ELITE;
    //copy the elite over to the supplied destination
    for (int i=numberOfElite; i>0; –i)
    {
        for(int j=0;j<NUM_COPIES_ELITE;++j)
            destination.push_back(m_genomes[(POPULATION_SIZE - 1) –
                                            numberOfElite]);
    }
}
```

The `CopyEliteInto()` function copies a set number of the top members of the population into the next generation. These are straight copies, with no crossover or mutation. You might want to introduce some mutation into these elements, possibly with lower probability, or with somewhat nonintrusive mutations (possibly offset mutation with a very small offset). Again, these types of experiments and tweaks are the requirement of working with a GA system.

LISTING 20.8 `GAMachine::SelectRouletteWheel()` Implementations

```
//--------------------
Genome& GAMachine::SelectRouletteWheel()
{
    float wedge = randflt() * m_totalFitness;
    float total = 0.0f;

    for (int i=0; i<POPULATION_SIZE; ++i)
    {
        total += m_genomes[i].m_fitness;
        if (total > wedge)
            return m_genomes[i];
```

```
    }
    return m_genomes[0];
}
```

`SelectRouletteWheel()`is a straightforward implementation of the roulette wheel algorithm. Also known as fitness proportional selection, it is built on the idea that the higher your fitness, the better your chances of being chosen for reproduction. However, because it relies completely on the random call at the top of the function, the real results of this selection process might not match your expectations. Indeed, it may completely miss the best individuals altogether, hence the reason that elitism is commonly used in conjunction with this type of selection. For certain problems, especially those with very small populations, stochastic universal sampling (SUS) or tournament selection are sometimes better for this reason.

LISTING 20.9 `GAMachine::CrossUniform()` Implementation

```
//--------------------
void GAMachine::CrossUniform( const vector<Gene> &parent1,const
vector<Gene> &parent2,

vector<Gene>&offspring1,vector<Gene>&offspring2)
{
    if ( (randflt() > m_crossoverRate) || (parent1 == parent2))
    {
        offspring1 = parent1;
        offspring2 = parent2;
        return;
    }

    for (int gene=0; gene<GENOME_SIZE; ++gene)
    {
        if (randflt() < m_crossoverRate)
        {
            //switch the genes at this point
            offspring1.push_back(parent2[gene]);
            offspring2.push_back(parent1[gene]);

        }
        else
        {
            //just copy into offspring
            offspring1.push_back(parent1[gene]);
            offspring2.push_back(parent2[gene]);
```

TeamLRN

```
            }
        }
    }
```

The implementation of uniform crossover is as simple as that. You pick a random location within the gene, swapping everything before that point, and straight copying over everything after it. The operator checks (as does all the crossover operators) to see if you've passed in identical parents, in which case it can skip the real algorithm. Identical parents will have identical offspring, which is the reason that too much convergence of your genetic material will lead toward a population with almost no variation (this is fine only if you've found the solution to the problem).

LISTING 20.10 `GAMachine::MutateOffset()` Implementation

```
//--------------------
void GAMachine::MutateOffset(vector<Gene> &genes)
{

    for (int gene=0; gene<genes.size(); ++gene)
    {
        //check for thrust mutation
        if (randflt() < m_mutationRate)
        {
            genes[gene].m_thrust += (randint(0,1)?
                                    -m_offsetSize: m_offsetSize);
            //bounds check
            if(genes[gene].m_thrust > NUM_THRUST_STATES)
                genes[gene].m_thrust = 0;
            if(genes[gene].m_thrust < 0)
                genes[gene].m_thrust = NUM_THRUST_STATES;
        }

        //check for angle mutation
        if (randflt() < m_mutationRate)
        {
            genes[gene].m_sector += (randint(0,1)?
                                    -m_offsetSize: m_offsetSize);
            //bounds check
            if(genes[gene].m_sector > NUM_SECTORS)
                genes[gene].m_sector = 0;
            if(genes[gene].m_sector < 0)
                genes[gene].m_sector = NUM_SECTORS;
```

```
            }

        }
    }
```

The offset mutation simply changes the real value of the variable by +or– the offset value. It also checks for wrapping of the value because we don't want the variables to hit a hard floor or ceiling. Instead, we want them to be able to move that little bit that might just find a better solution. The size of the offset is usually a tradeoff between being large enough to actually get the solution from the local maximum (without going right back in), without being so large that you skip over solutions. Notice too, that smaller offsets are usually better, but your algorithm takes longer to find a solution.

The only function of note within the new `GAAIControl` class is its `UpdatePerceptions()` method, which is shown in Listing 20.11.

LISTING 20.11 `GAAIControl::UpdatePerceptions()` Implementation

```
//--------------------
void GAAIControl::UpdatePerceptions(float dt,int index)
{
    Ship* ship = (Ship*)Game.m_ships[index];
    if(!ship)
        return;

    //determine current game evasion state
    int collisionState = -1;
    int directionState = -1;
    int distanceState  = -1;

    //store closest asteroid
    m_nearestAsteroid = Game.GetClosestGameObj(ship,
                                        GameObj::OBJ_ASTEROID);

    //reset distance to a large bogus number
    m_nearestAsteroidDist = 100000.0f;

    if(m_nearestAsteroid)
    {
        Point3f normDelta = m_nearestAsteroid->m_position –
                            ship->m_position;
        normDelta.Normalize();
```

TeamLRN

```
            //asteroid collision determination
            float speed = ship->m_velocity.Norm();
            m_nearestAsteroidDist = m_nearestAsteroid->
                                m_position.Distance(ship->m_position);
            float astSpeed = m_nearestAsteroid->m_velocity.Norm();
            float shpSpeedAdj = DOT(ship->
    UnitVectorVelocity(),normDelta)*speed;
            float astSpeedAdj = DOT(m_nearestAsteroid->
                                  UnitVectorVelocity(),-
    normDelta)*astSpeed;
            speed = shpSpeedAdj+astSpeedAdj;
            speed = MIN(speed,m_maxSpeed);
            collisionState = (int)LERP(speed/m_maxSpeed,0.0f,9.0f);

            //direction determination
            directionState = GETSECTOR(normDelta);

            //distance determination
            distanceState  = MIN((int)(m_nearestAsteroidDist/
                                       m_nearestAsteroid->m_size),4);
        }
        if(collisionState == -1)
            m_currentEvasionSituation = -1;
        else
            m_currentEvasionSituation=
             (collisionState*10)+(directionState*18)+distanceState;
    }
```

`UpdatePerceptions()` works just like it did in the previous controller classes: It computes the perception values that the decision making portion of the program will use in making up its mind. In our case, its primary job for this demonstration is to compute the variable `m_currentEvasionSituation`, which the ship will use to employ the correct evasion rule. The reasoning behind this value's computation was covered earlier in this chapter.

PERFORMANCE WITHIN THE TEST BED

Even with the low level of genetic complexity being applied to our AIsteroids program, you can begin to see improvement in overall evasion behavior with only a few generations (50 or so), and letting the program run for thousands of generations leads to some very unusual, although still useful, behavior. Figure 20.8 shows a screenshot of the test bed running the GA solution. There are, however, some low

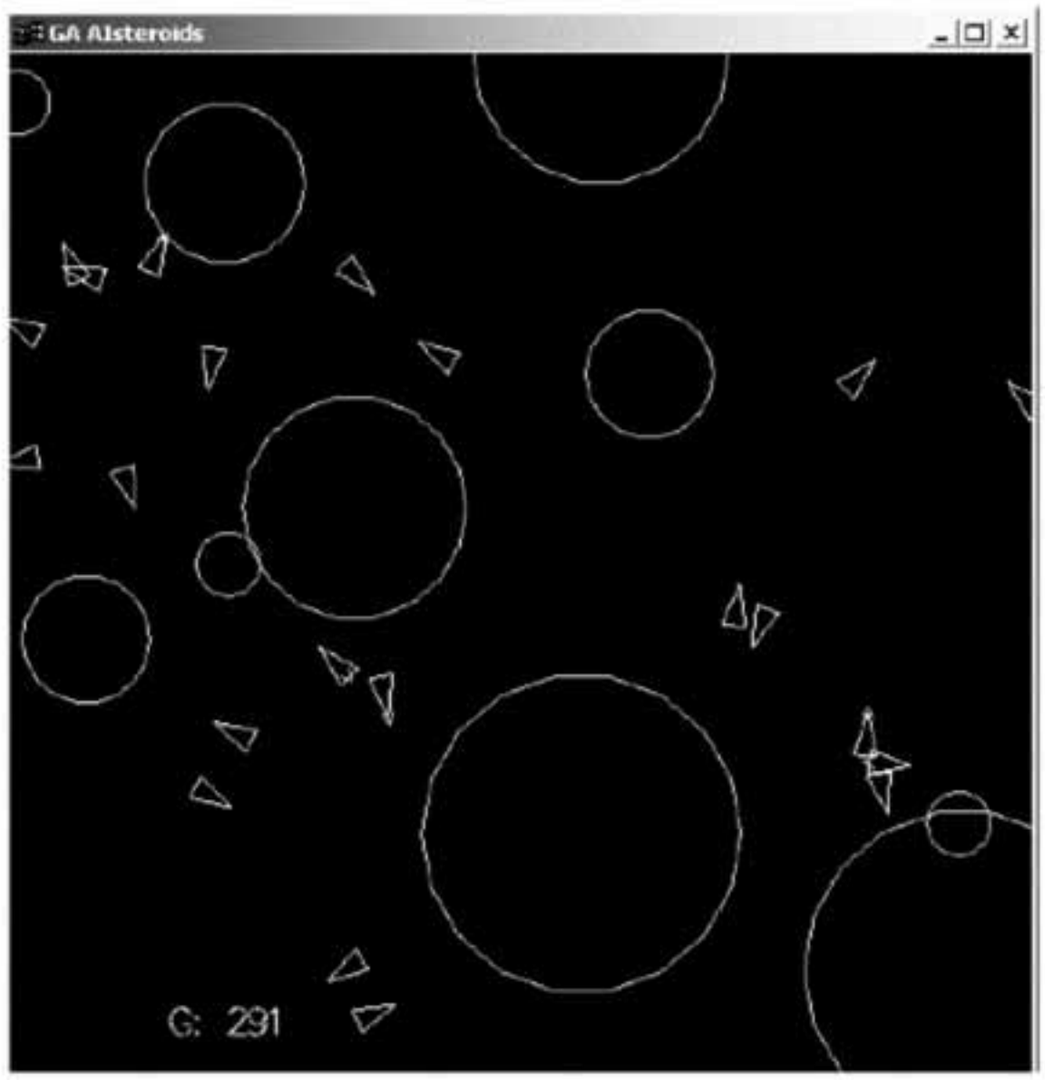


FIGURE 20.8 AIsteroids running the GA implementation.

points to the method as we have currently implemented it, and the performance of the GA could be improved in many ways. Some of them include the following:

- The system seems to converge too quickly on a few individuals; a different selection operator might improve on this, as would more individuals in the population, or slightly less elitism.
- We're trying to optimize a rule set with a substantially large number of rules. This means that to get a truly optimal set, we're going to have to let the system run for a very long time indeed—hundreds of thousands of generations or more. Another way to encode the genes might have been to use analog values that could have been used as coefficients in a function that computed the best direction and thrust. Then, instead of trying to genetically search for the solutions to all the evasion states, we are only searching for the necessary amount of coefficients within a sufficiently complex algorithm to represent our evasion calculations. A method like this would run somewhat close to being a simple neural net (albeit one that used a genetic algorithm to train), which will be discussed in the next chapter.
- The test application really needs to be fully time independent, to allow the GA to run through generations at much higher speed. Involved in this would be

TeamLRN

small things like making sure that the `GameSession::m_timescale` variable is incorporated into all calculations dealing with time, including speed determinations. Also, the collision detection would have to allow for collisions "within" a game tick. What this means can be shown within Figure 20.9. What can happen is that the change in position that a game object might perform from one game loop to the next can become so large (when time is scaled very highly) that one game object could move straight through another, but because they were never touching during a collision check, one is never triggered. Solving this anomaly involves keeping track of the old position and actually performing a line of sight check to your new position, to check the entire path of motion along the delta. If another object resides along that path, then the object can't go all the way to its new position; the game should register a collision, and stop the object at the obstruction.

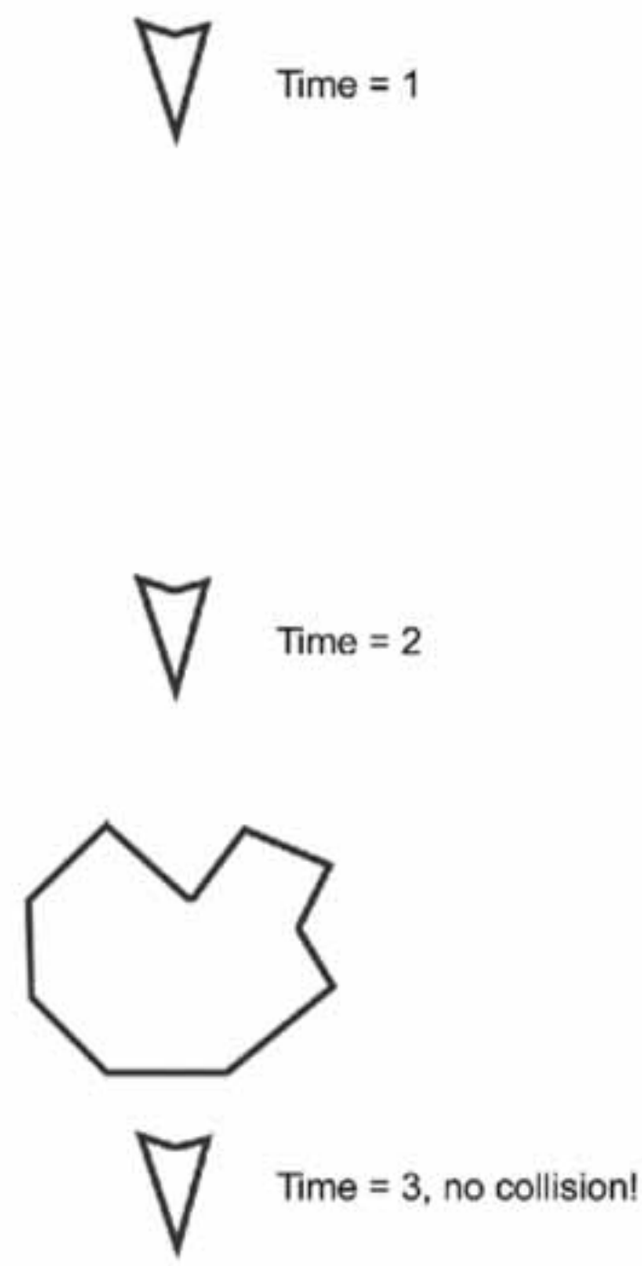


FIGURE 20.9 Highly scaled game-time problem.

Pros of Genetic Algorithm-Based Systems

Although GAs are clearly not a universal tool in game AI construction, they do have a number of areas in which they work particularly well, including the following:

- *When you have a number of parameters that interact in highly nonlinear or surprising ways.* The more your parameters work in tune with each other, the easier it is to find a more traditional method for algorithmically solving your problems.
- *When you have many local maximums, and are searching for the* best *one.* An example might be the earlier stated case of tuning the physics parameters for the different cars and AI personalities in a driving game. Many different combinations will work, but the developers are looking for a particular feel; finding it will require more than just experimentation.
- *For solutions that involve discontinuous output.* Perfectly continuous output would be a simple mathematical function that always maps inputs to outputs with a simple function call. Semicontinuous output might be a state machine, where there are always actions in result of any given game state, but to encapsulate everything, it had to be broken up into separate, individual states that are islands of separate behavior. Discontinuous behavior is not smooth and contains islands of action that are not connected to each other with any sort of relation.
- *When complementing traditional techniques.* GAs can be incredibly modular, fitting easily into the larger AI system when the need arises. Within a game, you might have an isolated state whose decision-making needs make it a good candidate for GA methods. If you can create a means within your game framework in which to allow a GA to evolve, and can come up with a suitable fitness function, then you're well on your way to evolving a solution instead of being stuck with trial and error methods.
- *When actual computation of a decision might be too costly.* Then, if you can find a suitable GA solution, you can probably save quite a bit of CPU time. GA solutions can be implemented as black box functions to replace complex mathematical constructs that algorithmically solve game problems, thus optimizing the AI. This is a bit like a neural net's ability to abstract mathematic constructs. You can, in essence, construct a nonlinear function, with some coefficients and extra parameters that become your genome. The fitness function then becomes the difference in output that your genetic function produces from your complex math function, and you can then set your GA on solving for the genome that minimizes that difference. Of course, you only win if the nonlinear function you've come up with is less CPU-costly than the one you're trying to outdo.

TeamLRN

GAs also have a number of *general* pros that are inherent to the method. They are very easy to set up, and start getting results, even if you don't know how to solve the problem otherwise. During evolution, you have an entire set of candidates to try out in your game, which could result in many of them being used to create variety or personality within your game AI characters. GAs are often a very strong *optimization* algorithm, meaning that you can frequently find the optimal solution to a given situation. Finally, GAs tend to find global solutions rather than getting stuck in local ones precisely because they operate *in parallel.* In contrast with more traditional numerical or search-based techniques, which iteratively refine a single solution in hopes of coming up with an answer, GAs work by evaluating an entire population of candidate solutions simultaneously. In effect, more standard methods are asking the question "Do you know the time?" whereas GAs ask, "Does anybody in town know the time?"

Cons of Genetic Algorithm-Based Systems

GAs are not a free lunch. Like any AI system, the more time you put into them, the better the results you will receive. Some of the shortcomings of GA systems include that it can be time consuming, performance can be hit or miss, there are weak definitions of success and failure, there is no guaranteed optimal solution, and it is tough to tune and add functionality to GAs.

Time-Consuming Evolution

Often, evolution takes many generations, even with a good genome design and the right operators, to see results that are "game ready," meaning that they work well most of the time (a local maximum rather than a global maximum). Couple this with the fact that you might have to frequently change any one to all the component parts of the system while trying to increase performance (the gene makeup, one or more of the genetic operators, tweaks to the mutation or crossover rates, etc.), and then restart evolution, and you see the importance of ensuring that you have a considerable amount of time set aside for this portion of the method. Again, this can be lessened if the system for which you are evolving a solution can be either arbitrarily sped up, or you can calculate an estimated fitness algorithmically from the genome, thus reducing each generation to be the act of running this function on each individual, instead of having to play through the game loop for some amount of time. Usually, however, this cannot be done. For game time, the algorithm as set up currently is quite slow to perform adequately. We couldn't allow the algorithm to run during real gameplay, evolving as it goes, because it would be dead long before it evolved a good evasion technique. Because of this, we perform the evolution offline, before the game is released, like most applications of the technique.

Hit or Miss Performance

With the myriad different ways you could encode a problem genetically, the vast array of different operators for selection, crossover, mutation, and fitness scaling, the large number of secondary parameters such as population size and mutation rate, as well as the highly subjective creation of a suitable fitness function, GAs are the absolute pinnacle of tweakability. Given your particular game problem, you may get very good performance with a certain crossover operator, but only if you use elitism, low crossover, a large population size, and real values for your genes. But figuring out that exact mix of usage might take a long time, experimenting with different combinations of these factors until you discover the right set of conditions necessary to find the solution. The only real way to become good at knowing how to use the right factors with any given GA problem is through experimentation, especially because of the implementation-specific nature of GA solutions.

Weak Definition of Success and Failure

Your GA doesn't seem to be working, but you have no idea why. Is it that your mutation operator is scrambling things too much, and so you're never converging on a solution, or have you converged on an inferior local maximum, and therefore need *additional* or more invasive mutations? Once more, you are left to the mercy of experimentation or gut feeling to try to divine these types of issues. In fact, the *reason* your GA might be not working or, even working, may be a bug in your fitness function or genetic operators. Because of the nature of GA output, it's hard to tell the difference between buggy code and unevolved behavior. Real care must be taken in coding GA functions, so that this kind of problem doesn't come back to haunt you.

No Guaranteed Optimal Solution

GAs use stochastic techniques, and anytime there's randomness, we have no guarantees. There are methods for ensuring a measure of safety (meaning, that you don't accidentally throw away winning solutions with bad selection operators or ill timed mutations) while keeping the usefulness of GA methods, but it's all a gamble in the end. Another problem is that you most likely don't know the optimal solution (which is why you chose a GA in the first place), so you don't realize that if you'd just tweak a few things in your algorithm, you could get much better responses.

Tough to Tune, and Even Tougher to Add Functionality

Once you have a GA-developed solution to a problem, especially if that solution is hard won after a lengthy period of evolving and tweaking of the GA implementation, the tendency is to "leave well enough alone." Meaning that you don't want to

TeamLRN

muck with things very much, for fear of losing your hard-won system. Game developers rarely have the foresight to include everything into the up-front design of the games' AI requirements, however. Most often, AI tuning is performed during the final period of the game, with many play testers and other personnel giving feedback. With code-based systems, these kinds of tuning issues or even the addition of small features (called *feature creep* by some, *polish* by others) can be accomplished relatively easily, especially if you've designed your AI system with this in mind (using a data-driven system, or some kind of extensible system). But with a GA-based system, these kinds of issues are much more difficult to approach. Basic tuning might still be capable, by slightly re-evaluating how you interpret the GA solution data (as an example within in our test bed, we could point the ship toward a slightly different angle then originally planned). But actually adding even small features might involve completely starting over as far as the gene structure and GA evolution is concerned. This reason alone is the primary killer as far as games are concerned, and is the main reason why games are still using GAs for small parts of their games that *do* have locked down designs and aren't subject to last minute changes.

EXTENSIONS TO THE PARADIGM

Ant Colony Algorithms

Ants, it turns out, are not very good problem solvers. Individually, that is. Put a solitary ant in an environment, and he's as good as dead. He'll amble about in all directions, with no apparent plan or strategy. But this all changes when you have a large group of ants. If you put half a million ants into the same environment, they will centralize, build a colony, find food, defend themselves, and even conquer neighboring colonies. How can they do this? Through what is known as *collective* (because it is brought about by a group) or even *emergent* (because it seems to come from nowhere) intelligence. One facet of this kind of intelligence that works well with GAs is the method by which ants find food. As ants walk around, they secrete a small amount of a special chemical, called a pheromone, onto the ground. The more they use a particular trail, the more pheromone is laid down. This chemical attracts other ants, and so the cycle continues until the ants have essentially built themselves a "freeway" to the nearest food source. This all sounds remarkably like a kind of influence map, doesn't it. Actually, you could build an LBI system to encode the information necessary to help implement an ant colony algorithm, but it also involves (like GA methods in general) a hefty dosage of randomness and genetic recombination. In effect, we're using the notion of collective intelligence to help guide the genetic fitness of our GA populations. What this does for our GAs is to allow them to still start with the massive, random population that they do now

while building toward solutions based on the successes of the entire population, rather than the success of an individual member with exceptional genes.

Coevolution

Another fascinating area of GAs is the concept of cooperative and competitive evolution. If the fitness function of your GA can be maximized only when two or more creatures work together, you are encouraging cooperation. When you allow two elements within your game to evolve at the same time, and increasing the fitness of one decreases the fitness of the other, they are competing. In both cases, evolution by both creatures can be sped up dramatically because of the synergistic effect of the process between multiple entities [Hillis91]. This idea has also been expanded to involve entire populations of entities, which in some ways model whole societies competing with each other. Sometimes referred to as *societal evolution,* this kind of GA evolution could be used to develop real-time strategic (RTS) civilizations that wage war on each other in the most efficient means given the specific groups, or could be used to build realistic animal communities within a game's ecosystem.

Self-Adapting GAs

The efficiency shown by our GAs depends heavily on how we use the various operators and parameters within the system. Often, it is hard to tune these systems manually. Several types of GA designs have been proposed that try to evolve the inner parameters of the GA as well as the problem-specific genetic material [Bäck92]. So, during the evolution process, the crossover or mutation rates are influenced and changed. These types of GAs again sometimes work very well, and sometimes not at all for a particular GA problem. They sometimes have a tendency to converge too quickly, but various methods have been constructed to deal with this.

Genetic Programming

In this paradigm, the genetic material encoded in the genes is composed of actual program code itself. You are evolving the program that the individual will run to solve the problem, rather than coming up with a series of magic parameters that optimize a fitness function. Crossover and mutation of game code sequences is particularly difficult (at least, to do it and still have a legitimate program afterwards), so this type of GA system is rare. But with a data-driven game AI, where your data is a series of small program instructions that represent behavior, the technique could be used to evolve AI character scripts instead of having to create them. Or, you could have the designers give you a series of working scripts as your initial population, and evolve several offshoots of these to give your AI agents some variation and personality.

TeamLRN

DESIGN CONSIDERATIONS

GAs are a brute force method that can find solutions in very difficult or computationally expensive areas of game AI, as well as come up with interesting solutions that may not have been found by a programmer. GAs are usually used offline because the evolutionary process is extremely slow in most cases. When designing your game, the question of whether to use GA methods should include the following reflections: types of solutions, agent reactivity, system realism, genre, platform, development limitations, and entertainment limitations.

Types of Solutions

Heavily strategic AI decision-making systems usually require numerous changes to their feature set during creation, as well as significant tuning for gameplay feel late in development, so they usually aren't a good match with GA methods. Tactical decisions are much more modular and can usually be split off into an evolutionary program more easily. Trying to evolve a diplomacy system within a large *Civilization*-style game would require each member of a population to play the game for quite a long time to reach any sort of fitness determination, and would probably require optimizing an entire host of parameters. But a smaller, yet still difficult problem like city building could use a GA to evolve the best ways to optimally fit buildings into an area, while maximizing defense, utility, and the like.

Agent Reactivity

The application of GA-spawned solutions are mostly black box, so they are exceptionally fast to use and usually represent somewhat optimal solutions. Therefore, reactivity of your game agents is up to you, and can be tuned to whatever you require.

System Realism

GA solutions are a mixed bag when it comes to realism. They can find solutions that are almost *too* optimized sometimes. Solutions could consider the effect of the randomness used in finding the solution, or combine all the elements of the AI character so well that the game no longer plays like a person would (for example, in some first-person shooters/third-person shooters (FTPS) games you can use the blowback from your own weapons to blast yourself to places you couldn't normally get to; a GA derived deathmatch bot might perceive this and blast itself around the map continuously, never touching the ground). This kind of behavior can be constrained, however, because off color activities like this merely represent exploits in your fitness function.

Genre

Almost any genre can use GA techniques for some aspect of its game: RTS games could solve tough problems such as building order determination or fending off a particular tactic such as rushing (a common human technique involving creating a mass of units early in the game and attacking quickly, hoping to finish off the AI opponent while it is in its buildup phase); FTPS or platform games could evolve better ways of dealing with map features, racing games can evolve more efficient racers, fighting games could coevolve whole characters.

Platform

This is generally not an issue of concern for GAs because the work is mostly done offline. In fact, the optimization effect of a GA black box might actually improve its chances of being a viable consideration on CPU-limited platforms.

Development Limitations

For GAs, development matters are really the area of most concern. GAs are not debuggable in any real sense, so extra time must be allotted to tracking down small problems with the solutions. Will you also have enough time to actually evolve your solution, given that it might take a good amount of time to tune the GA (with different parameters, operators, and gene design) to get good results? Are the designers on your team going to require last minute tweaks or changes that could endanger a proven GA solution? Do you want the evolutionary portion of the product to keep evolving in the field, or are you going to disable that part of the process and lock in a solution? Is your product set up so that testing and feedback of the GA results is built into the pipeline from the beginning, so that you can get fast turnaround on GA solutions within the game? All these types of questions will require you consider the team you are working with in addition to the game you are working on.

Entertainment Limitations

Game-specific concerns such as difficulty settings would probably best be handled by separate GAs for each level of difficulty, with a separately tuned fitness function. Tuning and game balance can be difficult. The real power of GAs is when your game design specifically calls for somewhat varied or surprising AI behavior, so that the anomalies that may be present within your GA solution can be accounted for within the game universe.

TeamLRN

SUMMARY

Genetic algorithms are a fascinating way to solve or optimize difficult AI problems. They are easy to set up, but can be difficult to perfect, because of the numerous settings and usages. GAs can find novel solutions to game situations, which is a big goal of today's games.

- Evolution in nature uses genes as encoded rule sets. Pairs of organisms are chosen, largely by their performance within the environment, to reproduce and pass on their genetic material to their offspring. But this new generation undergoes genetic crossover and mutation, which can further optimize its fitness.
- GAs are usually used offline because evolving solutions is time consuming and requires many iterations before useful behavior begins to appear.
- GAs are stochastic methods and are considered a form of brute force search.
- They do not guarantee performance, or success.
- The basic algorithm can be stated as: (starting with a random initial population) run the population through a fitness function, then select favorable individuals to reproduce, apply a random mutation, and run this next generation. Keep doing this until the fitness of the individuals reaches some acceptable level.
- The gene and genome structure represent the solution to the problem you are trying to solve with the GA.
- The fitness function is the factor for which your GA is trying to optimize a solution. Its value can be used raw, or after some form of scaling to prevent or encourage data spread and clumping.
- Reproduction by the system culls out bad genes and helps promote good genes through selection. It also blends good individuals together through crossovers to help find optimal solutions, and mutates genes to keep solutions from being stalled in local maximum. There are many types of selection, crossover, and mutation operators.
- Implementing a GA into the test bed involved creating a new application that can be used for the evolving process, and creating the `GAAIControl` class, which handles the main algorithm.
- GAs are strong with problems that have many parameters related nonlinearly, have many local maxima, or involve heavily discontinuous output. They are well suited to complementing more traditional techniques and can be considered an optimization if replacing costly computational decision making.
- Evolution within GAs can be time consuming and provide hit or miss performance. The reasons behind success or failure are somewhat hidden, so they are hard to tune and debug.
- Extensions to the paradigm involve ant colony algorithms, coevolution, self-adapting GAs, and genetic programming.

21 Neural Networks

Neural networks (NNs, sometimes called artificial neural nets, because the original ones are in real brains) are an attempt by computer scientists to use lessons learned from biology in our AI solutions, somewhat like with genetic algorithms. But where GAs use survival of the fittest techniques to evolve a solution out of the possibilities, NNs strive to find solutions by using a method somewhat grounded in how the brain works, both organizationally and functionally. Although they don't do a very realistic job of modeling an actual brain, they do give us a very straightforward way of pattern matching and predicting trends in input data.

NEURAL NETS IN NATURE

Animals' brains are essentially a large cluster of interconnected nerve cells called *neurons*. The term "large cluster" is something of an understatement when dealing with some of the more intelligent creatures on the planet; human brains are composed of about 100 billion neurons, elephants have about 10 times that many. Each neuron has a number of connections (humans have about 10,000 connections *per* neuron) to other neurons, both coming in and going out. The connections coming in are called *dendrites*, and those going out are called *axons* (see Figure 21.1). Although considered connections, neurons don't technically connect, but rather the dendrites of one neuron come very close to the axons of other neurons (usually about 0.01 micron), and the space between them is called a *synaptic gap*, or synapse. Neurons are essentially electrical (although their conductivity, overall charge, capacitance, and other factors are due in some part to internal chemistry).

A heavily simplified description of what happens within a single neuron is that its large number of dendrites is transmitted electricity from nearby axons, which gradually builds up like a capacitor charge within the neuron. If this charge gets too large (above a certain *threshold*), it fires the collected energy (always a set amount, much like the on or off state of a computer bit) down its axon, in what is called an

TeamLRN

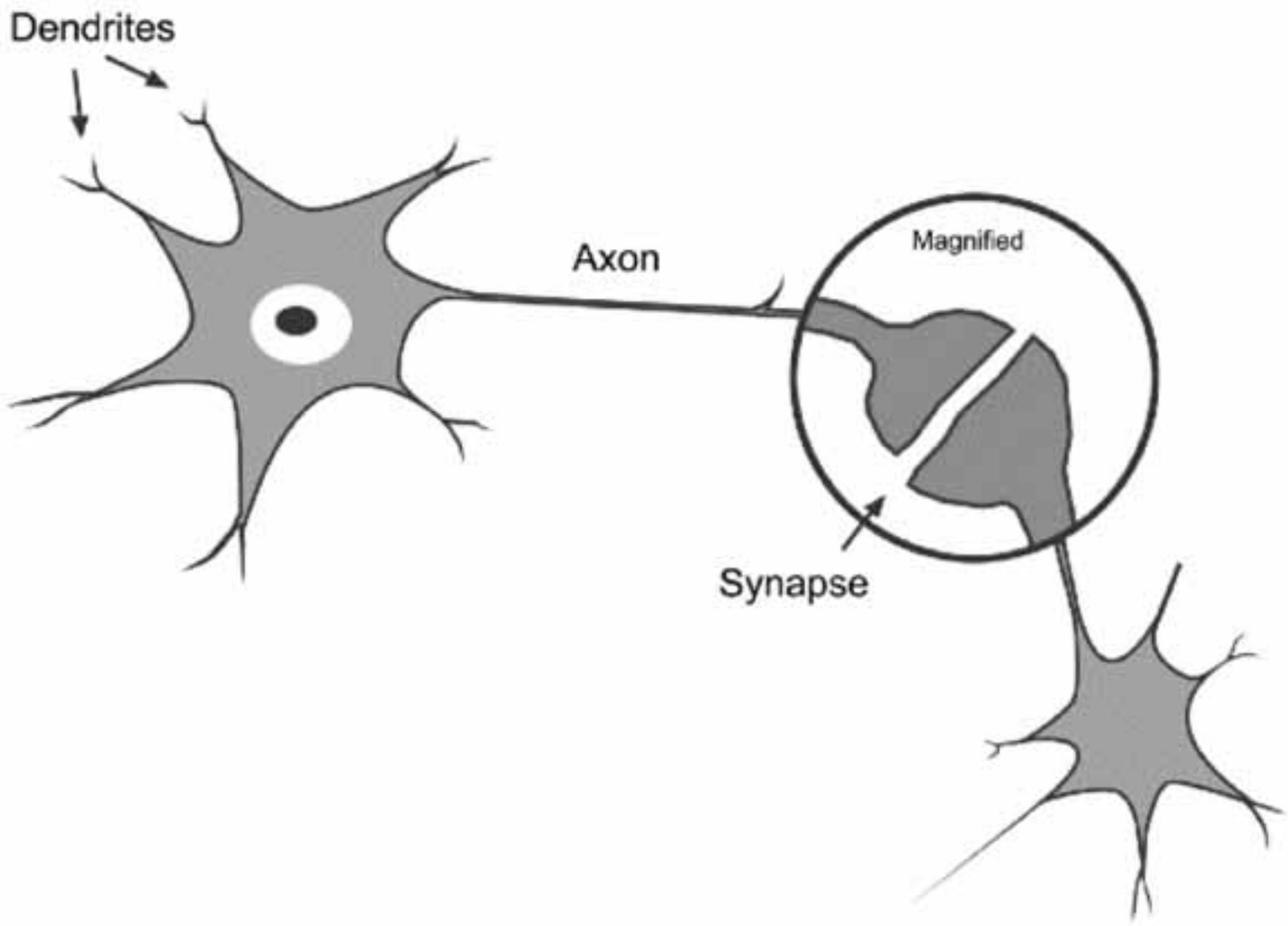


FIGURE 21.1 Basic diagram showing parts of a neuron.

action potential, where it may be transmitted to the dendrites of other neurons. If a particular neuron fires often enough, this will bring about small, biological changes within the neuron (such as a decrease in the electrical resistance along the dendrites and axon, an increased sensitivity to charge at the synapses, even the size of the nerve fibers between various points), causing the electricity necessary to fire its potential lessen. In effect, the neuron has "learned" that it usually requires firing and will do so with less electrical resistance, rather than waiting for the entire charge to build up. The opposite effect can occur as well, where a particular neuron almost never fires and thus, is subject to atrophy. Although this obviously gives us a biological notion of learning through anticipation, it also establishes the concept of pattern recognition at a cellular level.

Another concept that occurs between neurons is that of exhibition and inhibition. A particular neuron could be said to be inhibitory to another if it deadens the electrical charge that reaches another neuron, or exhibitory for the opposite effect. This isn't the same as the synaptic changes within the cell, because it is not connection specific. All connections coming into a particular neuron would be inhibited if the neuron in question were biased in such a way, whereas each individual synapse coming into the cell would have to atrophy for it to be inhibitive synaptically.

In essence, the brains of animals work by taking input, recognizing patterns within the input, and making decisions based on those patterns, which is precisely what we want to emulate with a NN in our software.

We are also trying to take advantage of the parallelism that the connectivity within the brain apparently gives our problem-solving ability. The human brain operates at roughly 100Hz, a fraction of the speed of modern computers. But although computers are dealing with one instruction at a time (or possibly a few, given multiprocessor systems), the human brain can perform millions of instructions at once. Because of the symbolic way that our brains store knowledge and solve problems, we can mentally employ many levels of efficiency that allow us to use a tremendous amount of parallel processing. Obviously, unless you are using a parallel processing CPU, you will not be able to emulate actual human parallelism. But the hope is to employ the many parallel levels of correlation that can be encoded into a NN that would otherwise be difficult or impossible to find otherwise.

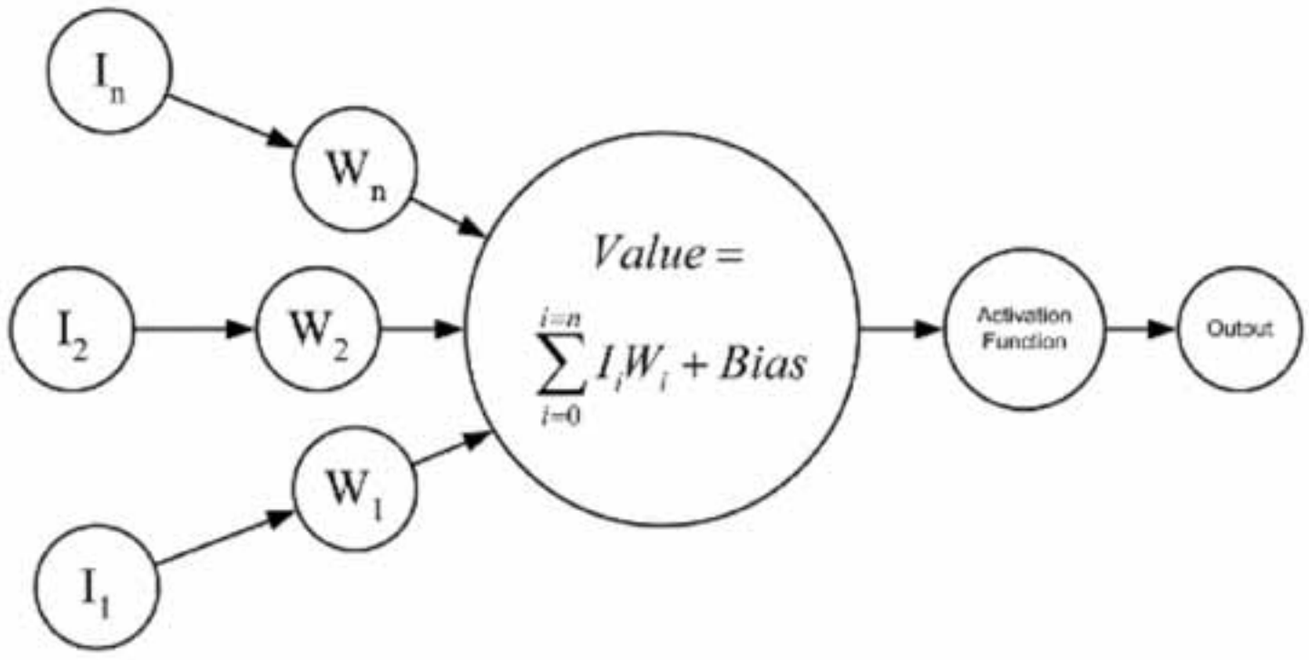
ARTIFICIAL NEURAL NETS OVERVIEW

Figure 21.2 shows the parts of an artificial neuron and a basic NN overview diagram. Note that the value associated with a neuron is the sum of all the input values multiplied by their connection weights, added to the neuron's bias value. Bias refers to the inhibitory or exhibitory effect the neuron has within the network. The "axon" on the neuron is represented by its output value, optionally filtered through an activation function, which will be discussed under later, in the section "Using a Neural Net."

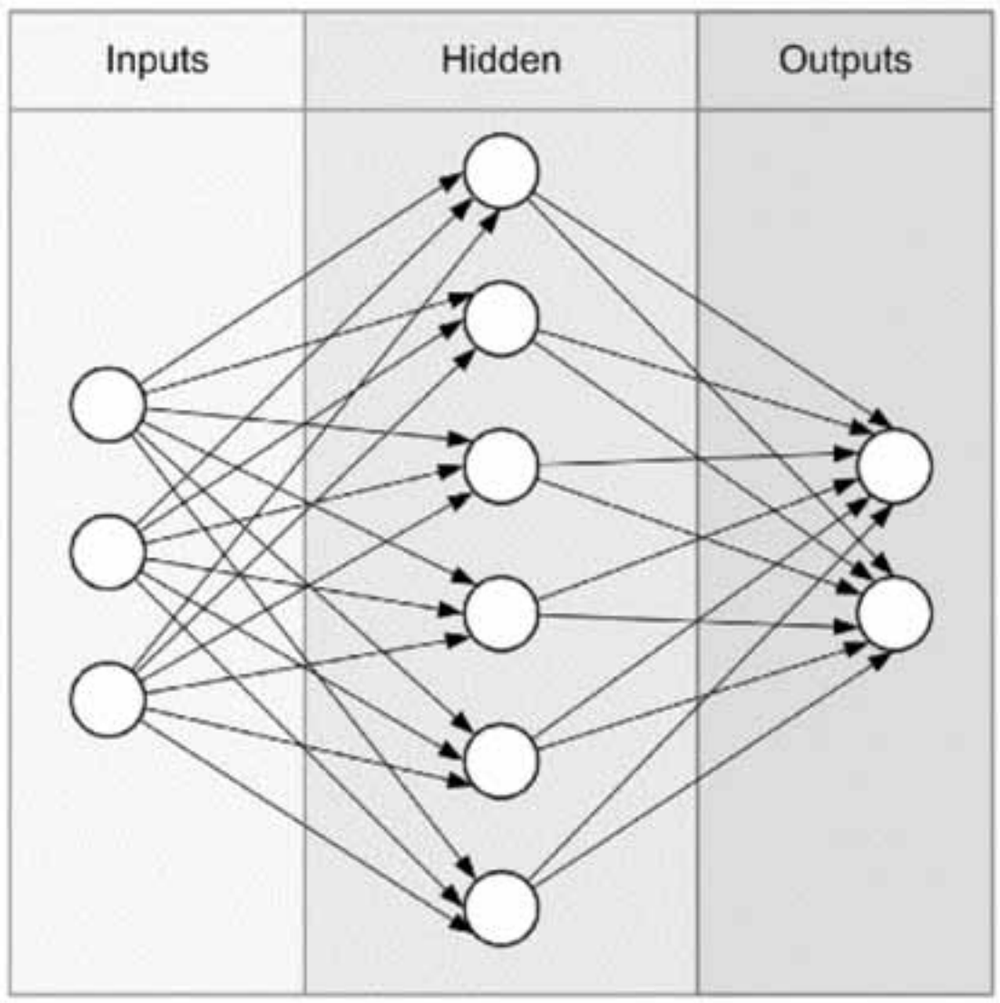
Within the overview diagram, the circles are the neurons (or *nodes*, as they are sometimes called when talking about the artificial ones), and the lines between them represent connections between neurons. The nodes in column one of the figure are the parts of what is called the *input layer*. The nodes represent entry points into the network; places where outside inputs come in to be classified by the NN. The second column encompasses the *hidden layer*, which represents internal data storage for the network. These nodes are useful in that they give the network room to grow, but also give the network greater ability to handle larger variation in patterns. The hidden layer may comprise one set of nodes, as shown, or multiple sets, to whatever complexity you are willing to work with. A special case is when you have no hidden layer at all, with the inputs directly mapping to the outputs, which is then called a *perceptron*. These are very low-functioning NNs, but they can still be used to do some linear pattern recognition. The third column is called the *output layer*, and it corresponds to the actual categories that the network is trying to impose on its inputs.

Also notice the actual connections themselves in the diagram. Each connection has an associated value, and a direction. The value represents the *weight* associated to the link, and is biologically equal to the strength of the connection between two neurons. As for direction, the NN shown in Figure 21.2 is an example of a feed forward (FF) network because each layer only propagates forward into the network.

TeamLRN



Artificial Neuron



Neural Net

FIGURE 21.2 An artificial neuron and a basic neural network diagram.

Another type of NN, which doesn't have this restriction, is called a recurrent network. In these NNs, information can go from input to output, and back again, allowing for feedback within the system. To facilitate this, recurrent networks have a number of state variables associated with them and are thus a bit more complex than feed forward NNs. In games, AI programmers almost universally deal with FF

systems because they are easier to understand, more straightforward to tune, and less expensive to run (because the feedback phase requires data to be run through the network multiple times). Recurrent networks are technically more capable than FF systems, but you can generalize FF nets so easily that many of the benefits of recurrent systems can be gained by running multiple FF nets instead.

One last property of NNs is the amount of connectivity they exhibit. The diagram shows a fully connected NN, because each node is connected to every node in the next layer. If there were some nodes that weren't following this rule, for whatever reason, the NN would be called *sparsely* connected. Although building a sparsely connected node isn't much more difficult than the far more common fully connected version, they tend to slow down the performance of the system, and a good NN will determine that a connection is unnecessary and adjust the weight of the connection accordingly. Thus, sparsely connected NNs are not commonly used.

In the business world, NNs have successfully infiltrated many different industries. One of the first large scale successes with NNs was the United States Postal system, which uses a heavily trained NN for handwriting recognition when reading the addresses on mail. Other uses include trying to predict the weather, judging credit card fraud, voice recognition, diagnosis of diseases or other health problems, artificial vision techniques, or even filtering of Internet sites against pornography or other graphic material.

Games have the same kinds of problems as the rest of the world, so NNs have been used for a number of the same sorts of issues dealing with pattern recognition or prediction. Anytime you can identify a pattern within a system, it logically follows that you should be able to use the pattern to directly make a decision about the system, determine what kinds of decisions another is making within the system, or use previously stored data to try and predict what's going to happen in the future. All three of these are useful in the world of game AI. At the basic level, NNs can be trained to become a black box for potentially expensive operations like animation selection (which dunk animation should the basketball player perform right now, given the state of the game, his skill, the surrounding players, the point spread, the difficulty level of the game, etc.), which is roughly analogous to using a pattern to directly make a decision. The pattern recognition gleaned from a suitable NN could be used to form the basis of a player modeling system, to keep the AI on top of the human player by being able to predict what the human will do. Finally, although uncommon in games, a NN could be used to "store" information, by allowing the learning element to continue to run during live gameplay, thus allowing the NN to potentially *learn* adaptive techniques. The reason this is uncommon is because of the unpredictable, as well as unstable, nature of the learning that NNs use. Some systems allow for this, but restrict the areas of learning severely to try to minimize the random element into the game world. *Black & White* would have suffered

TeamLRN

greatly if people's creatures suddenly exhibited what is called *catastrophic unlearning*, and couldn't perform any tasks at all after taking in a piece of knowledge that effectively unraveled the entirety of the relationships stored within the network.

USING A NEURAL NET

The basic steps for implementing a NN system within your game is to set up your network, train it using specially prepared data that is treated as inputs, and then actually use it on live game inputs. The first step, designing the NN architecture for your game problem, requires that you consider several factors, including structure, learning, and training data:

Structure

Structure refers to both the type (FF, recurrent) and organization (how many nodes, how many hidden layers) of the NN to be constructed. Most people stick with FF networks because some level of feedback can be built into an FF net, and they are much cheaper, performance wise.

The number of variables you want the NN to categorize or pattern match on determines the number of input nodes in your NN. A NN might only have one input, in effect asking, "What is this, or What should I do with this?" But it might also have several pieces of information that it needs to make a decision. Try and pare the number of inputs to the most essential because any additional elements you add here will translate to a much larger state space for your NN to search though. You are pretty much asking your system to find a pattern that links every one of your inputs together. So with two inputs, your NN only has to find a "line" that connects them, but with twelve inputs, you ask your NN to find the nearest "dodecahedron" that fits nicely on your data points. Not such an easy task. Note that abstract variables, that represent combinations or calculations based on simpler variables, tend to be better suited to NNs. So, in our test bed, a variable called "danger" might be better than a bunch of inputs about the closest few asteroids positions, speeds, and so on.

There is only one basic rule about the number of nodes within the NN: the fewer you can get away with, the better. Again, the more nodes you include in a NN, the larger the search space becomes that the NN is slogging through to arrive at a suitable solution.

There are no real guidelines about how many hidden nodes you require (although one hidden *layer* seems to be fine for most of the problems that game AIs come up with). A common practice is to use a medium number of hidden nodes (two times the number of input nodes) and then go up or down a few and compare

the performance until you see it tapering off. Many sources will state guidelines for the number of hidden nodes, or even give rules "that aren't to be broken." But most of this information is useless, mostly because these sources determine these rules based on the number of input and output nodes, and don't take into account essential factors like the number of training cases, the complexity of the function being solved, or the amount of noise (variance) in the outputs.

The number of output nodes is equal to the number of outputs that you require from the NN. Are you building a system that tells you whether or not you can see the game hero? Then your NN will only need two output nodes: Yes and No. Building a character recognition system that can recognize all the numbers? Then you'll need 10 output nodes, one each for the numbers 0–9.

Know that each output doesn't have to be binary; it can have continuous values of activation. Thus, your output neurons could be "Turn Left" and "Turn Right," and the level of the neuron's activation would tell you how much to turn. Smooth activations are achieved by using a suitable *activation function.* Some of the common activation function types include the step function, the hyperbolic tangent and logistic sigmoid functions, and the Gaussian function.

Shaping the output is the not the only reason for using an activation function on the final value of a given neuron. Using activation functions on the hidden nodes is also done for an entirely different reason. One of the most powerful capabilities of NNs is to encapsulate a nonlinear function that maps the inputs to the outputs. However, it can only do this if the NN itself can represent a nonlinear function. Without a hidden layer, a perceptron is only capable of finding linear correlations between the inputs and outputs. But adding a hidden layer to a perceptron isn't enough; we must also use a nonlinear activation function on the nodes to give an element of nonlinearity to the network connections. Almost any nonlinear function will do, except polynomials. For backpropagation learning (which will be discussed later), the activation function must be differentiable, and it helps if the function is bounded, hence the choices for the common activation functions.

Learning Mechanism

Once you have set up your NN, you need to determine how you want to train it. There are two main types of NN learning: the aptly named supervised and unsupervised. Supervised learning involves having training data that consists of input-output pairs. You feed the input into the NN, and then adjust the weights of the network if there is a discrepancy between the output from the NN and the expected output given in the training data. Training continues until a certain level of accuracy is achieved. The name of this method is *backpropagation* because the way you adjust the network parameters is from the back to the front. Another form of supervised learning is called *reinforcement* learning. In this system, desired outputs are

TeamLRN

not given to the algorithm, but the network is rewarded (or its behavior is reinforced) when it performs well. Some implementations also punish when the system performs poorly, but this is usually overkill.

Unsupervised learning involves statistically looking at the output and adjusting the weights accordingly. One technique for this is called *perturbation learning*, which is very similar to an academic AI technique called *simulated annealing*. In perturbation, you test the NN, then adjust some of the values a small amount, and try it again. If you get better performance, you keep going by repeating the process; otherwise, you go back to your last net settings. Another technique that is becoming quite common is to use a genetic algorithm to adjust the weight values of your NN. The relationship between the two methods really shows each technique's strengths: the NN is determining the pattern between inputs and outputs, whereas the GA is optimizing a set of numbers to maximize some fitness function.

Creating Training Data

Now you have your network, and you know how to train it. If you have chosen to use supervised learning, your next job is to actually acquire the test data that you will use in training the NN. There are several ways to do this.

You could actually record a human performing the same kinds of tasks you are trying to teach your system, and then create a bunch of test cases based on his behavior. This kind of training data is great because you can also use it to build human-level performance into the AI (because you can use data points when the human didn't do the right thing, or was subtly wrong), which will help your AI to seem not as robotic as it might if it were using a hardcoded algorithm. But, this method is extremely time consuming, and your AI's skill is then limited to the skill of the person being mimicked.

Another way is to write a separate program to generate reasonable input scenarios and have a human say which output should come up. This is fine (although again, very time consuming for the person involved) for binary or discrete output values, but is futile with real value or numerous outputs. You could generate random input and output pairs, and check them for validity, storing them only if you get winners. Or, you could use some kind of expert knowledge about the problem to try to generate some training data points. This might be hard, considering that the reason you're using a NN in the first place is because you might not have this kind of data.

The number of training cases required depends on the amount of noise in the targets and the complexity of the function you are trying to learn, but as a starting point, it's a good idea to have at least 10 times as many training cases as input units. This may not be enough for highly complex functions. For classification problems, the number of cases in the smallest class should be at least several times the number of input units. Optimally, you should strive for a training suite more along the lines of 10N, where N is the number of inputs.

AN ASIDE ON NEURAL NETWORK ACTIVITY

The pattern recognition that NNs are capable of should really be understood to truly see why they do what they do. A good understanding of the process can also help in debugging or perfecting the performance of an NN that is acting up. The two primary (and somewhat similar) tasks that NNs are good at performing are *regression* and *classification.* Figure 21.3 displays an example of regression; Figure 21.4 shows a few classification cases.

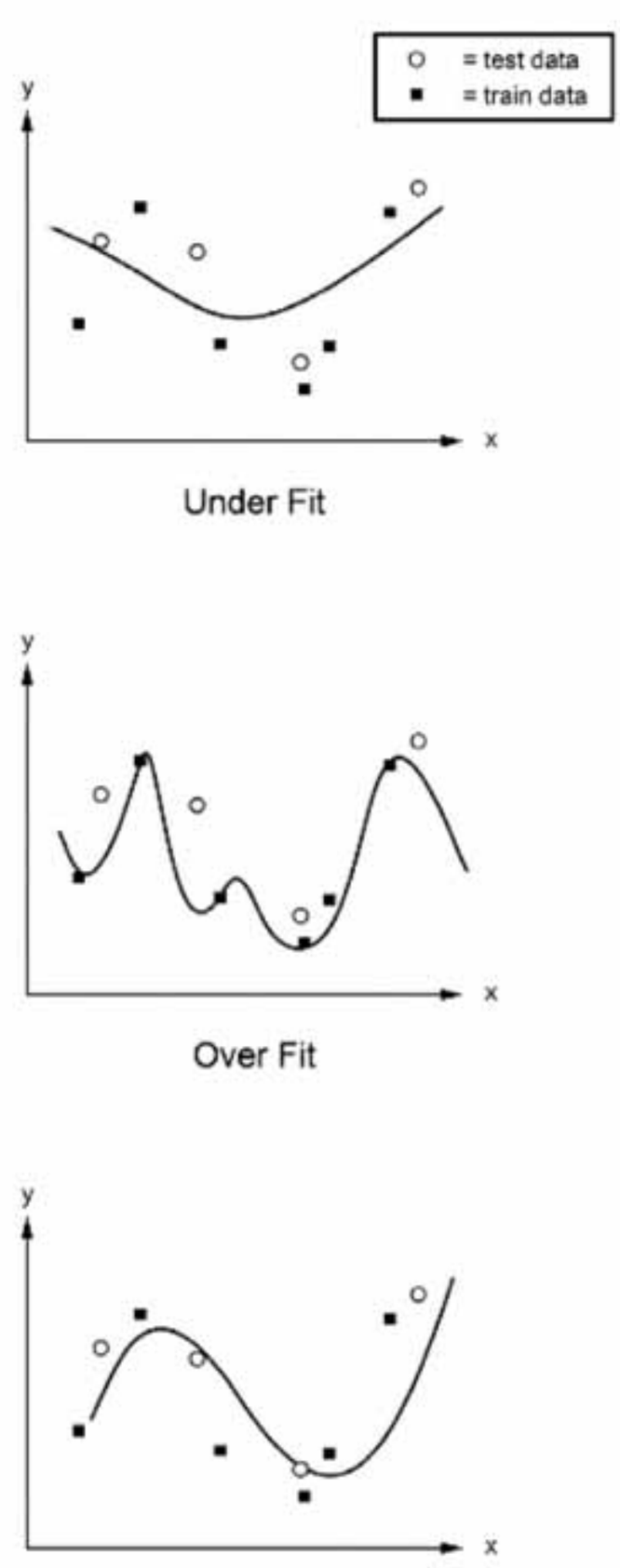


FIGURE 21.3 Examples of regression.

TeamLRN

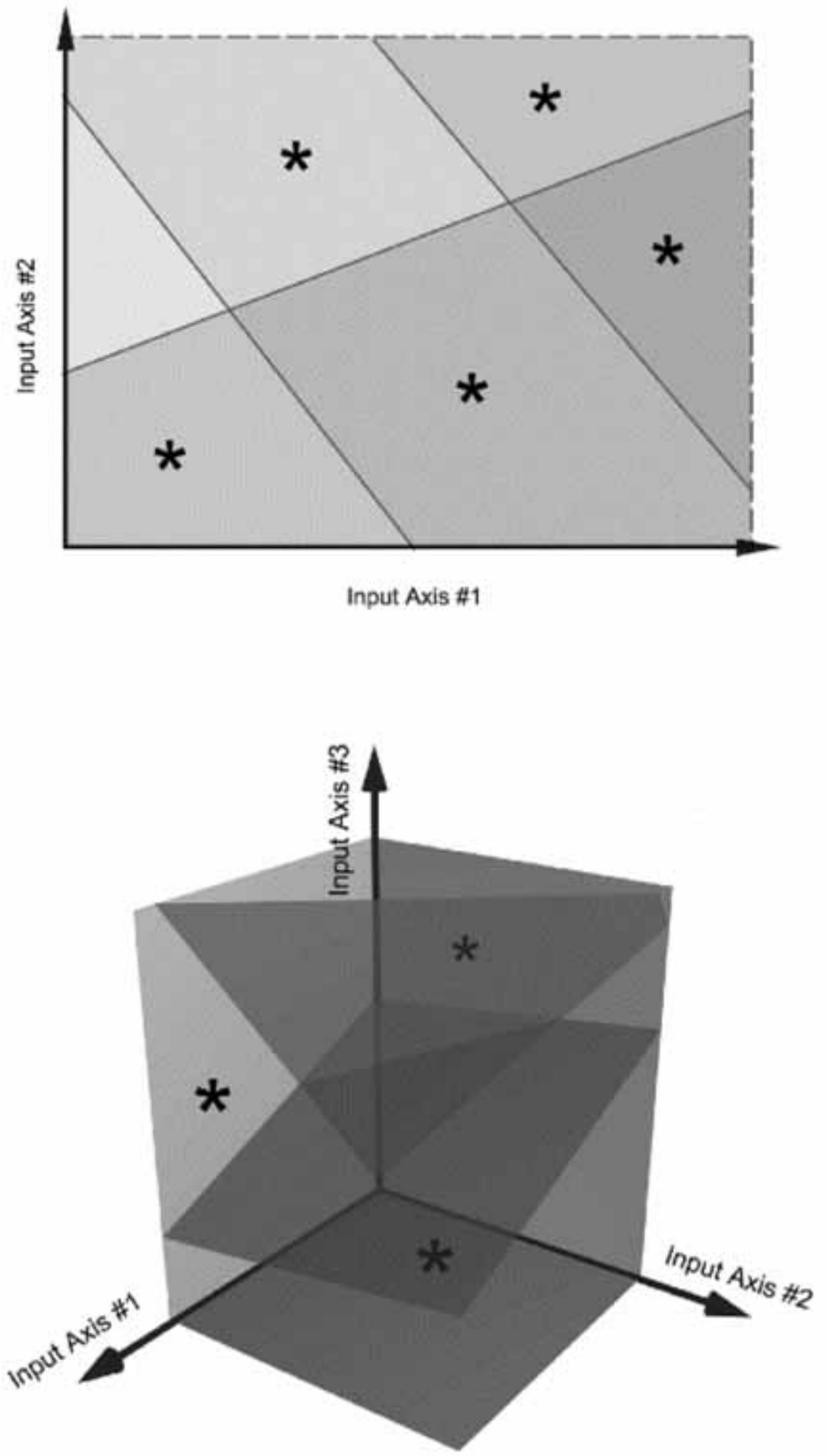


FIGURE 21.4 A two-input and three-input NN search space, displaying classification lines.

Regression is defined as finding a function that fits all the data points within some tolerance. Say you're going to create an NN to help your AI enemy evade bullets shot by the player by sidestepping out of the way. You input the enemy's facing direction, the position of the player, and the position of the enemy. Assuming that the bullets are going to head directly from the player to the enemy, the NN will determine a movement vector for the enemy. What is the NN really learning in this

example? If you were to solve this problem algorithmically, you would calculate the vector between the two, and then take the dot product of that vector with a unit vector built from the enemy's facing angle, giving you the angle you need to turn from your current facing. But left or right? You need to perform the same operation again, but with a unit vector perpendicular to the enemy's facing. If you combine these mathematic operations into one large function, then you would have the exact function that the NN has to learn to solve your problem. The NN, in essence, is *learning* how to perform the right dot products and comparisons.

If you take this visual a step further, it will give you a hint about the structure requirements of your nets. Imagine that all your NN's outputs are merely a *linear* function of your inputs. Say there was a smooth, ramplike hill in your game. An enemy on this hill knows how far he has traveled along the hill, but wants to know his current altitude. If you coded a NN to solve this conundrum, it would very quickly find the solution, and it could do so with no hidden nodes. That's because the function it would be finding is the slope of the hill, a linear function of the input. But, if your game hill was like a real hill, and had ever changing slopes with valleys, plateaus, pits, and other features, then the NN would have to find a non-linear equation (the equivalent of some form of complex Fourier transform, or the like) to approximate the "function" that the altitude is following. To do that, it would need plenty of hidden layers with nonlinear activation functions to store this kind of information.

Another benefit that this visualization helps with is estimating or debugging the amount of training on your NNs. In the bottom graph in Figure 21.3, you can see how the line follows the trend of the data well. This NN has been trained the right amount. It does a good job of determining the spirit of the data, without underfitting (the function is too smooth, and misses key variation trends within the data) or overfitting (the regression function isn't smooth, it takes into account noise in the data, and thus has somewhat surprising results). Realizing what is occurring within your network is the first step toward being able to build NNs without having to spend a large amount of your time experimenting to get them to operate smoothly.

The other, similar task that NNs thrive on is classification. If you are given a pile of buttons, and asked to separate them by color, you would push them into piles of each color represented in the group. You would *classify* them. When you give inputs to a NN, what you're asking it to do is categorize the inputs into the number of piles of output nodes. But the NN is dealing with an entire search space instead of individual objects (by giving an NN two inputs, you're not giving it two distinct numbers, you're giving it two *axes* of inputs). It won't be dividing objects into piles. Instead, the output nodes represent lines of separation within the state space of input possibility. Visualizing a classification NN in this way is very powerful and useful. Think of each input as a theoretical axis in a graph, and each output as a line (or plane, or hyperplane, depending on the dimensionality of your system)

TeamLRN

separating distinct inputs into isolated categories. Figure 21.4 shows two examples, with 2 and 3 NN inputs.

When you think of your NN in this way, it becomes easy to see a number of aspects of NN behavior. Incorporating unnecessary input nodes makes it harder for the NN to solve the problem, by making the search space exponentially larger in dimensionality. Additional output nodes make for tighter categories, but also make the job of differentiation that much more complex. Although this is a simplified view of the internal workings of an NN, it is helpful in picturing the effect.

IMPLEMENTING A NEURAL NET WITHIN THE AISTEROIDS TEST BED

First, we must determine what it is that we want an NN to do for our test bed application. Although nothing in our simple game is screaming for an NN solution, we can definitely create a suitably difficult problem to see how well the NN handles solving the problem. For this chapter, we will once again attack the problem of asteroid avoidance. We'll strive to *teach* the ship how to avoid the asteroids. To do this, we can first create the necessary training data by recording a human player perform in the game. Then, we'll use this data to train the NN. Finally, we can start a fresh game, load in the trained NN data, and use it to perform the correct avoidance behaviors.

In our example, we'll be using a fairly simple NN, with one hidden layer with eight neurons, four inputs, and three outputs.

The inputs we'll be using are the following:

- Two inputs, which will be the X and Y components of the vector between the ship and the nearest asteroid.
- The speed that the two objects are moving together, which is determined by taking the moving velocity of each object and finding the component of velocity that lies along the direct path to the other object.
- The ship's moving direction, which gives the NN a frame of reference with which to make correlations between the other inputs.

The outputs that the system will provide are simply Boolean values for the simple ship's controls. They will determine whether or not the ship should thrust, turn left, and turn right.

The NN system implementation will comprise four main parts:

- A `Neuron` is the basic element of an NN. A neuron structure stores several data fields, including the weights of each incoming connection to the neuron, the

output value, and the error gradient computed from the expected outputs during training.

- The `NLayer` is a set of neurons that constitute a particular layer in the network. At this level, various operations can be performed on the inherent neurons, such as propagation (feeding the inputs forward through the network), backpropagation (calculating the error gradients at each neuron backward through the net), and steepest descent adjustment of the connection weights. The various kinds of activation functions are also found at this level.
- The `NeuralNet` class is the main interface for the network. All the functions are necessary to actually run and train the net are here.
- `NNAIControl` is the `Controller` class we'll be using for our networks. It provides a location for game specific usage of an NN. Our controller will be set up to handle a few different "modes" of control, namely *training* of the network, versus actually *using* the network to perform AI tasks once it has been trained.

The `NeuralNet` Class

To use an NN within a game, we will need to *construct* the actual network structure, *train* the network with a set of data, and then *use* the network to determine what to do with new data.

Listing 21.1 shows the header for the `NeuralNet` class; Listing 21.2 shows the implementation.

LISTING 21.1 `NeuralNet` Header

```
class NeuralNet
{

public:

    NeuralNet(int nIns,int nOuts,int nHiddenLays,int
nNodesinHiddenLays);
    void Init();

    //access methods
    void Use(vector<float> &inputs,vector<float> &outputs);
    void Train(vector<float> &inputs,vector<float> &outputs);
    float GetError()    {return m_error;}
    void WriteWeights();
    void ReadWeights();
```

TeamLRN

```
protected:
    //internal functions
    void AddLayer(int nNeurons,int nInputs,int type);
    void SetInputs(vector<float>& inputs);
    void FindError(vector<float>& outputs);
    void Propagate();
    void BackPropagate();

    //data
    vector<NLayer>  m_layers;
    NLayer*         m_inputLayer;
    NLayer*         m_outputLayer;

    float           m_learningRate;
    float           m_momentum;
    float           m_error;

    int             m_nInputs;
    int             m_nOutputs;
    int             m_nLayers;
    int             m_nHiddenNodesperLayer;
    int             m_actType;
    int             m_outputActType;
};
```

The `Init()` function is the primary set-up function for the network. It builds the internal structure of the net, by iteratively calling `AddLayer()` to instantiate each layer's neurons. The system is set up to handle simple nets with only an input and output layer (perceptrons) as well as general, multilayer NNs.

`Propagate()` takes the inputs to the net and spreads their influence forward through the network. `BackPropagate()` effectively reverses this operation by taking the error of the final outputs and finding the correct error gradients throughout the network, from the last layer backward to the first.

`Train()` and `Use()` are the two main functions for actually using the NN. During training, you call the `Train()`method with the input-output pair you want to train. It then propagates the inputs through the NN, finds the error from the expected outputs, and backpropagates that error. `Use()` assumes a trained net. It just takes the inputs and returns the network's outputs.

`FindError()` determines the output error of the network from given outputs during training. Using the derivative of the activation function, it determines the error gradient for each output neuron, which will then be used to backpropagate the necessary changes to the connection weights within the network, to close in on the optimal weights to perform well.

LISTING 21.2 NeuralNet Implementations

```
//--------------------
void NeuralNet::Init()
{
    m_inputLayer   = NULL;
    m_outputLayer  = NULL;
    m_actType      = ACT_BIPOLAR;
    m_outputActType= ACT_LOGISTIC;
    m_momentum     = 0.9f;
    m_learningRate = 0.1f;

    //error check
    if(m_nLayers<2)
        return;

    //clear out the layers, incase you're restarting the net
    m_layers.clear();

    //input layer
    AddLayer(m_nInputs, 1, NLT_INPUT);

    if(m_nLayers > 2)//multilayer network
    {
        //first hidden layer connect back to inputs
        AddLayer(m_nHiddenNodesperLayer, m_nInputs, NLT_HIDDEN);

        //any other hidden layers connect to other hidden outputs
        //-3 since the first layer was the inputs,
        //the second (connected to inputs) was initialized above,
        //and the last one (connect to outputs) will be initialized
        //below
        for (int i=0; i<m_nLayers-3; ++i)
            AddLayer(m_nHiddenNodesperLayer, m_nHiddenNodesperLayer,
                     NLT_HIDDEN);

        //the output layer also connects to hidden outputs
        AddLayer(m_nOutputs, m_nHiddenNodesperLayer, NLT_OUTPUT);
    }
    else//perceptron
    {
        //output layer connects to inputs
        AddLayer(m_nOutputs, m_nInputs, NLT_OUTPUT);
    }
```

TeamLRN

```
    m_inputLayer = &m_layers[0];
    m_outputLayer= &m_layers[m_nLayers-1];
}

//--------------------
void NeuralNet::Propagate()
{
    for (int i=0; i<m_nLayers-1; ++i)
    {
        int type = (m_layers[i+1].m_type == NLT_OUTPUT)?
                        m_outputActType : m_actType;
        m_layers[i].Propagate(type,m_layers[i+1]);
    }
}

//--------------------
void NeuralNet::BackPropagate()
{
    //backprop the error
    for (int i=m_nLayers-1; i>0; --i)
        m_layers[i].BackPropagate(m_actType,m_layers[i-1]);

    //adjust the weights
    for (i=1; i<m_nLayers; i++)
        m_layers[i].AdjustWeights(m_layers[i-1],
                        m_learningRate,m_momentum);
}

//--------------------
void NeuralNet::Train(vector<float> &inputs,vector<float> &outputs)
{
    SetInputs(inputs);
    Propagate();
    FindError(outputs);
    BackPropagate();
}

//--------------------
void NeuralNet::Use(vector<float> &inputs,vector<float> &outputs)
{
    SetInputs(inputs);
    Propagate();
    outputs.clear();
```

```
    //return the net outputs
    for(int i =0;i< m_outputLayer->m_neurons.size();++i)
        outputs.push_back(m_outputLayer->m_neurons[i]->m_output);
}

//--------------------
void NeuralNet::SetInputs(vector<float>& inputs)
{
    int numNeurons = m_inputLayer->m_neurons.size();
    for (int i = 0; i<numNeurons; ++i)
        m_inputLayer->m_neurons[i]->m_output = inputs[i];
}
//--------------------
void NeuralNet::FindError(vector<float>& outputs)
{
    m_error = 0;
    int numNeurons = m_outputLayer->m_neurons.size();
    for (int i=0; i<numNeurons; ++i)
    {
        float outputVal = m_outputLayer->m_neurons[i]->m_output;
        float error = outputs[i]-outputVal;
        switch(m_actType)
        {
        case ACT_TANH:
            m_outputLayer->m_neurons[i]->m_error = m_outputLayer->
                                                InvTanh(outputVal)*error;
            break;
                case ACT_BIPOLAR:
            m_outputLayer->m_neurons[i]->m_error = m_outputLayer->
                                            InvBipolarSigmoid(outputVal)*error;
            break;

        case ACT_LOGISTIC:
        default:
            m_outputLayer->m_neurons[i]->m_error = m_outputLayer->

InvLogistic(outputVal)*error;
            break;
        }
        //error calculation for the entire net
        m_error += 0.5*error*error;
    }
}
```

TeamLRN

The NLayer Class

Because most operations on nets are on the connections from one layer to another, this is the real workhorse of the system. Listing 21.3 shows the header for the NLayer class, and Listing 21.4 shows the implementation.

LISTING 21.3 NLayer Header

```
class NLayer
{
public:
    NLayer(int nNeurons, int nInputs, int type = NLT_INPUT);
    void Propagate(int type, NLayer& nextLayer);
    void BackPropagate(int type, NLayer& nextLayer);
    void AdjustWeights(NLayer& inputs,float lrate = 0.1f,
                       float momentum = 0.9f);

    //activation functions
    float ActLogistic(float value);
    float ActStep(float value);
    float ActTanh(float value);
    float ActBipolarSigmoid(float value);
    void  ActSoftmax(NLayer& outputs);

    //derivative functions for backprop
    float DerLogistic(float value);
    float DerTanh(float value);
    float DerBipolarSigmoid(float value);

    //data
    vector<Neuron*> m_neurons;
    int             m_type;
    float           m_threshold;
};
```

LISTING 21.4 Important NLayer Implementations

```
//--------------------
void NLayer::Propagate(int type, NLayer& nextLayer)
{
    int weightIndex;
    int numNeurons = nextLayer.m_neurons.size();
    for (int i=0; i<numNeurons; ++i)
```

```
{
    weightIndex = 0;
    float value = 0.0f;

    int numWeights = m_neurons.size();
    for (int j=0; j<numWeights; ++j)
    {
        //sum the (weights * inputs), the inputs
        //are the outputs of the prop layer
        value += nextLayer.m_neurons[i]->m_weights[j] *
                 m_neurons[j]->m_output;
    }

    //add in the bias (always has an input of -1)
    value+=nextLayer.m_neurons[i]->m_weights[numWeights]*-1.0f;

    //store the outputs, but run activation first
    switch(type)
    {
        case ACT_STEP:
            nextLayer.m_neurons[i]->m_output = ActStep(value);
            break;
        case ACT_TANH:
            nextLayer.m_neurons[i]->m_output = ActTanh(value);
            break;
        case ACT_LOGISTIC:
        nextLayer.m_neurons[i]->m_output = ActLogistic(value);
            break;
        case ACT_BIPOLAR:
            nextLayer.m_neurons[i]->m_output =
                                            ActBipolarSigmoid(value);
            break;
        case ACT_LINEAR:
        default:
            nextLayer.m_neurons[i]->m_output = value;
            break;
    }
}
//if you wanted to run the Softmax activation function, you
//would do it here, since it needs all the output values
//if you pushed all the outputs into a vector, you could...
//uncomment the following line to use SoftMax activation
```

TeamLRN

```
        //outputs = ActSoftmax(outputs);
        //and then put the outputs back into the correct spots

        return;
    }

    //--------------------
    void NLayer::BackPropagate(int type, NLayer &nextLayer)
    {
        float outputVal, error;
        int numNeurons = nextLayer.m_neurons.size();
        for (int i=0; i<numNeurons; ++i)
        {
            outputVal = nextLayer.m_neurons[i]->m_output;
            error = 0;
            for (int j=0; j<m_neurons.size(); ++j)
                error+=m_neurons[j]->m_weights[i]*m_neurons[j]->m_error;
            switch(type)
            {
                case ACT_TANH:
                    nextLayer.m_neurons[i]->m_error =
                                            DerTanh(outputVal)*error;
                    break;
                case ACT_LOGISTIC:
                    nextLayer.m_neurons[i]->m_error =
                                        DerLogistic(outputVal)*error;
                    break;
                case ACT_BIPOLAR:
                    nextLayer.m_neurons[i]->m_error =
                                    DerBipolarSigmoid(outputVal)*error;
                    break;
                case ACT_LINEAR:
                default:
                    nextLayer.m_neurons[i]->m_error = outputVal*error;
                        break;
            }
        }
    }

    //--------------------
    void NLayer::AdjustWeights(NLayer& inputs,float lrate,
                                        float momentum)
    {
        for (int i=0; i<m_neurons.size(); ++i)
```

```
{
    int numWeights = m_neurons[i]->m_weights.size();
    for (int j=0; j<numWeights; ++j)
    {
        //bias weight always uses -1 output value
        float output = (j==numWeights-1)? -1 :
                              inputs.m_neurons[j]->m_output;
        float error  = m_neurons[i]->m_error;
        float delta  = momentum*m_neurons[i]->m_lastDelta[j] +
                       (1-momentum)*lrate * error * output;
        m_neurons[i]->m_weights[j]   += delta;
        m_neurons[i]->m_lastDelta[j]  = delta;
    }
  }
}
```

The class houses the activation functions and their derivatives. Also, each layer has a list of its constituent neurons, as well as an `m_type` field (is this an input, hidden, or output layer?), and a threshold value (which is normally set to 1.0f, this value represents the output value the neuron must accumulate to fire if using a simple step activation function, or the *gain* of the sigmoid function being used, which corresponds to the smoothness of the *s* shape in the output graph: very small values approach a flat line, and very large values approach a step function shape).

`Propagate()` is the layer extension to the function with the same name at the net level. It cycles through all the neurons in the level and performs the standard NN formula: sum all the inputs to the neuron, multiply by the corresponding connection weights, and then run it through the specified activation function.

`BackPropagate()` is also the layer-specific continuation of this operation. It sums the total weight on each neuron, and then calculates the gradient by multiplying it with the output value, after having run the output through the derivative of the activation function. Several activation functions have been supplied. The standard logistic function gives values between 0 and 1. Both the `tanh` and `bipolar sigmoid` functions give values from –1 to 1. The linear function is the equivalent of *no* activation function, meaning that the output isn't scaled at all.

`AdjustWeights()` performs steepest descent method on the weights because we've computed a gradient of the delta we're looking for. Steepest descent is a greedy algorithm, meaning that it gets stuck in local `minima` very easily, so care must be taken with this method. Hence, we're using momentum within our weight adjustment, which just means that adjustments have to come more frequently to make large changes because earlier changes have a much larger priority associated with them. This helps with the steepest descent getting stuck, but it does make training slower, so you will want to adjust the momentum value.

TeamLRN

The NNAIControl Class

The NNAIControl class will serve as the AI controller for the neural network technique. This class houses the network itself and the technique-specific usage code that links it to the AIsteroids game proper. As you can see in the header (Listing 21.5; Listing 21.6 shows some of the important function implementations), this class stores all the usual controller information (perception data and update methods, as well as being inherited from the FSMAIControl class so that it can also deal with the states of the AI ship), but also contains all the data and functionality for training and using the NN.

LISTING 21.5 NNAIControl Class Header

```
class NNAIControl: public FSMAIControl
{
public:
    //constructor/functions
    NNAIControl(Ship* ship = NULL);
    ~NNAIControl();
    void Update(float dt);
    void UpdatePerceptions(float dt);
    void Init();
    void Reset();
    void GetNetOutput();
    void TrainNetAndSave();
    void ReTrainNetAndSave();

    //perception data
    float       m_powerupScanDist;

    //network output variables
    bool        m_shouldThrust;
    bool        m_shouldTurnLeft;
    bool        m_shouldTurnRight;

private:
    int         m_numIterationsToTrain;
    int         m_numSavedTrainingSets;
    float       m_maximumAllowedError;

    //network input variables
    float       m_speedMovingTogether;
    Point3f     m_nearestAsteroidDelta;
    float       m_shipMovingDirection;
```

```
    //net, used for training and for actual usage in game
    NeuralNet*    m_net;
    vector<float> m_inputs;
    vector<float> m_outputs;
    int m_numInputs;
    int m_numOutputs;
    int m_numHiddenLayers;
    int m_numHiddenNodes;
    int m_netMode;
};
```

The constructor for this class sets itself up to do what needs to be done based on whether we're instantiating the controller in training mode, retraining mode, or the regular "use" mode. During the training modes, the network is instantiated by the training functions themselves and closed down after execution. The regular game use mode instantiates the network right away because the game will potentially be using it to avoid obstacles.

In regular training mode, there is no real AI running because the training uses real input from a human player. As you can see in the `Update()` function, the `NNAIControl` structure stores what will be the network input and output variables whenever the `m_willCollide` perception is true. When 1,000 sets of data are collected, the `Update()` method then instantiates and trains a network using the data, and finally saves off the network weights so they can be reused later.

Retrain mode works by loading the saved input and output training data from a file and training the network, than exiting from the game. Retraining is useful when you want to try different network designs (such as adjusting the number of hidden layers or nodes, changing to different activation functions, using more or less training iterations, etc.). Of course, if you decide to change the number of inputs or outputs, you'll need to recapture new training data using the regular `NM_TRAIN` mode.

Use mode actually operates a finite-state machine (FSM) to run the ship. A slightly adjusted evade state (the `Update()` function from the new class, `StateNNEvade`, is shown in Listing 21.7), then uses output from the controller's NN to determine what to do in the case of an imminent collision. The net's output is determined in the `NNAIController::Update()` function, which checks the collision perception and updates the net output if necessary. `GetNetOutput()`runs the values through the net to get the current outputs and converts those outputs back into Boolean values. You might be asking, why not just have the net output Booleans directly? Because using analog values makes it easier to determine error gradient information, which will help us train the network better and faster. Plus, we can then determine the amount of generalization we want from our net. If an output is 0.4f, you might have some systems where that would still be a positive output; the game could also set a

TeamLRN

secondary action to occur, or adjust the primary behavior to take into account the low level of net output given the current input data. The inverse would be that you want very high levels of output before you set off an action, but again, it is much easier to make these kinds of determinations if the values coming out of your network are analog instead of purely digital.

LISTING 21.6 `NNAIController` Function Implementations

```
//--------------------
NNAIControl::NNAIControl(Ship* ship):
FSMAIControl(ship)
{
    m_net      = NULL;

    Init();

    if(m_netMode == NM_USE)
    {
        m_net = new NeuralNet(m_numInputs,m_numOutputs,
                              m_numHiddenLayers,m_numHiddenNodes);
        m_net->ReadWeights();
    }
    else if (m_netMode == NM_RETRAIN)
    {
        m_numSavedTrainingSets = 1000;
        ReTrainNetAndSave();
    }

}

//--------------------
void NNAIControl::Update(float dt)
{
    Ship* ship = Game.m_mainShip;
    if(!ship)
    {
        m_machine->Reset();
        return;
    }

    switch(m_netMode)
    {
```

```
case NM_TRAIN:
    UpdatePerceptions(dt);
    if(m_willCollide)
    {
        //write test data to file
        FILE* pFile;
        if ((pFile =fopen("NNtrainingdata.txt","a"))== NULL)
            return;

        fprintf(pFile,"%f %f %f %f ",
                    m_nearestAsteroidDelta.x(),
                    m_nearestAsteroidDelta.y(),
                    m_speedMovingTogether,
                m_shipMovingDirection);
        fprintf(pFile,"%d %d %d ",ship->IsThrustOn(),
          ship->IsTurningRight(),ship->IsTurningLeft());

        m_numSavedTrainingSets++;
        m_inputs.push_back(m_nearestAsteroidDelta.x());
        m_inputs.push_back(m_nearestAsteroidDelta.y());
        m_inputs.push_back(m_speedMovingTogether);
        m_inputs.push_back(m_shipMovingDirection);
        m_outputs.push_back(ship->IsThrustOn());
        m_outputs.push_back(ship->IsTurningRight());
        m_outputs.push_back(ship->IsTurningLeft());

        fclose(pFile);
    }

    if(m_numSavedTrainingSets==NUM_TRAINING_SETS_TO_AQUIRE)
    {
        TrainNetAndSave();
        Game.GameOver();
    }
    break;

case NM_RETRAIN:
    Game.GameOver();
    break;

case NM_USE:
default:
    UpdatePerceptions(dt);
```

TeamLRN

```
                if(m_willCollide)
                    GetNetOutput();
                m_machine->UpdateMachine(dt);
                break;
        }
    }
    //--------------------
    void NNAIControl::TrainNetAndSave()
    {
        m_net = new NeuralNet(m_numInputs, m_numOutputs,
                              m_numHiddenLayers, m_numHiddenNodes);

        vector<float> tempIns;
        vector<float> tempOuts;
        for(int i =0;i< m_numIterationsToTrain;++i)
        {
            for(int j = 0;j< m_numSavedTrainingSets; ++j)
            {
                tempIns.clear();
                tempOuts.clear();
                //get training set inputs
                for(int k = 0;k<numInputs;++k)
                    tempIns.push_back(m_inputs[k+j*numInputs]);
                //get training set outputs
                for(k = 0;k<numOutputs;++k)
                    tempOuts.push_back(m_outputs[k+j*numOutputs]);

                m_net->Train(tempIns,tempOuts);
            }
            float totalError = m_net->GetError();
            if(totalError < m_maximumAllowedError)
            {
                //save out net and exit
                m_net->WriteWeights();
                return;
            }
        }
    }

    //--------------------
    void NNAIControl::ReTrainNetAndSave()
    {
        FILE* pFile;
        if ((pFile = fopen("NNtrainingdata.txt","r")) == NULL)
            return;
```

```
m_net = new NeuralNet(m_numInputs,m_numOutputs,
                      m_numHiddenLayers,m_numHiddenNodes);

vector<float> tempIns;
vector<float> tempOuts;
for(int i =0;i< m_numIterationsToTrain;++i)
{
    for(int j = 0;j< m_numSavedTrainingSets; ++j)
    {
        tempIns.clear();
        tempOuts.clear();
        //get training set inputs
        for(int k = 0;k<m_numInputs;++k)
        {
            float temp;
            fscanf(pFile,"%f ",&temp);
            tempIns.push_back(temp);
        }
        //get training set outputs
        for(k = 0;k<m_numOutputs;++k)
        {
            float temp;
            fscanf(pFile,"%f ",&temp);
            tempOuts.push_back(temp);
        }

        m_net->Train(tempIns,tempOuts);
    }
    float totalError = m_net->GetError();
    if(i> 100 && totalError < m_maximumAllowedError)
    {
        //save out net and exit
        m_net->WriteWeights();
        return;
    }
  }
}

//--------------------
void NNAIControl::GetNetOutput()
{
    //clear out temp storage
    m_inputs.clear();
    m_outputs.clear();
```

TeamLRN

```
        //set up inputs
        m_inputs.push_back(m_nearestAsteroidDelta.x());
        m_inputs.push_back(m_nearestAsteroidDelta.y());
        m_inputs.push_back(m_speedMovingTogether);
        m_inputs.push_back(m_shipMovingDirection);

        //get output values
        m_net->Use(m_inputs,m_outputs);
        m_shouldThrust    = m_outputs[0] > BOOL_THRESHOLD;
        m_shouldTurnRight = m_outputs[1] > BOOL_THRESHOLD;
        m_shouldTurnLeft  = m_outputs[2] > BOOL_THRESHOLD;
    }
```

LISTING 21.7 StateNNEvade::Update() Method

```
//--------------------
void StateNNEvade::Update(float dt)
{
    NNAIControl* parent = (NNAIControl*)m_parent;
    Ship* ship = parent->m_ship;

    if(parent->m_shouldThrust)//thrust
        ship->ThrustOn();
    else
        ship->ThrustOff();

    if(parent->m_shouldTurnRight)
        ship->TurnRight();
    else if(parent->m_shouldTurnLeft)
        ship->TurnLeft();
    else
        ship->StopTurn();

    parent->m_debugTxt = "Evade";
}
```

PERFORMANCE WITHIN THE TEST BED

Although slow, training the network with these parameters and setup is fairly successful. Most of this success is based on capturing good evasion data, which is the

issue with most NN systems. Given the right data, you can get the network to use many of the same techniques to evade collisions. In game, the CPU hit of using the trained network is negligible, which is always a good thing.

Training the NN with the largest possible training data set will allow the best results, especially in this situation, where the net needs to learn a fairly complex task. However, if your training set (for whatever reason) simply can't be very large and varied, you might need to watch out for overfitting of the data. An overfit NN is one that doesn't generalize well because it has matched the patterns of the input *too* closely and is no longer flexible enough to accurately include errant data points. One way to counter this involves what is called *early stopping* to keep the network from overfitting the data. The technique is simply to stop training at the point at which you have balanced the line between generalization and error. You want an accurate NN, which makes the right decisions most of the time, but you still want it to intelligently "guess" if the input variables are a bit off kilter. Finding the best point at which to stop training is another tricky problem, one that requires experience and experimentation to solve. Most systems monitor the error coming out of the network, and stop training when error starts to increase after a long period of decreasing. However, this is not a hard and fast rule: the net may be finding its way out of a local maxima, rather than degrading performance caused by overfitting.

OPTIMIZATION

Optimizing NNs generally involves optimizing the training phase to get better because most NNs are used offline, and using an already trained network to make game decisions is very fast. To speed up training of the algorithm, try to remove any unnecessary inputs (or consolidate inputs into more complex calculations) or hidden nodes. Also, experiment with the amount of error you are willing to live with because even a small decrease in maximum error allowance can allow a savings of many thousands of training iterations.

The other level of optimization of using NN systems lies in lessening the time it takes to construct a viable network design and creating highly effective, relevant training data. For any NN task that is nontrivial, both of these tasks are difficult and can take up a lot of programmer time. Optimizing this step of the process, however, involves having an understanding of how NNs work and knowledge about the specific task at hand. In short, the more knowledge you have up-front about the relationships you're trying to model with the net, the better you will be at picking the right net inputs and choosing the minimum needed outputs, and the better your training data will be. Some general things to think about if you find this process taxing you:

TeamLRN

- If your network seems to be getting stuck too easily in local maxima, where the error becomes stable, but is still higher than you'd like it to be, then you might be using too few training sets, or your hidden layer might be too small (by not having enough neurons at the hidden level, you haven't given your network enough degrees of freedom in which to search for the best solution).
- If your training seems to be unstable (meaning that the error seems to jump all over the place, never seeming to settle down or lessen consistently), you might have too many hidden layer neurons, and the network has essentially been given too much room to experiment within.
- Overfitting, as we have mentioned before, can happen when you have too few training set because even a very simple net can store a lot of information about a limited amount of data. Another point is that overfitting might happen when you have too many training iterations with the data, you have trained *too long*. Try reducing the amount of iterations for each training set.
- Underfitting could occur if you have a large amount of very noisy training sets, or you don't train for enough iterations. An underfit NN is all generalization, with almost no accuracy. If your data is very noisy, it can exacerbate the problem by making it hard for the net to filter the noise from the real data. Finding a way to scale these training sets, to bring out the differences between real data points and noise, can help this process.
- If error seems to be oscillating between values, you may be using too large a learning rate, or your momentum might be too high. Gradient descent is a greedy algorithm and will perform poorly if the step size (in this case, the learning rate) is too high. Possible solutions might be to simply lessen the learning rate (which would help, but might also dramatically lengthen training times), to dynamically change the learning rate (if the network's error is going down, slowly increase the learning rate; decrease the rate if the error is going up), or even to use a more costly method like Newton's (which involves finding the second derivative of error for pinpointing the nearest minimum).

PROS OF NEURAL NET-BASED SYSTEMS

NNs are a great way to find abstract relationships between input conditions. They are great at storing esoteric knowledge in a very usable and optimized way. Some of the other benefits of the method include the following:

- NNs can extract very complex mathematical function solutions. These mathematic functions are essentially approximated into the weights of an NN, so that when you use the net in game, you essentially save yourself the CPU cost of having to perform the actual math. It has been mathematically proven that

an NN with at least one hidden layer and nonlinear activation functions can accurately depict almost *any* finite dimensional vector function on a compact state set.

- Nets have an excellent ability to derive meaning from nonlinear or imprecise data. They can generalize connections and relationships between data in ways that are unintuitive or even impossible to see for a human. A well-trained, well-designed NN can generalize better than a human expert.
- Training takes a fraction of the CPU time that trial and error methods take, once a suitable network design has been determined.
- Humans "make sense" of them. The way that NNs organize data and knowledge seems to appeal to people in a way that we can get a grip on, and so they are easier to debug or experiment with then something more esoteric, like fuzzy logic systems.

CONS OF NEURAL NET-BASED SYSTEMS

NNs are a great way of solving some problems. But they are not a free lunch. Determining how to *train* an NN is usually the cost. The problem has thus been shifted. Instead of figuring out how to solve the problem (which *may* be an exponentially hard problem) we have replaced our work with now figuring out how to teach the NN to solve the problem (which has definitely been shown to be an exponentially hard problem). Other points of contention are the following:

- NNs are not magical, garbage in = garbage out. If you use arbitrary, numerous, or even bogus inputs to the network, there's a good chance that the net will find *some* correlating factors between them. This does not mean that you're going to want the output of those correlations. In fact, NNs are *famous* for learning the wrong thing. Most of the difficulty in finding the correct inputs and training data is to weeding out input and training set relationships that you *don't* want the network to learn. Usually these bad relationships are only found after a network has been trained, and has learned the unwanted abstraction. Only then might you stumble on the realization why your network is behaving errantly. But the correlation might be so obtuse that it might never occur to you, and hence, you would be stuck with pure trial and error in getting around the problem.
- An NN is a mathematical black box, and thus, hard or even impossible to debug. Once trained, the weight data within an NN is incomprehensible. You can't look at them like you can the nodes of a decision tree structure and determine what is going on within the net. The information in the network is distributed throughout the connections in highly parallel or multiply efficient means; not alphabetically, in some kind of hierarchical fashion, or even based

TeamLRN

on the order of training. Debugging an NN solution usually involves going back to the starting board to adjust the pretraining parameters or data, and then retraining.

- All input fields must be numeric. Fuzzy values, or inputs that might be represented better by an expression, cannot be modeled within an NN. It would be better to use a hierarchical system in this case, with the NN being used for the more straightforward elements, and a different overhead structure (like a decision tree, or simple FSM) handling the strange or lesser defined cases.
- NNs are difficult to implement, due to the high number of factors that must be determined without guidelines or rules for the best way to go about it. These factors involve network structure, input and output choices, activation function, learning rate, training data issues, and so forth. Nets are also very sensitive to somewhat random factors like weight initialization or redundant inputs.
- Overfitting, or noise learning ruins generalization power and must be countered with the techniques described earlier.
- NNs can sometimes suffer from a phenomenon known as *catastrophic unlearning*. This occurs when an almost fully trained network is given additional training data that completely undoes all previous learning. Late addition of NN functionality, possibly suggested because of feedback from testing or focus groups, should be handled with care, unless you have given yourself ample time to deal with problems from mucking with the network.
- Lots of training data and CPU time may be required for training, especially for complex learning scenarios within large search spaces. If bugs come up in quality testing because of the NN portion of your game AI, retraining the network might become prohibitive, so be sure to consider this when deciding to go with an NN system.
- NNs don't scale well. NNs larger than a thousand nodes are rare and not very stable. Although the reasons behind this aren't completely understood, it appears that the *curse of dimensionality* (as it is sometimes called) seems to cause the learning ability of these large nets to implode somewhat, where there is so much freedom of movement within the search space that the network can essentially cyclically vary its weightings forever, never getting closer to a solution. Luckily, the types of NNs you might use within games have almost no reason to get this large.

EXTENSIONS TO THE PARADIGM

The FF, backpropagated-trained NN used in this chapter is *far* from the only type of NN in the world today. The number of network types is large and each is specifically designed for unique performance within a particular area. Some of these

models do not really apply to gaming use, but it is still important to know of their existence, so that future exploitation can occur. Most of these come from the academic or business world, where NNs have evolved during the almost 40 years they have been around. Some of these other types or extensions to the method are listed here. Note that this is not an exhaustive list by any means.

Other Types of NNs

Simple recurrent networks. These are basically a variation on regular multilayer NNs. In this scheme, the hidden layer of the network is also connected back to the input layer, with each connection having a weight of 1. The fixed back connections result in basically maintaining a copy of the previous values of the hidden units (because the net propagates over the connections before the learning rule is applied). Thus, the network can maintain a sort of state, allowing it to perform such tasks as sequence-prediction, which are beyond the power of a standard multilayer networks.

Hopfield nets. Designed to mimic *associative memory* within the brain, these networks allow entire "patterns" to be stored, and then recalled, by the system. Also like the brain, if some of the connections between various parts of the system fail or are severed, the recall still has a good chance of succeeding. These structures use a set of completely connected neurons, each of which can only store a Boolean value. There are no dedicated input and output neurons in this system. Rather, input is applied to *all* the neurons, and then allowed to propagate until a steady state is achieved, at which point the state of all the neurons is considered the output of the system. These are useful for pattern recognition, especially with recall of these patterns. If you store a number of images within a Hopfield network, and then input one of those images, but destroy or corrupt parts of it, the network will usually be able to determine which stored image you started with. One thing that Hopfield nets provide over normal NNs is that the number of nodes necessary to store information, as well as additional information, can be calculated directly. Unlike normal NNs, where the number of nodes is somewhat mystical, the nodes within a Hopfield net are simply distributed storage for the array of patterns, and so their number is simply a function of how much information needs to be absorbed.

Committee of machines. This is a technique where multiple nets are trained on the same data, but each is initialized differently. Then, during usage, all the nets are run on the input data, and the collection of networks "votes" on the final output, by taking statistical notion of the output of all the separate networks. This has statistically been shown to smooth out the problems dealing with neural nets. It is, however, even more costly in initial time investment than normal nets.

TeamLRN

Self-organizing maps (SOM). Useful for classification tasks (or clustering, as SOM users call it), SOMs have two network layers: an input layer, and a *competition* layer. The sum of all the input connections to any one neuron in the competition layer is called a *reference vector* in the input space. In essence, the SOM consists of a number of input *vectors* that are represented by a set of neurons in the competition layer, which are usually laid out as a two-dimensional grid of neurons. SOMs use a form of unsupervised learning called *competition learning*, hence the layer's name. When a new input pattern is introduced to the net, the first step is to find the competition neuron whose reference vector is closest to the new pattern. The "winning" neuron is then singled out and becomes the focus of weight changes within the network. Not only does this neuron's weights change, however, but also those neurons within its *neighborhood* (defined as all the neurons within some grid distance from the winner), will change in proportion to their distance from the focus neuron. The size of this neighborhood shrinks over time, so that when fully trained, the neighborhood size is zero. The effect of this method is that the organization of the input data becomes grouped within the net so that inputs that are the most similar will be located closer together. The chief usage of these kinds of maps is to visualize relationships and classes within large, highly dimensioned inputs. In games, it might be useful in player modeling, taking a number of dimensions of behavior into account and giving the AI system a better picture of the kind of player the human tends toward.

Other Types of NN Learning

Reinforcement. The backpropagation technique used in this chapter is a form of supervised learning which some call *learning with a teacher*, because you are providing the network with output target values. Reinforcement learning also involves supervision, but only evaluative help. It has thus been called *learning with a critic*. When the network outputs some value because of the input information, the supervisor simply declares if the result was a good one. This can be done by a human expert or can be delivered to the network as an additional input signal broadcast from outside the network (the environment, or something else that the net has to interact with), which would allow the net to perform mostly unsupervised.

Unsupervised learning. These techniques allow the network to train itself completely on its own. It does not require hand-fed training input data, or target outputs. Examples of unsupervised techniques include using a genetic algorithm to find the best set of NN connection weights (this technique becomes much like reinforcement learning, where the GA fitness function becomes the

critic), perturbation learning (where the weights are iteratively and randomly adjusted and tested for improvement), or competitive techniques (which involve a number of neurons "competing" for the right to learn by having their weights adjusted within the net; the winner is usually based on the inputs being given to the net rather than a stated output). Another branch of unsupervised learning involves problems for which the output cannot be known ahead of time. In this case, the main job of the network is to classify, cluster, find relationships, and otherwise compress the input data in specific areas. SOMs are an example of this.

DESIGN CONSIDERATIONS

Neural networks are definitely not a "one size fits all" AI technique, especially in gaming, where their inflexibility toward debugging and extension make them hard to tune for gameplay concerns. Take careful note of the engine design considerations from Chapter 2, "An AI Engine: The Basic Components and Design": types of solutions, agent reactivity, system realism, genre, platform, development limitations, and entertainment limitations.

Types of Solutions

NNs are great when you have simple, modular systems that map inputs to outputs (in surprising, nonlinear, black box ways). Because of this, they tend to be much more *tactical* in nature, rather than high-level strategic solutions. You wouldn't want to train an NN to run the diplomacy campaign in your real-time strategy (RTS) game. That task is simply too large, too complex, and needs far too much tuning and tweaking. But you might use it to decide which animation you want to play to catch a baseball. Here we have a very atomic task, with specific inputs (the incoming ball, the player's position, and his skills) as well as specific outputs (each of the possible catch animations). The logic for mapping these together with more common techniques might be fairly CPU intensive or complicated to construct. A small NN will be far less CPU intensive, and the finished network wouldn't need changing unless additional catch animations are added to the game (in which case, traditional game logic might have to be added anyway).

Agent Reactivity

NNs can actually optimize CPU-intensive calculations, so their use might actually contribute to faster reactivity by a game agent.

TeamLRN

System Realism

In many ways, NNs help to make systems seem much more realistic, mostly because they are *general* pattern-matching systems, meaning that they are not special case systems like FSMs or scripted entities that can react specifically to particular scenarios. Because of this, they might react wrongly to something, but still in a way that *seems right,* because the pattern still holds for whatever reason. People routinely run into the same problems, mostly because we're using the same sorts of general case pattern matching in our own minds. So you see people running into glass doors from time to time, or reaching out to where they *thought* the wall was to stop their fall. Whether or not this behavior translates into the video game world is up to the context of the action, the type of game you're creating, and the intended audience. A comedy adventure game primarily being played by more mature people might catch small details like this and perceive it as much more realistic and humorous. But a young child playing a serious, heavily action-based game might see such "mistakes" as "stupid AI."

Genre and Platform

Not really a limiting factor for NNs. They are truly a modular technique, useful when you have a specific need for their categorization or predictive powers.

Development Limitations

Here is the primary concern for NN usage. As stated in other parts of the chapter, NNs require both an upfront investment in time and energy to design and collect training data. They also require a significant period to actually train, as well as deal with tuning issues later in development. Online learning (during live gameplay) in NNs requires an even larger commitment in time and design. If it is hard (or even impossible) for testers to restage crash events in your game because the adaptive NN in your AI system keeps dynamically changing behavior or game events, you are going to be hard pressed to completely debug the game. Plus, the sheer effort of trying to test a gameplay system that has large areas of adaptive or evolving elements is obviously much greater than static content that can be tested from A to Z.

Entertainment Limitations

The difficulty in tuning, tweaking, and adding to an NN-driven system has been discussed throughout the chapter. For this reason, NNs should really only be used on areas within the game that require the kind of "do it once and leave it," black box sort of solution that NNs provide.

SUMMARY

Neural nets are another AI technique that can help you solve difficult problems, especially if they are nonlinear or unintuitive in nature, and you can either come up with test data for training, or determine some way of using unsupervised methods to teach the system.

- Natural brains work by clusters of brain cells called neurons transmitting electrical impulses to each other over synapses crossing from axon to dendrite.
- Artificial neurons have a number of inputs (each with an assigned weight), an internal bias, and an output value with an optional activation function.
- Neural Nets are connected systems of neurons in particular formations. The usual structure is a number of input nodes, followed by a number of hidden nodes, followed by a number of output nodes. Feed forward networks are only connected in one direction. Recurrent networks are completely connected, in both directions. Other systems also exist.
- When using an NN, you must determine the NN structure, choose a learning type, and create training data. Then, you can implement your NN, and being tuning the implementation to optimize your results.
- Pros of the method include the ability to extract and compress complex mathematic functions, their powers of generalization, fast usage in game, and the fact that their operation "makes sense" to most people.
- Cons of NNs involve their difficulty to implement and debug, their sensitivity to training, the time investment requirement to train and test them, and that they usually don't scale well to larger problems.
- Some extensions to NNs include recurrent networks, Hopfield nets, self-organizing maps, and the committee of machines usage. Other learning techniques besides supervised learning involve reinforcement and unsupervised learning.

TeamLRN

22 Other Techniques of Note

In this chapter, we will discuss a few of the remaining AI techniques that show promise in the game AI programming field. Each technique will be dissected with an overview of the method, some general usage notes, pros and cons, and design considerations.

ARTIFICIAL LIFE

Science in general has always been a search to find what some call "governing principles." That is, rules or truths so universal, so fundamental, that they cannot be broken down any further and that can be used without fail to understand aspects of nature and predict outcomes based on hard equations. Newtown gave every scientist a collective warm fuzzy feeling when he "proved" Descartes' *clockwork universe* view of things (in which God set up everything like a clock; including the planets, plants, animals, and everything nonhuman), by providing the world with the equations necessary to boil down the movements of everything into neat, mathematical bundles. We have since found entire oceans of situations where Newton's theories break down, and scientists are once again on the prowl for governing principles, and this is in the inanimate world. Now that we are able to look at the behaviors of living organisms at such close distances (or even at the behaviors of subatomic particles), we are increasingly astounded by the diversity and complexity that almost every system shows. But with inanimate systems, we can begin to break things down, to extract deeper knowledge by deconstructing physical systems into components that can be isolated and studied. Life, however, does not yield to these methods. Its very nature disallows this kind of disassembly, and so we are at a loss. Life scientists are also at a loss for examples of early work because the basic living system doesn't exist anymore; it was replaced by more complex evolutions billions of years ago. An equivalent task would be for an alien race to try to determine how writing started by studying the *New York Times,* and nothing before. The chances

TeamLRN

of finding any governing principles is very limited. So, maybe we should try to construct our own, ultrasimplistic "life simulations" and, by doing so, find out more about how life in general operates.

Artificial life (or *alife*) is the field of studies that hopes to understand natural life better by attempting to recreate biological phenomena from within virtual computer environments, or other artificial means. It is actually the name for an entire *collection* of computer science and engineering disciplines, although they do share some ideas. One main tenet of the alife camp is that life is simply an emergent property of nature following some very simple rules over and over again. An emergent property refers to *a trait or behavior exhibited by a creature that reaches beyond the capabilities of its constituent parts.*

Within games, we search for emergent behaviors and gameplay situations as well. This has led many to investigate or try to employ alife principles in the search for new ways to have fun within the confines of a game environment.

Artificial Life Usage in Games

Some popular games (and by popular, I mean critically popular; neither game was wildly *financially* popular) that are considered alife include *Black & White* and the *Creatures* games, which were discussed in Chapter 14, "Miscellaneous Genres of Note." Both of these titles had beings that used very simple rules (your totem animal in *Black & White* was supposed to watch you and try to make you happy while catering to its own needs; *Creatures* actually modeled whole systems of chemistry and genetics to try to simulate "building" an entire being, its Norns) that combined in interesting ways to dynamically foster behaviors that hadn't been specifically programmed in by the game authors.

Artificial Life Disciplines

Some of the disciplines that are considered alife include Cellular Automata, self-organizing behavior and flocking, genetic algorithms, and robotics.

Cellular Automata (CA)

CAs are a group of algorithms that show a stunning amount of complex behavior with the very simplest of rules. One of the most famous CAs, *Conway's Game of Life,* is played using a two-dimensional (2D) collection of cells, each of which can be populated or empty, and each of which has eight neighbor cells. By then applying four simple rules to the cells of the playfield, a vast amount of complex behavior can be seen.

"The Rules" for a populated cell:

- Each cell with one or no neighbors dies, by loneliness.
- Each cell with four or more neighbors dies, because of overpopulation.
- Each cell with two or three neighbors survives.

"The Rule" for an empty cell:

- Each cell with three neighbors becomes populated, because of a birth.

Certain constructs in *Conway*'s CAs have even been shown to be able to perform mathematics. Others can reproduce various structures from nature with incredible detail, including plant structures, seashells, and coral reefs. The freeware program *MCell*, written by Mirek Wojtowicz, can display a vast amount of CA behavior. It has a general graphical interface for building and watching CAs in action. Some of its output can be seen in Figure 22.1.

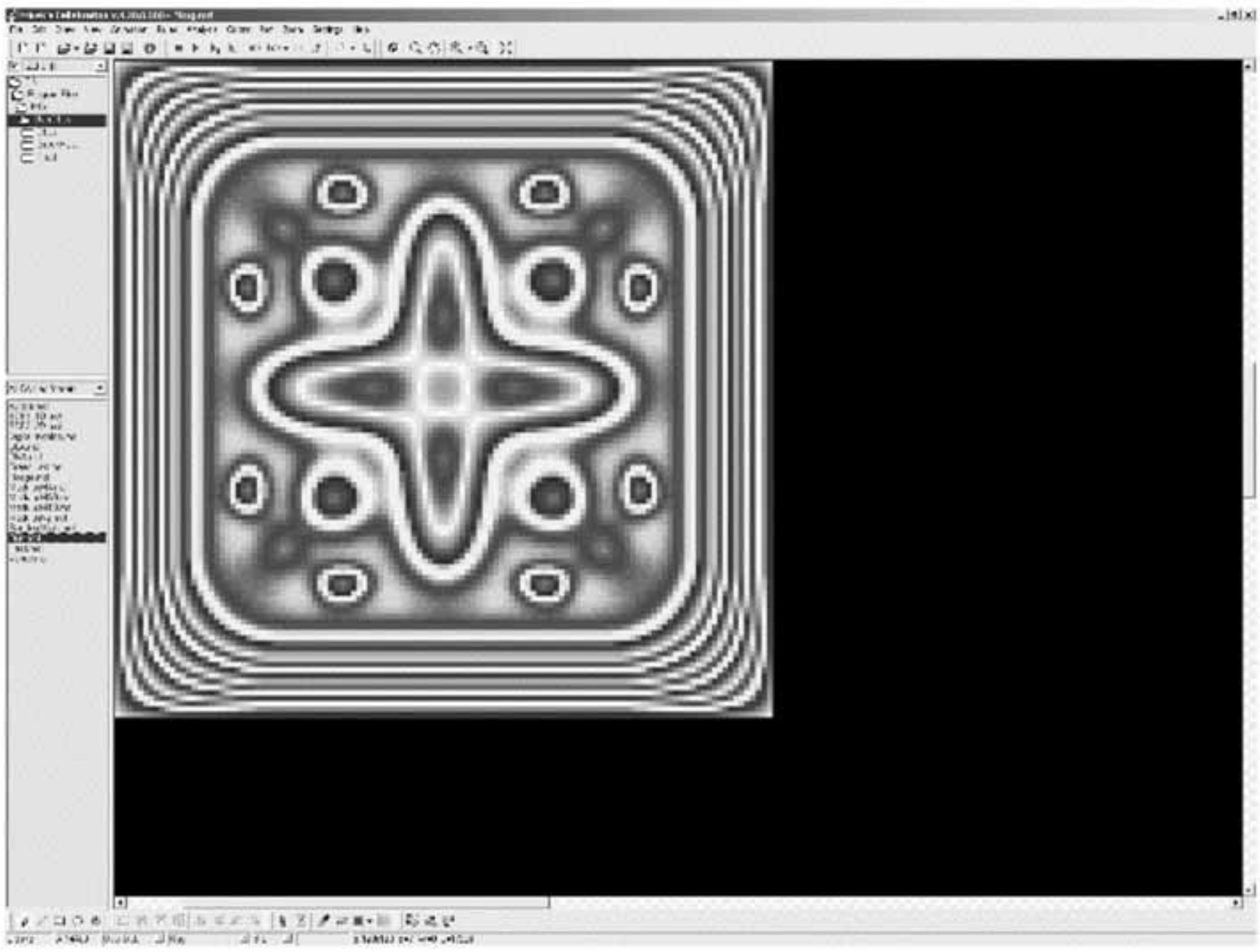


FIGURE 22.1 An example of output from the program *MCell*.

TeamLRN

CA behavior patterns can be found at the microscopic *and* macroscopic levels. At the small side, these patterns can be used to simulate the growth of mold, or the spread of amoebae, and at the other end of the spectrum, can be used to discover trends in traffic jams, or city building.

Self-Organizing Behavior and Flocking

Large groups of creatures (such as fish, ants, and birds) can organize their movements within groups quickly, easily, and with what appears to be a unified mind. Investigations into how they do these things have given us algorithms that allow us to replicate these kinds of behavior using simple rules. The most famous of these are Craig Reynolds's *Boid* research [Reynolds87], where he described how by using a few key concepts (namely *separation* to avoid crowding, *alignment* to the average group heading, and positional *cohesion* of the group), you can achieve a remarkable simulation of many types of flocking movement. Games employing flocks of animals or crowds of people almost universally use Reynolds's algorithms for simulating their movement. Other useful algorithms from this area that have found their way into games that involve ant and bee colony studies (both animals construct very complex structures, and work together, with a kind of organization that defies their simplistic level of intelligence), economics (current theory shows that economies appear to construct themselves and will either collapse or not based on somewhat-hidden rules, such as Massively Multiplayer Online RPGs [MMORPGs], game developers are beginning to find out).

Genetic Algorithms

Genetic algorithms are sometimes lumped into alife, although many alife researchers would say that is a wrong classification. When you consider that GAs are a pretty abstract system, and that the one thing GAs try to model (evolution through genetic manipulation) is almost nothing like the simplified version used by the algorithm as we know it, you should rather think of GAs as an interesting tangent that computer science has devised by drawing on the *idea* of evolution.

Robotics

Although most of robotics deals with creating systems that can perform in places or ways that we cannot, it can be said that some roboticists are trying to create artificial beings, hence *physical* alife. They aren't trying to understand nature but, rather, are trying to create their own. In trying to emulate life, they end up with an understanding of how things are done in nature and the problems that nature is solving with its solutions, an example of very cyclical scientific thinking. Different researchers are going in opposite directions toward this end. Some are creating robots that are very simplistic, but can communicate with other simple robots to create

hive-mind communities. Others develop robots that are being trained by humans to act like humans, including emotional response, a sense of personal space, and personality development.

Pros

- *Emergent behavior.* Alife is one of the best ways we currently have of creating emergent situations semiconsistently. The more scripted a behavior, or sequences of behaviors are, the less emergence you are going to see, by definition. Conversely, emergent behavior will be most likely found in open games (meaning they allow the players and AI characters to perform many types of activities) with simple actions that can be combined in many different ways, leading to a wealth of different final behaviors.
- *Behavior reuse.* Alife techniques force developers to build games out of building blocks, distilling down the gameplay until it can be expressed as simple rules that only have meaning over many iterations. In fact, most alife game creations are simple to code for, but take lengthy amounts of time to tune.

Cons

- *Emergent behavior.* Emergence in the game industry is a huge double-edged sword. Alife *can* create solid, compelling gameplay situations out of thin air. But, that creation might never come, leaving you high and dry, with no real game to speak of. The emergent behavior might not be that entertaining, or too difficult for most audiences. Anytime you have free-form behavior, with no set outcome in mind, you open the door for both the magic *and* the mundane.
- *Tuning issues.* What if tuning the game destroys the emergent behavior? Small changes to game parameters, or gameplay systems, could easily unravel the very thing that is the most compelling part of your game. In fact, fixing a bug sometimes sets in motion a chain of events that might subtly change gameplay for the worse. This situation can happen to any game, however, and sometimes (with some work) you can find out the reason behind the advantageous configuration of the buggy code and incorporate it into a bug-free version, minimizing this problem. However, there are no guarantees because what you are dealing with is several factors working in conjunction in obviously nonintuitive ways.

Areas for Exploitation within Games

- Further use of more sophisticated flocking techniques, for city crowds, and so forth.
- Other types of movement can also be simulated using *Conway's Game of Life* rules, like the exploratory creeping of single-cell organisms, or the spread of plant vines.

TeamLRN

- MMORPGs using alife techniques with creatures to create actual ecologies within the world, instead of random spawn points with scripted monsters.

PLANNING ALGORITHMS

Planning is defined as *deciding upon a course of action before acting,* specifically by using knowledge of a larger scope about the problem to chain together actions that will lead you toward a more long-term solution. A clear-cut example of this in real life is a creature that, although wildly successful, does no planning at all: the common housefly. Given its tiny brain, it can still fly rings around the typical human wannabe flykiller with almost comical efficiency. But a fly cannot, and will never, see a closed window and tell itself "Hmm. Better go around to the open door." This simple fact is the reason that more dead flies can usually be found in windowsills than anywhere else in the house. For flies, windows are the game AI equivalent of badly connected pathfinding nodes, "If I just keep trying, I should be able to get out . . ."

Conceptually, most planning algorithms follow a somewhat simple formula:

- Break the abilities of the AI into distinct *operators.*
- Design your AI character, as well as your game environment, so that it can be represented as being a member of a set of *states.*
- Either *construct a tree* that shows the transition connections between states (listing the operators that will cause these transitions), or have rules embedded in each state that details which operators are available. The AI then forms plans within a local working memory by applying these transition operators on a copy of his current state, testing for the best action to get him to the behavior he wants.

Then, planning involves knowing what state you're in, and what state you want to be in, and then finding the string of operators that will get you from your current state to the final state. In pathfinding, the operators are all movement types (physically running or walking, as well as including things like taking the train, or "use teleporter." The states in this case would be the pathnodes within the map that define the pathfinding network, which is the tree that you would then use with your trusty A* algorithm to plan a path.

Taken a certain way, our standard decision making paradigms, such as finite-state machines (FSMs) and all the rest, can all be considered a form of *preprocessed* planning, a sort of optimization on the planning process. Given a robust representation of the game, and a wide range of low-level operators, a planning algorithm should be able to find the best behaviors necessary to affect the game state in any

legal way. But because planning algorithms can be costly, we have historically used "hardcoded" planning (in our case, a state machine, a script, etc.) that allows us to *usually* do the right behaviors. With more complex game environments, invariably our set patterns of behavior have areas in which they fail, and these are where exploits are born.

Current Usage in Games

Most games use some form of planning algorithm already, in the form of the A* search they're using for pathfinding. The pathfinding system stores a wide scope of information about the game world and allows the AI-controlled creatures to make plans about how to travel from A to B.

Some games, especially real-time strategy (RTS) games, use the exact same system to also perform other planning tasks. Say an AI civilization in an RTS game sees a certain kind of enemy unit cruise by: the Laser Boat. It now knows that the enemy can build those units, and to defend its shoreline structures against this boat, this civilization will require its own Laser Boats, or a defensive structure called a Tower of Reflection. By having a technology tree, which describes the prerequisites necessary for researching any given skill or structure, the AI can effectively generate a "path" from where it is in the technology tree to where it needs to be to build one of the two units it requires. It can even determine which defense is "closer" (or cheaper, or whatever metric it might currently favor), and go there. Also, by noting that the enemy has a particular technology, the AI can update its tech tree model for the enemy by checking off all the units that are prerequisites along the path to that technology.

Planning algorithms have just begun to be seriously used within games, mostly because of the advanced strategic thinking of RTS games. Some earlier genres, such as war games, had large quantities of advanced strategies, but most war games are historically based, and follow a semiscripted pattern that mimics the real historic battle; this usually works better than trying to model Napoleon, hoping that the game will fight the same way he did.

Planning is finally being seen as a primary human quality, however, so advanced AI systems are increasingly turning to planning to seem more intelligent and humanlike. In an FTPS, for instance, endowing a bot with anticipation can make it seem much more lifelike. Anticipation is another form of planning. An AI bot sees a human player enter a room. He could run a planning algorithm that would try and conclude what the human is going to do in that room. If it's a dead end room, with a nice powerup, then a planning run might come back that the plan is to get that powerup, and then come back out the same door. Not only does the AI have a good idea that the human will come back out the door, but because he has the action plan, he can even estimate about how long it will take for the human to

TeamLRN

appear in the door. The AI can set a very effective ambush. Another planning scenario might involve seeing the human with an inferior weapon, and chasing him down. But, the AI is also checking the human's potential "plan," and notices that in the direction the human is headed, there is a *much* better weapon around the corner. If the AI is not very healthy, and was only pursuing because of a firepower advantage, he might be smarter to break off and head for a health powerup, knowing that he has some free time because the human is going to be busy getting the weapon.

This is serious AI behavior and needs to be used in gameplay situations where the human is expecting a serious opponent. You wouldn't want just any first-person shooters/third-person shooters (FTPS) enemy in a long, story-filled game firing a rocket into a door just as you got there; that would seem pretty cheap, and not fun. But in a deathmatch setting, on a high level of difficulty, the human player almost expects this kind of behavior (because he himself uses it), and will make a mockery of bots that don't use it. The next level of this is to have the AI bot anticipate the human anticipating the bot. So if the bot runs into a room with no other exits, and had some notion that there was a human following it, it might fire a rocket toward the best ambush angle through the door, or simply wait in the room until the human gives up camping outside. This can get pretty expensive, but you can see the concept, and the benefit is that you have AI bots that cannot only exploit your moves, but can step out of their own routines if they sense you exploiting them, leading to advanced, humanlike performance.

Typically, even for planning that wasn't pathfinding, games have done this searching using A* (because most games have already implemented efficient, typically load-balanced versions of A* already). This is usually fine, especially because most computer games don't have large numbers of operators or agent states. However, if you find your planning algorithm slowing down your game, you should look into some of the more optimized planning search techniques, such means-end analysis (MEA). MEA combines forward and backward searching of your tree, and tries to minimize unnecessary search specifically for planning algorithms.

Another common planning optimization is called patch recalculation, where a "broken" plan (where a step has been invalidated because of some game event) doesn't invalidate the entire plan but, rather, is sent to a function that will come up with planning steps that will work around the broken link, thus *patching* the hole in your plan. This method is only useful if the length of your entire plan is long enough to justify not just tossing the entire plan and starting over. But for long or computationally expensive plans, this method provides a way for keeping plans up to date without having to start from square one all the time.

Minimax is another planning algorithm, which considers that your opponent is going to be working against you every chance he can get. Although minimax has been mostly used in board games like *Chess* and the like, certain turn-based RPGs

or civ games could (some already do) benefit from its use, by replacing the more scripted, repetitious combat sequences that are the norm of these games, and using a basic minimax to perform simple planning based on the abilities of the enemies and the humans. Specific battles could still be scripted, but most battles would not feel quite so monotonous and unchanging. The planner could also take into account some reinforcement learning, if you wanted to give the player a challenge by disallowing him (through effective defensive blocks) to get away with repeating similar, very effective combat maneuvers.

Pros

- Planning algorithms provide *much* more intelligent-looking behavior. Very few decisions in life require no forethought whatsoever. In fact, it could be said that anything larger than basic reflexive actions require at least some plan. Even scratching your nose requires a plan if you're wearing a motorcycle helmet and mittens.
- Planning is a generic algorithm and can provide data-independent solutions. So, the same pathfinding search algorithm in your RTS game can also help your AI research technology in the right order, set up the necessary orders to sequence a large scale attack on an enemy, and set up its bases in such a way that it doesn't run out of room later in the game.
- Like most generic algorithms, planning can be implemented hierarchically. You can layer your planning system, so that each layer has a much easier time creating its plan, thus optimizing the overall planning costs. An example of this would be a high-level planner finding the plan "Build Large Army, then Attack Next Town." The next layer down would then make a lower-level plan for "Build Large Army" and another for "Attack Next Town." The process repeats until you've developed plans at a low enough level that the resultant plan involves giving behavioral orders to the individual characters involved. Each layer in the system can use just enough detail as is required to simplify a given *particular* layer of the planner but still give meaningful plans.

Cons

- Planning can be computationally expensive, if unnecessarily long plans are attempted. Most games (even strategic games) rarely require their AI to plan too far in advance. Lengthy plans are costly to create because the human players are so unpredictable that a long plan rarely ever pans out. Plan depth is a careful balance between speed and flexibility of your plan versus having your plans be too short range to avoid gaffs. For long or expensive plans, some time can be regained by using patch recalculation.

TeamLRN

- Planning can make the AI seem sluggish or unreactive if plans are too monolithic, or take too long to adapt to new situations. Of course, this is within the confines of the game you are working on: large scale civ games can require more planning then most, but they are also usually turn based and, thus, will not be considered "sluggish."

Areas for Exploitation within Games

Games could make use of planning algorithms when creating strategic AI systems that require many steps to achieve goals. The previously mentioned RTS tasks are prime candidates. But action games could use some simple planning as well.

- FTPS opponents can use anticipation to set ambushes and traps.
- AI drivers can plan more complex racecar movements to pass an opponent in a more realistic way. Instead of strategic "speed-ups" that might actually cheat, AI drivers could feign on key corners, and then try to pass at critical times by planning maneuvers based on the other cars' positions, time left, and so forth.
- Fighting games could plan combos like human boxers have to: a boxer knows that if he strategically drops his guard and openly allows himself to take a specific punch that his opponent will then be open for a much more damaging combination.
- A football game could plan the order in which to perform plays to confuse the human, or to best take advantage of the time it has left on the clock.

PRODUCTION SYSTEMS

Production systems are sometimes referred to as *expert systems*—you might be using a primitive version right now. This is because production systems are essentially *rule-based systems that strive to encompass expert knowledge within a specific area*, the simplest example of which is that of using hardcoded conditional if-then branches within your AI engine to make decisions. Back in the old days of AI, researchers tried very hard to create *general* computer intelligence; they believed that they could solve every problem with the suitable brute force application of logical rules.

The trend continued until 1969 when Alan Newell and Herbert Simon released their theory of the General Problem Solver (GPS) [Newell61], which gave a basic set of rules for supposedly solving any problem, somewhat based on how they believed the human mind to operate, a process called means-end analysis. All that the algorithm needed was a statement of the goal to be achieved and a set of the problem's "rules." Although GPS was found to be very versatile with the simple puzzles and

chess problems that were defined well enough for its limitations, it didn't take long to discover that it was definitely not a solver of general problems. What it did do, however, was to introduce the concept of using actions as operators to transform your current world state. Production systems are the field that grew out of their theory.

Ironically, production systems are used to perform the exact opposite of what GPS was intended for, generality. Instead, these systems are now used to store expert knowledge about a highly *specific* problem. The first expert system was used to interpret mass spectra, and these systems have since been diagnosing specific diseases and giving mortgage tax advice. Games do this every day, with reams of code dedicated to the storage of expert rules necessary to play hockey, gobble power pellets, and rocket-jump. However, a full production algorithm is much more organized and separated into four parts: a global database, production rules, a rule/situation matcher, and a conflict resolution function (for use with rule collisions). The global database represents all the current facts the system knows about its environment. The production rules are the actual if-then clauses that serve as operators to transform our environment. The matcher is the function for deciding which operator to use next upon the database to get closer to your goal. The simplest matcher can be as simple as a function that searches the database, and compares rule "if" clauses to the current world state. However, specialized algorithms are significantly faster than brute force. Conflict resolution happens when multiple rules are matched to the database simultaneously. Most resolution schemes are very simple, even random.

One thing to note is that traditionally, production systems have only used what is known as *forward chaining inference* (meaning that they can only perform logical inference in the forward direction: if I AM ON FIRE, then I SHOULD JUMP IN THE LAKE), but modern extensions have allowed for *backward chaining inference* to be used as well (I JUST JUMPED IN THE LAKE, therefore I MIGHT HAVE BEEN ON FIRE).

In practice, production systems can be used to code regular game logic, serve as a planning system (because they can solve order of operation problems in tasks that require more than one step), and can even be used as memory and learning devices (by allowing the additional and removal of data from the global database).

True rule production systems haven't *quite* made their appearance in mainstream gaming, but a forerunning academic project *is* making use of gaming to improve their production system. *Soar*, a project started by Alan Newell, John Laird, and Paul Rosenbloom (the same Newell from GPS, mind you) as a test bed for Newell's theories of cognition, have been in use by the academic community since 1983. Soar provides an open source, ANSI C, general production system for cognitive scientists, and anybody else who wants to use it. See Figure 22.2 for a high-level

TeamLRN

Soar system overview. John Laird, after doing some Soar work with Defense Advanced Research Project Agency (DARPA) developing intelligent air combat agents, began experimenting with using Soar as a means for advancing AI performance within commercial video games. His team at the University of Michigan Artificial Intelligence Lab has successfully interfaced Soar with both *Decent 3*, and *Quake 2*, and created competent, nonscripted opponents for each game. Using a system of more than 700 rules, the team created a quake bot that could navigate arbitrary game levels, use all weapons and level elements (such as bounce pads and teleporters), and give good players a challenge. Also, the system performed planning, and so it could anticipate human actions, create custom level routes of travel to maximize the amount of powerups it could collect, and perform intelligent ambushes and hunting behaviors.

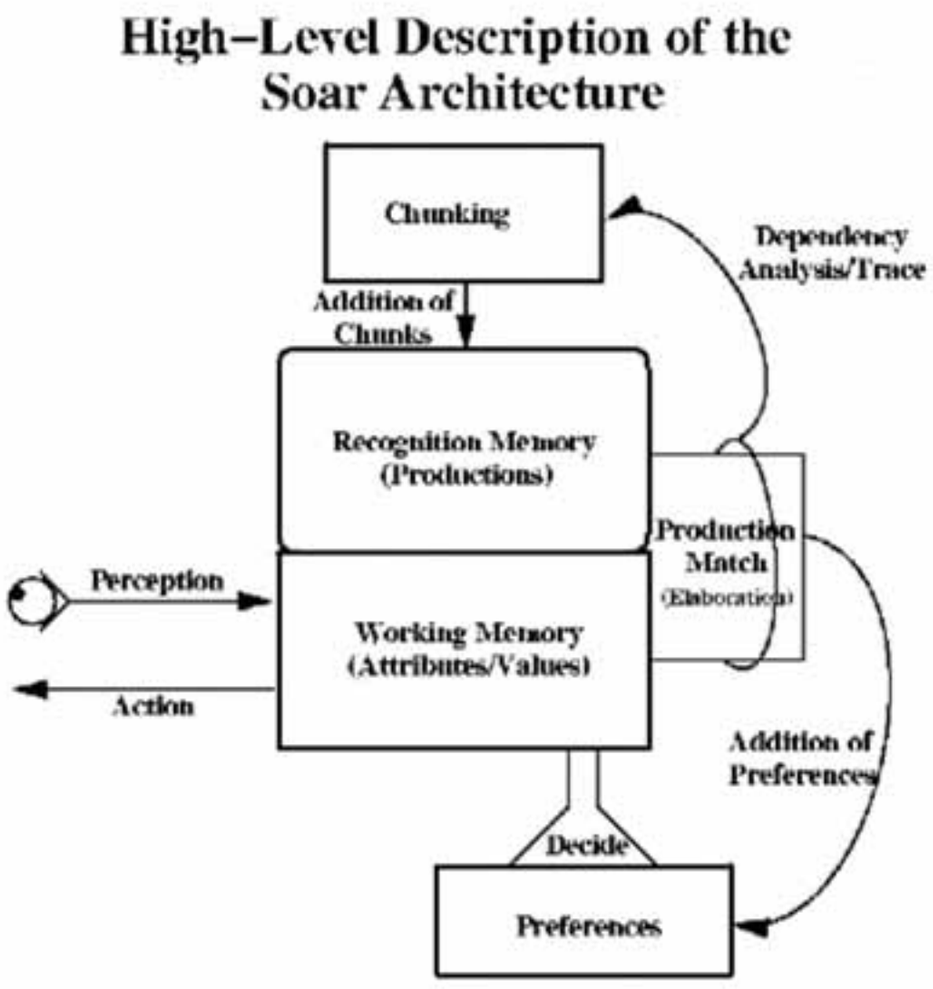


FIGURE 22.2 Soar architecture diagram.

Pros

- *General algorithm.* Again, like planning algorithms, a production system's decisions are data independent, so separate areas of the game can use production systems to provide disparate systems with rule-based decision making. This can be within separate databases, with different rule sets, or with any combination of sharing.

- *Research.* Tons of research has been done on production systems. Fast matching algorithms like RETE and TRETE (a stateless variation of RETE) have been created to dramatically speed up condition checking within a production system.
- *Goal directed.* Production systems are generally *goal directed,* meaning they pick an overall goal and find a way to make it happen. This creates a much greater illusion of intelligence than purely reactive behavior does.
- *Highly reactive.* These systems can be highly reactive and offer real similarity to human performance (with a good set of production rules).

Cons

- Also like planners, production systems can be computationally expensive, especially with games having a large rule set or nonarbitrary match collision resolution. If the game has to *find* matches and must also perform heavy calculation to *arbitrate* matches, the cost of using the system can be high.

Areas for Exploitation within Games

- A production system could feasibly be written in a highly data-driven way, so that new rules, perceptions, and the like could be added to the game world by simply adding to the game's data files. The production system would just perform the same algorithm given the new set of production rules. A system like this would be highly extensible and infinitely reusable.

DECISION TREES

Decision trees are another way in which commonly used code structures can be simply reorganized for greater flexibility and more functionality. Instead of having pages of `if-then` statements, you can implement each statement as a node on a tree and construct the tree such that you traverse the tree instead of a bunch of nested ifs. Figure 22.3 shows an example decision tree structure, representing the AI necessary to run a *Joust* opponent. The tree starts at a *root* node, which can also be labeled as the "question" node. What question is the tree answering? In this case, it is "What should I do now?" Note that our illustration diagram is a *binary decision tree* (BDT) because all of the answers to any given question nodes are Boolean, yes-or-no decisions. Special optimization algorithms, and even methods of reorganization after insertion and deletion are available to binary trees (because they are essentially red-black tree structures) that are not available to trees with arbitrary decision type.

TeamLRN

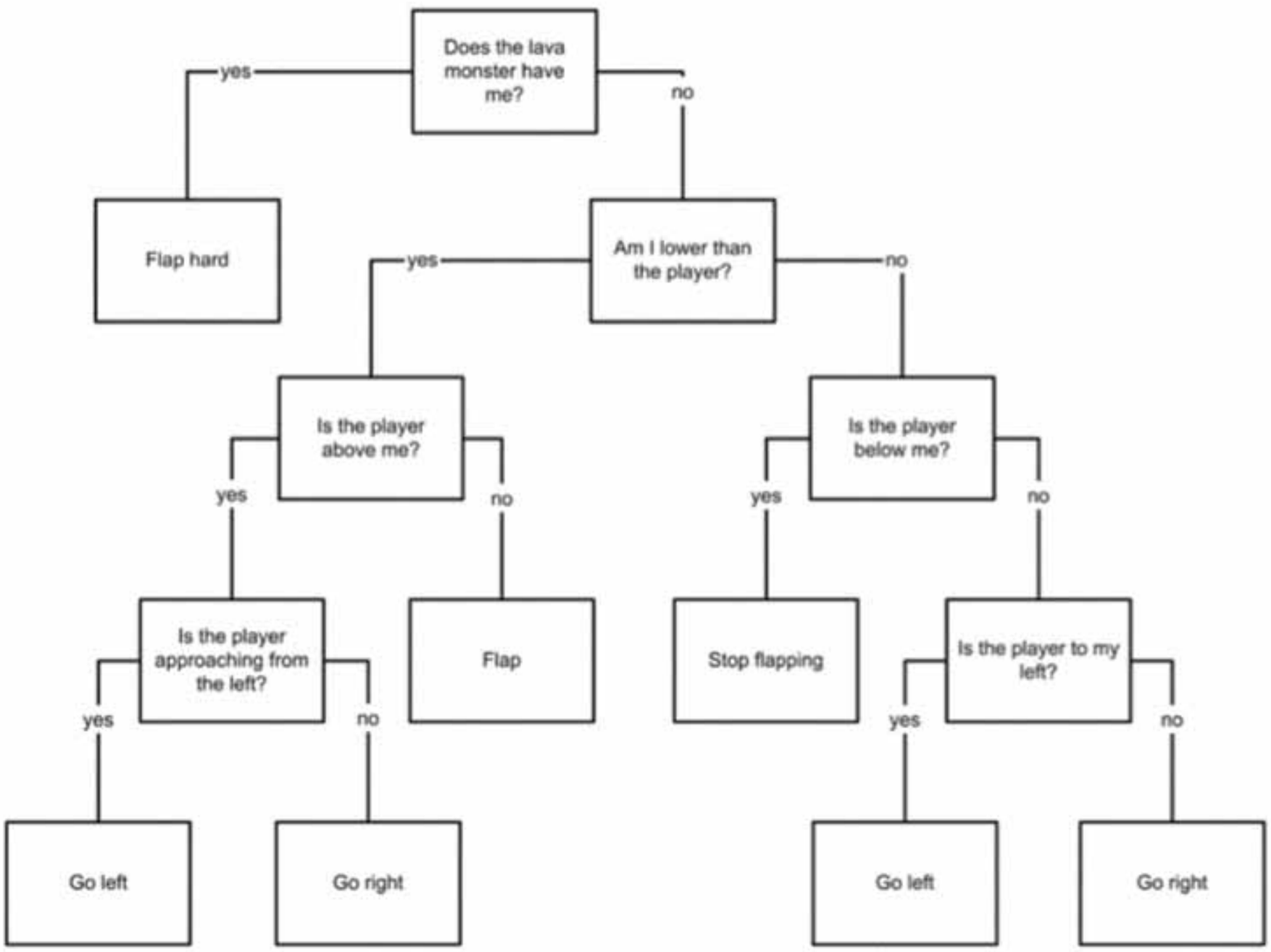


FIGURE 22.3 Decision tree structure for *Joust*.

There are two commonly used branches (bad pun) of decision trees: *classification* trees, and *regression* trees, which are both statistical methods that allow the construction of decision trees through algorithmic means by way of using a set of training data. Both methods can be considered a sort of "poor man's Neural Net," in that they try to generalize inputs to outputs. The differences between using decision trees and NNs is important to point out:

- NNs are a black box system; their internal weights cannot be meaningfully inspected and understood, whereas a fully "trained" decision tree and can be very descriptive and easy to understand.
- Decision trees can only comprehend *hard* comparison limits within a (typically) binary outcome. Because of this, their predictive ability is somewhat limited and rigid. NNs are precisely the opposite in that they excel at gracefully handling very noisy data or data with gaps and strange jumps in behavior.

- Output from decision trees is always discrete values, where output from an NN can be a continuous value if using the right activation function.
- Decision trees consider a single variable at a time; this is referred to as a *monothetic* algorithm. It may miss the case when multiple variables are weakly influential separately, but are heavily influential on behavior in combination. NNs are considered *polythetic*, in that they consider multiple variables simultaneously. This is one of the things that make NNs hard to work with (in that an NN may find polythetic relations where you didn't expect them within test data and adversely affect its learning), but this trait is also precisely why they are so effective in areas that decision trees are not.
- NNs are usually *much* more accurate than tree methods. A statistical examination of the relative error in a traditional example data set might be 10 times or more for tree-based methods over a backpropagated NN.

Classification trees are BDTs that work on categorical input variables, whereas regression trees deal with continuous input variables. BDTs do allow a combination of variable types, but most game AI problems that would call for this method will usually be one or the other. A classification task might involve trying to determine what "type" of player the human is behaving like in an FTPS game (hunter, sniper, purely defensive, berserk, etc.) and setting the AI to a specific chunk of code tuned to deal with that type of player. A regression task would be more along the lines of a prediction task, where the same AI system might try to predict the perceived difficulty that the human is encountering with the game, and then adjust its AI behaviors if this difficulty is too hard or too easy.

Pros

- Decision trees are easy to use and understand. This makes them perfect as a "rough pass" AI system that can take example data, and find logical connections where the programmers might not have seen it. It also provides this information in simple, logical rules. This readability also allows manual tuning, if necessary.
- There is plenty of research on performance and ways of improving functionality. Decision trees are a huge part of the world of statistics, as well as its progeny, data mining. Because both of these fields are big business, decision trees have been dissected and reassembled by think tanks all over the globe. Many tried and true algorithms exist for designing, training, debugging, tuning, and optimizing decision trees.
- Game AI problems are usually restrictive enough that decision trees actually make more sense than NNs do. The additional modeling power of NNs isn't usually necessary, and the readability of decision trees can be a huge boon to

TeamLRN

last minute tweaking and improvements (both common game development tasks which are almost impossible with NNs). Plus, additional complexity with trees can sometimes be attained with hierarchical decision trees. Any given node within a tree could be another whole tree, as long that the subtree evaluates to a yes/no decision.

Cons

- BDTs tend to be brittle because they are dealing with distinct states and hard-coded boundaries between them. So, like large quantities of `if-then` statements in general, they don't scale well and are difficult to maintain or extend once they get to a certain level of complexity.
- The size of your trees is a direct inverse correlation between accuracy and size, so if you need specific outputs, go with NNs (or some other method) instead because it will increase the size of your tree tremendously to provide high-resolution outputs.
- Decision trees tend to lack the *finesse* that makes more emergent systems like NNs so desirable. But many games aren't looking for finesse; they're looking for scalable ways of adding content.
- In nonbinary trees, there is no universal consensus that the additional complexity of multiple children in the tree will give you anything in the way of benefit when you consider the extra effort it is going to take to construct and use the tree. Specific cases within a particular game situation could set up whatever structure it wanted, but for a more generic solution that can be used across the AI system for various tasks, the standard framework is much more desirable.

Areas for Exploitation within Games

- Using a classification tree to perform simple player modeling, where you would *want* the AI to have broad categories because your responses are going to be limited.
- By data driving the trees, you could potentially insert or remove nodes from the tree, as well as adjusting the binary check parameters, using it as a form of memory or learning. *Black & White* used this to record the high level "thinking" that your avatar did about his environment, given his experiences and training, about which actions to take at any given time. This tree could change dramatically during the game.
- Having a general BDT system in your game can give organization and structure to several binary decisions, and by making these definitions data driven, you allow designers to make these binary decisions in the way they see fit. You might have a tree that determines who wins a jump ball situation within a basketball game. The same system could also be used to provide custom results

from binary decisions like "Can I get past my defender to get to the basket?," "Did the defender bite on the juke?," or even, "Did team X beat team Y in a simulated game?" Thus, designers could tweak a good part of your perception system, giving them another vector with which to approach the game design.

FUZZY LOGIC

We covered fuzzy-state machines (FuSMs) in Chapter 16, Fuzzy-State Machines." However, true Fuzzy logic is a far more advanced system, complete with its own logical proofs and methods. Fuzzy logic is a superset of Boolean logic that was introduced by Dr. Lotfi Zadeh of University of California—Berkeley in 1965 [Zadeh65] to handle the concepts of partial truth: values somewhere between *totally true* and *totally false*. Zadeh originally used the concept to model the uncertainty he encountered when dealing with natural language.

There are very few examples of true fuzzy logic being used within nongame applications, much less games. Fuzzy logic has been slow to catch on, until recently. The Sony PalmTop is reported to use a decision tree based on fuzzy logic to classify handwritten Kanji characters. Another implementation found its way into a prototype Mitsubishi car in 1993, where they showed an in-car safety system that studied the driver's normal driving habits. The car had built in radar system and could sense oncoming obstacles. The car would then try to decide whether or not the driver was responding to the threat, and if not, it would take over the controls to avoid a collision.

True fuzzy logic allows you to perform calculations on equations or rules using entirely fuzzy values, for example:

```
if Health is low AND WeaponStrength is lame OR Bravery is meek, then
Camping is high
```

In this formula, there are four fuzzy variables: `Health`, `WeaponStrength`, `Bravery`, and `Camping`. `Camping` is an *output* variable, the rest are inputs. There is also, associated with each variable, membership functions or *fuzzy subset methods* that determine relative fuzzy values: `low`, `lame`, `meek`, and `high`. These functions are specifically written to make quantitative measurements about the relative degree of membership in the variable's range. So, `low` can range from 0.0 to 1.0, depending on the value of `Health`. In determining the truth of this statement, each membership function is applied to its associated variable, to determine degree of truth. These truth values are then also subjected to the AND and OR operators defined in the rule, which in fuzzy terms are defined as:

TeamLRN

```
Crisp: truth (x and y) = Fuzzy: minimum (truth(x), truth(y))
Crisp: truth (x or y)  = Fuzzy: maximum (truth(x), truth(y))
```

Because of the ability to conclude logical truth given these fuzzy parameters, these kinds of rules can be used within a "Fuzzy-logic Production System," which follows all the rules of regular production systems, but performs all its inference using fuzzy logic instead. The general inference process proceeds in three (optionally four) steps [Kant97]:

1. During *fuzzification,* the membership functions defined on the input variables are applied to their actual values, to determine the degree of truth for each rule premise.
2. Under *inference,* the truth value for the premise of each rule is computed, and applied to the conclusion part of each rule. This results in one fuzzy subset to be assigned to each output variable for each rule. Usually only MIN or PRODUCT are used as inference rules. In MIN inferencing, the output membership function is clipped off at a height corresponding to the rule premise's computed degree of truth (fuzzy logic AND). In *product* inferencing, the output membership function is scaled by the rule premise's computed degree of truth.
3. Under *composition,* all of the fuzzy subsets assigned to each output variable are combined together to form a single fuzzy subset for each output variable. Again, usually MAX or SUM are used. In MAX composition, the combined output fuzzy subset is constructed by taking the pointwise maximum over all of the fuzzy subsets assigned to a variable by the inference rule (fuzzy logic OR). In SUM composition, the combined output fuzzy subset is constructed by taking the pointwise sum over all of the fuzzy subsets assigned to the output variable by the inference rule.
4. Finally is the (optional) *defuzzification,* which is used when it is useful to convert the fuzzy output set to a crisp number. There are more defuzzification methods than you can shake a stick at (at least 30). Two of the more common techniques are the *Centroid* and *Maximum* methods. In the Centroid method, the crisp value of the output variable is computed by finding the variable value of the center of gravity of the membership function for the fuzzy value. In the Maximum method, one of the variable values at which the fuzzy subset has its maximum truth value is chosen as the crisp value for the output variable.

These kinds of systems are in use within pattern recognition, financial systems, and other fields involving heavy data analysis. They are usually only implemented within systems that require a heavy amount of realism, since almost nothing in the

real world is black and white, perfectly aligned, and exactly positioned. In games, on the other hand, these kinds of conditions *do* occur, and so the additional computation isn't really necessary for most games.

A common misstep in understanding fuzzy logic lies in thinking that fuzzy values constitute some other way of thinking about probabilities. But this is an untrue line of thinking. An example that contrasts the two ways of thinking involves a set of objects called "Drinkable," that is a subset of objects called "Liquids." If you come across two glasses, one labeled "90% Drinkable Probability" and "90% Drinkable Fuzzy Membership," which one do you drink? The answer is the 90% Drinkable Fuzzy Membership. The 90% corresponds with it belonging to the drinkable set of liquids "almost completely," whereas the probability-based glass has a 1-in-10 shot of being *poisonous.* In fact, given the glass labeler's definition of drinkable, the fuzzy-labeled glass might contain cheap wine, or super tart fruit juice, or some other drink that he didn't consider fully "drinkable." Probability deals with statistical levels of chance that something *might be* true. Fuzzy set membership discusses to what extent something already *is* true.

Pros

- Extends normal Boolean logic to encompass more loosely defined variables, so that arguments with these types of values can still be mathematically proven.

Cons

- Computationally expensive; fuzzy systems that contain numerous fuzzy vectors can suffer from rule overload. Different methods have been created to combat this (like Combs method), but this remains an issue.

Areas for Exploitation within Games

- Handling game problems dealing with a large amount of unknown or partially known information, like player modeling, is a great job for fuzzy logic usage. With heavily strategic games, like RTS, civ games, or even poker, player modeling is important if you are striving to really attack the problem intelligently. If your system isn't going to cheat by giving it omniscient access to all in-game data on the human player, then the AI system is going to have to rely on perception data to build a model of the human to react. This type of information will most likely be sketchy, uncovered in small pieces that might not be in sequence, and may even contain falsified elements (the human trying to fool the AI), so a fuzzy system could very well be the best way to approach this problem.
- For online or multiplayer games, player modeling could actually be used by a game as a "helper" AI entity that could play the game for you temporarily, if

TeamLRN

you had to go to the restroom or answer the phone. This feature would be like pausing, except that you wouldn't interrupt the game for the rest of the players. The fuzzy system would try to model the way that you play the game and continue to use this style while you were away.

SUMMARY

- Artificial life techniques try to use lessons learned in biology to create emergent behavior from simple rules.
- Alife can create emergent, reusable behavior; it also suffers from unpredictability and brittleness.
- Alife techniques pertaining to flocking, other types of organic movement, and ecology building are possible future game considerations.
- Planning algorithms try to use additional information about a problem to decide what to do before you start, one hopes not making local mistakes.
- Planning is already used in many games, in the form of pathfinding using the A* algorithm, or some variant.
- Planning can provide AI systems with a large amount of additional intelligence, and are general enough to be used in different parts of the game engine. They can be CPU intensive and can make the AI sluggish if misused.
- Uses for planning in games could be anticipation of human actions, and a greater degree of intentionality in AI actions leading to combinations of behaviors.
- Production systems are generic methods of dealing with large amounts of expert knowledge about a specific subject.
- Production techniques are generic, heavily researched, goal directed, and highly reactive. They can be CPU intensive.
- Games could benefit from not only using production systems to organize their rule-based logic code, but could also be served by data driving the production system to maximize game extensibility.
- Decision trees are another way of organizing `if-then` structures using binary separation methods to optimize the structure and number of your variable checks.
- Both classification and regression trees are *similar* to neural nets, but have many important differences including readability, types of data they work well with, output types, variable consideration, and accuracy.
- They are easier to tune than some classifying systems, heavily researched, and very easy to implement. They can be brittle, don't scale well, and only provide discrete outputs.

- Games could use decision trees to perform simple player modeling, store memory like data, and provide a general, data-driven system for determining binary perception data.
- Fuzzy logic is a means of extending logic to encompass partial truth values.
- It allows games to use more loosely defined variables while relying on mathematically proven methods. Fuzzy logic can be CPU expensive.
- Player modeling, and helper AI systems are ways that games might use fuzzy logic systems.

TeamLRN

Part V Real Game AI Development

Although the artificial intelligence engine for each game can be unique, there are almost always common concerns during their development. In Part V of this book, we will delve into some of the development and entertainment issues that AI programmers must deal with in creating their systems.

Chapter 23, "Distributed AI Design," will introduce a method for breaking up the design of an AI system for any game, by dividing the AI tasks among a number of separate layers, called Distributed AI.

Chapter 24, "Common AI Development Concerns," will address a number of general game AI concerns at multiple levels of development: design, entertainment, and overall production.

In Chapter 25, "Debugging," we will discuss debugging and tuning issues as they pertain to AI programming, and we'll also introduce a handy Windows-based debugging tool that you can use on any AI project to help you find bugs and tune parameters within your game.

In Chapter 26, "Conclusions, and the Future," we'll wrap up the book with some short conclusions and a section on the future of game AI.

TeamLRN

23 Distributed AI Design

There are many individual goals within an AI engine no matter what type of game you're developing. Designing and creating an entire AI system is not a small undertaking. In this chapter, we will discuss a general design paradigm for approaching any type of large scale AI engine design, called distributed AI design, that should help you to break AI systems into manageable pieces. We will discuss the various parts to the method, and along the way, give plenty of examples. Finally, we will break down a classic game into different ways that it could have been coded, given a modern day AI engine.

BASIC OVERVIEW

In Chapter 2, "An AI Engine: The Basic Components and Design," we went over the various components of an AI engine (the main pieces being the decision making system, the perception system, and navigation). Now that we have covered the main types of coding techniques that are used in games, we can discuss more properly the different AI methods that work well with each section of an AI engine, as well as determine which pieces work well together.

The first rule of ALL game programming (as well as game design, most people will tell you) is the old standby: Keep It Simple, Stupid. The design and creation of an entire AI engine is not an easy task, and over engineering can stifle an already overwhelming experience. The purpose of the Distributed method is to simplify overall AI creation and maintenance, by spreading out the AI tasks into modular, as well as layered systems that work with each other to create rich AI behaviors without overcomplicating any one system.

A Real Life Example

To help with describing the distributed design technique, we shall present a real-life (RL) example. Say that you are sitting at your desk. Suddenly, somebody from the

TeamLRN

next room calls your name. You get up, and go to where the voice came from to see who called you. Let's break down what exactly happened, except that we're going to assume that you are an AI character:

1. You are sitting; performing a behavior called `DoingWork`. You received an input, in the form of a game event (your name being called), or a changed perception variable (such as `DistanceToNearestDisturbance`, or `BeingCalled`, etc.) depending on how your perception system is set up.
2. Your decision-making system determined that the incoming perception was important enough to change your behavior in favor of a different, more applicable behavior.
3. The new behavior you have transitioned to (or been assigned), called Investigate, first gives you a new target location (which is approximately the location of the incoming sound that called you), which was created by either some algorithm for guessing sound locations, or by cheating and telling you the location of the call.
4. You run your pathfinding algorithm to determine how to get to the target. The pathfinder gives you a set of points to move to in sequence to get to there.
5. Your movement code finally has a destination, and so it figures out which of your movement animations to play to look the best, and fit the motion closest.
6. You start moving to the nearest pathnode in the path list, but your desk is in the way. Your obstacle avoidance system takes over, and moves you around the side of the desk. By heading toward the next pathnode, and using the avoidance system to help you navigate any dynamic objects in your way, you make it out the door and toward the sound.
7. As you leave the room, it occurs to you that you that the door is open, and you have your wallet out on your desk. You stop and close the door, and then continue.
8. Ten feet later, you see Bob and recognize that it was his voice you heard.
9. This new input perception (that it was Bob that called) changes your state from `Investigate` to `ActCynical`, and so you say "What the heck do you want?"

And on and on it goes. The point being made is this: the first description we used was "I heard my name being called, and got up to see who it was." whereas the chain of determinations and levels of intelligent behavior that you used to do all of that is much more involved than that initial sentence implies. Such is the plight of our AI systems. Every task is actually a dozen smaller tasks, strung together in concert.

THE DISTRIBUTED LAYERS

Distributed AI design is the name this book gives to the technique of fully embracing the multilevel quality of behavior in the real world and applying it to the organization and implementation of an AI engine. We do this for a number of reasons: it produces cleaner, more maintainable code that is also easier to understand and extend. It spreads out the intelligence among a number of different systems, several of which will most likely be reusable by other elements within your game. Plus, you don't end up with a huge "AIPlayer.cpp" file (or its equivalent), where you store tons of special case code dealing with the AI characters. Instead, we will partition our intelligence into several layers:

Perception/Event layer. Filters incoming sense data for relevance and various other factors.

Behavior layer. Describes the specifics of how to perform a given action.

Animation layer. Determines which animation to play to fit the game state.

Motion layer. Handles aspects like pathfinding, collisions, and avoidance.

Short-term decision making. The narrow-view intelligence for the AI entity, primarily concerned with just the entity.

Long-term decision making. Handles wide-view intelligence issues, like planning, or team-based considerations.

Location-based information layer. Includes information transmitted to the entity from influence maps, smart terrain, or the like.

The Real Life Example Revisited

We shall step through the RL example to introduce the main layers of the system, to dissect the tasks our systems must perform into more manageable, layered levels of smarts.

1. The AI character's name is called. Let's assume this game is using an event system. So, our AI entity receives a message from the game stating that his name is being called.
2. Let's also assume the game is using a hierarchical finite-state machine (FSM) (with a state stack, to use as a memory) for it's primary decision-making scheme, as described in Chapter 15, "Finite-State Machines." The current state the entity is in at the start of the example, `DoingWork`, has registered for the "MyNameIsCalled" message. Getting the message sets up a state transition by pushing the state machine's current state onto the stack,

TeamLRN

and entering the `Investigate` state. This is an example of short-term intelligence because the interrupted behavior (`DoingWork`) wasn't being performed because of some personal perception, it was part of a larger (and within the HFSM, next level up) state, that of `Afternoon`.

3. Upon entrance to `Investigate`, the state calculates an investigation target for the character, based on the incoming sound data. The logic for this calculation is therefore within the action itself, and is thus part of the behavior layer.
4. The behavior then accesses the navigation layer by using the pathfinder to resolve movement toward its target.
5. With a valid direction to travel in, the movement layer is activated, and starts the move. The first thing it does is use the animation layer to choose the right movement animation.
6. We start playing the movement animation, but immediately something happens: another perception tips us to the fact that we're going to collide with a dynamic object in the environment (rather than a level element like a solid wall, which I would avoid using the pathfinder instead), and so the motion layer engages the avoidance system to steer us clear of it, as well any other objects on your way out.
7. When you go to leave, your wallet sends out a message that you've left it behind (through a smart object system, it is programmed to mention if you're more than a certain distance from it, and the `Afternoon` high-level state is listening for it because you don't want to lose your wallet while you're out during the day), and forces you to temporarily give up your current goal to take care of this problem first. The long-term decision system does some checking, and because your wallet is in your office, it will be safe to leave if you shut the door (computed with a simple planning algorithm). You shut the door using the behavior layer to set up the behavior, which in turn uses the animation layer to pick the right door-closing animation. You then return to your last state when this state is popped off the stack.
8. The perception system visually recognizes Bob (once you get within a certain distance, and you're both facing each other), and so sends out a message that the `Investigate` state is registered for "See Friend." The `Investigate` state intercepts the message, and checks to see if Bob's location is close to its investigation target. If so, it figures that it was Bob who called. Again, this logic is part of the behavior system, embedded in the `Investigate` state.
9. The `Investigate` state then transitions to a new state, and makes a smart remark, by way of the familiar behavior-and-then-animation chain of layers necessary to set up the action as well as the animation to play.

Now that you can see the basic flow, let's break down each layer by discussing the reasons behind separating it, the decisions that we're going to delegate to each layer, the techniques that can be used to implement it, and some more examples of how each is used.

The Perceptions and Events Layer

The reason for splitting out perception calculation was primarily discussed in Chapter 2. Creating a central perception handler is a great way to optimize your AI calculations. It helps prevent game values from being recalculated in multiple places within a single game loop, and consolidation supports the development and debugging of important game values being tracked by your AI systems.

The intelligence being assigned to the perception layer is in the form of the reaction times and thresholds inherent in each individual perception. All these kinds of determinations can be separated from the act themselves by being tucked all the way down at this low level. Thus, all behaviors that incorporate a given perception, either for activation or transition conditions, benefits from the embedded "intelligence" in how the perception is updated.

Centralized perception systems work very well within message-based systems, so when a perception actuates, it can send out a message to a specific player, or your system can broadcast a more general event. The perception system encapsulates some additional AI computation by way of incorporating attribute data within the perceptions as well. So, if I swing a sword in the direction of two different characters, and one of them has very slow reflexes, he might be in for a nasty surprise, but the other character (who has superior reflexes) might have no problem dodging, parrying, or even shooting me, depending on the character. All simply because his perception system picked up on the incoming sword and the other did not.

The Behavior Layer

Within most games, each behavior is most likely considered a state that the character is in, with more complex actions being constructed out of a few states. As such, behavior layers are generally coded within whatever system of "state" or atomic actions you are going to use within your game. Even if you're writing a fighting game, where the characters are in one game state the entire time (the equivalent of "Fight to the Death"), the final behaviors that you're going to be using with the characters (in this case, each fighting move) will still likely require special game logic to be embedded in them, even if it is only starting and stopping an animation.

The logic placed at this level usually involves transitions within the game's state machine (or whatever technique your game is using), as well as describing the series of actions necessary to perform the behavior from start to finish. In Listing 23.1,

TeamLRN

you can see some pseudocode for a fighting game behavior, in this case a large punch animation sequence, showing the series of events that are required:

LISTING 23.1 Pseudocode for a Fighting Game Behavior Called BigPunch

```
Begin BigPunch
    ForInit
    {
        DoSound(GRUNT_BIG)
        UseAnim(rand(NUM_BIG_PUNCHES)+ANIM_BIG_PUNCH_FIRST)
    }
    ForFrames
    {
        1       AllowCombo(off)
        2..5    TimeScale(1.6)
        6       OffenseCollisionSphere(1,on)
        7..9    SpawnParticle(FORCE_LINES)
        10      DoSound(AIR_SNAO)
        11..16  TimeScale(0.8)
        17      OffenseCollisionSphere(1,off)
        18      TimeScale(1.0)
        19..25  AllowCombo(on)
    }
End BigPunch
```

Here you can see the behavior setting up things like sounds, animations, spawning particle effects, and turning on and off various game flags. Because this is a fighting game, each frame of the animation during the behavior may potentially have some code or an event associated with it, because the tuning and balance of fighting games is so precise. The behavior in Listing 23.1 includes several types of flags and events, like allowing other moves to interrupt (for combinations to occur), scaling the local time of the character (to fine tune execution of animation data for dramatic effect), launching sound and particle effects, and toggling collision spheres (so that the attack will only "hit" the opponent during prescribed parts of the animation). Notice that this example assumes that all the big punch animations have the same number of frames, this is somewhat important if you're going to try and balance gameplay with generic behaviors, but obviously isn't necessary if you're willing to create the code necessary to run fighting moves with any number of frames. Instead, the system could internally keep track of the number of frames in the animation, and also note the highest numbered frame the script refers to, and determine what percentage of the total time each frame should represent. In this way, frame counts would be relative, and as long as the "stages" of each animation

were roughly the same (the first half is the windup, the third quarter is the hit, and the last quarter is the follow through, or some equivalent determination), then the big punch moves could be whatever length the animator wanted.

In the RL example, when the code transitioned to the `Investigate` state, it called the `Enter()` method, where it then determined exactly what location the character was going to investigate, by calculating a target based on information attached to the original `MyNameIsCalled` message, namely the approximate angle and volume of the call. Like perceptions, the behavior layer should also be influenced by player attributes, to differentiate different characters when they perform the reusable behaviors. If you tell two very different characters to `Jump()`, they should check their attributes to determine how high, so to speak.

For many games, this layer is implemented in code, especially when they have a limited number of behaviors available to their AI characters. For the opposite case, however, this is a prime candidate for a data driven solution. If you can use scripting to write the behavior code (as in the above fighting game example), or some other form of data driven gameplay, this will prove to be a real boon to your system, as it represents a fairly major chunk of intelligence being in the hands of the designers. The more content they can put into this layer, the more virtually "calculation free" personality and intelligence your characters will exhibit. Not that scripts cannot contain math, or be slower to run, but rather that scripting is a means of recording the common sense–style intelligence that your behaviors require to seem realistic from your designers and into the game through the scripting language.

The Animation Layer

In the old days, art resources came at a premium. The first *Super Mario Bros.* game for the NES had only 8K of sprite art for the entire game, including background tiles and all character animations. As games have become more complex, the amount of animation data associated with any given game character has risen dramatically. Main characters in action heavy third person games can sometimes have 5 MB of animation data alone, and higher. The process of choosing the right animation to play at any given time has become a seriously nontrivial task in many games, especially in animation heavy genres, like fighting and sports games. Fighting games might have dozens of unique moves for each character. Sports games, which generally rely heavily on motion captured animations to retain the signature styles of the simulated players, might have 100 or more different animations for a single action (dunks in a basketball game, batter warm ups in baseball, end zone celebrations in football, etc.).

Not only is the total number of animations high, but also, some of these determinations require expensive logic and mathematical calculations. Many of these might be simply randomized (with some checks to avoid possible repetition) like

TeamLRN

end zone celebrations, for instance, which are simply fluff (defined as unnecessary for gameplay) animations that you only have a large number of because they add entertainment value to the game. But consider the determination, in the same game, for "Receive Pass" animation to play. Your code has to take a number of factors into account: your current direction of travel and speed, the amount and position of coverage, the angle and direction of the incoming football, the kind of player the receiver is as well as his skill level (he may not be skilled enough to perform certain catches, or rarely catches on his left side), and the direction he's going to want to travel after he gets the ball. Then, when he gets a list of the available catch animations, the code then has to run through each of the available animations to determine if one of them gets the player's actual catching hands within a certain distance of the future ball position (being where the ball and the player will intersect), or does it come close enough that your IK (inverse kinematics) system will be able to take over. On top of all of this, there are usually special considerations, like the fact that you had to jump (to reach the catch before going out of bounds, because someone is diving at your feet), or that you want to catch the ball and then run straight out of bounds to save the clock, or because you collided with somebody on the way to the ball, and now need to recalculate everything in an attempt to recover from the hit. In a game like basketball, where almost everyone is an eligible receiver, and a heck of a lot more often, this kind of calculation can cripple an AI system's performance if not done cleanly, and with a plan in mind before implementation.

In games with lots of animation resources, or heavy calculation requirements (thus meaning that these calculations are almost assuredly going to require plenty of tuning and balancing, due to their complex nature), this is one of the places to start with a data-driven approach. Usually, animation selection systems are table driven, like in Figure 23.1, where we see part of a database-style file describing a selection table for finding the correct layup animation in a basketball game.

In this table (which, by the way, doesn't take into account people in the way, or an entire host of other factors), we have a number of layup animations. Each animation has parameters which show how it fits into the overall decision structure description, or schema. The schema for this system involves four parameters: shot type (the difficulty of the action, skill rating needs to be at certain levels to achieve hard or flash layups), angle of approach (the allowable angles of approach for the potential shooter, labeled as "a-h," each letter is a pie wedge in a circle radiating out of the basket, with "a" being straight left), the speeds the player is allowed to be in to use the layup, and the final quadrant that the ball will enter the basket from relative to the person. The final parameter, ball quad, could probably have used the same angle system as approach, but the final behavior that used it needed quadrant values, so the data was preprocessed to save time during gameplay. The second reason you're using a table is to save you the time of calculation in determining these

FIGURE 23.1 Basketball Layup Animation Selection Table

//Anim Name	//Shot type	//Angle of Approach	//Starting Speed	//Ball Quad
LayupUnder2ft	hard	efg	stand,walk	right
LayupBaseLtStand	norm	abc	stand,walk	front
LayupBaseRtStand	norm	fgh	stand,walk	front
LayupCtStand	norm	def	stand,walk	front
LayupCornerLtStand	norm	bcd	stand,walk	front
LayupCornerRtStand	norm	efg	stand,walk	front
LayupUnderStand	norm	efg	stand,walk	right
LayupBaseLtJump	norm	abc	all	left
LayupBaseRtJump	norm	fgh	all	right
LayupCtJump	norm	def	all	front
LayupCornerLtJump	flash	bcd	all	left
LayupCornerRtJump	flash	efg	all	right
LayupBaseLtRun	norm	abc	run	left
LayupBaseRtRun	norm	gh	run	right
LayupCtRun	norm	def	Run	front
LayupCornerLtRun	hard	bcd	Run	left
LayupCornerRtRun	hard	efg	Run	right
LayupUnder	norm	abc	Run	back
LayupRev	flash	abc	run	back
LayupRevTrick	flash	gh	run	back

things when the game is running (the first is also to save you time, but in programming). Technically, you could actually run through each layup and determine all of these factors algorithmically, but it's an expensive process, and so we'll use a table instead. Generation of the table could use an algorithmic solution, however. An "Animation Table Tool" could be written to recognize schemas, and then fed a pile of animations, which it then crunches through to generate these tables. Any touch ups or overrides could then be performed on the final file, but it would definitely save your designers (or you, if you end up entering the data) some work.

Nowadays, multiple animation channels are also very common. Two channels means that a character could be running one animation on his lower half (performing movement), while another on his upper half (aiming a gun, or throwing a

TeamLRN

football). Three or more channels could control whatever other parts or secondary objects (remember, we're talking about games here, your main character might have three heads or four robotic arms) your game design calls for. All of these additional channels would add to the complexity of your animation selection, but this really just translates to additional, or even nested tables if you choose to use that method of handling this problem.

Other implementations might include a scripted system like we described for the behavior layer, because in some games there are very little differences between animations and behaviors. The same kind of frame-by-frame control could be exercised, complete with launching of effects, events, and the like all from the animation's `Update()` function.

The Motion Layer

As stated in Chapter 2, navigation tasks are another huge part of any game AI engine. Pathfinding (which is a form of planning, as we saw in Chapter 22, "Other Techniques of Note") and its younger brother, avoidance, are very influential factors when determining a character's behavior. Like the other layers in this system they are separated because they need to be reused by other parts of the AI engine (specifically, any behavior that requires map movement), so it wouldn't make much sense to embed this kind of logic within the behaviors themselves.

Techniques for implementing the motion layer was discussed in Chapter 2. Because of its links to robotics, both pathfinding and avoidance have received a bounty of useful material and algorithms from the academic world. Even a casual search on the internet for pathfinding methods useful within games will yield thousands of results.

But what other logic and functionality should we embed at this level? The answer, again, is to try and establish character personality and attribute level within this layer. By giving the basic pathfinding and avoidance systems attribute-based behavior patterns, you can get subtle (and not so subtle) classes of movement out of an otherwise totally utilitarian system. Maybe bigger characters don't dodge like smaller ones do, instead, they go around, or just pause and let the obstacle pass first. Different types of characters might traverse the level differently, meaning good jumpers might take precarious paths, wallclimbers might go places others might not, and smarter characters will know exactly which teleporter to use to get right to the player. Another extension to this system might be additional means of obstacle avoidance that are game or character specific. If you're creature is as strong as Superman, is he really going to go around that garbage can? Or is he going to pick it up and throw it into outer space? Or kick it with a mad grunt, out of his way as he runs at you? It makes an AI system look stupid when a character that can jump 20 feet through the air following a pathnode system has to go around 6-foot-high crates to get at the player.

All of these kinds of things can be handled within the wrappings of the motion layer, and the rest of the system need not be bothered with it. Our overall goal with distributed AI is to have any layer be able to perform its job, and not have it negatively affect any of the other layers. Yes, avoidance is technically detouring me from my path, which is keeping my behavior of picking up the shiny powerup from finishing, but if it didn't, I would be stuck mashing my face into the tree in front of me, and I'd never get to the powerup.

Short-Term Decision Making (ST)

At the ST level, decisions involve matters that usually involve the specific character, either because of his attributes, his current perceptions, or his past experience. A character might be almost dead, and so has a very personal overriding goal to get a health powerup or run away. He might be dealing with the specifics of his weapon (he's out of ammo), or possibly his morale or bravery attribute is low, and is therefore grappling with the desire to run off into the woods to hide. We again split off this layer, to allow another branch for personality and attribute data to flavor reusable AI behavior. The ST layer can be somewhat touchy to allocate behavior to: the more ST behavior you have, the more individual the character's movement is going to seem, obviously. But if you're trying to model a tightly regimented troop movement, and a third of the units have their girlfriend on their mind, things might get messy. Of course, your ST system could be state based, to enforce group control during crucial periods, and allow for more varied behavior other times. In fact, given the more direct, situational nature of most ST decisions, they are usually created with a state-based system in mind.

Long-Term Decision Making (LT)

LT decisions, compared with ST decisions, are likely to be outside of any one unit, as they involve many units, possibly even all the units in the game. At the LT level, we are usually concerned with strategic types of solutions, which take planning, coordination, and timing. Of course, not all games need these elements. But even a game as simple as single player *Gauntlet* could be said to have a basic LT determination: the constant ticking down of your health as time passes. Whatever ST goals you have determined an AI *Gauntlet* player has set up for himself will all get set aside if his health gets too low, and he is forced to concentrate on trying to get more to stay alive. Of course, multiplayer *Gauntlet* has even more LT usage, because you can more easily dispatch the enemies with teamwork than with completely independent actions.

LT systems can be constructed in roughly the same fashion as the ST layer. State-based systems work well, because games with a good amount of strategy can

TeamLRN

usually be broken into wide phases of strategy. Chess has its opening, midgame, and endgame; RTS games generally have a buildup phase, and then several cycles of exploration followed by conflict. Planning is usually important to some degree in LT systems—almost by definition the most LT decisions need to take a wide view of the problem, and make bigger decisions that involve the future. Many times, an LT system will also make use of fuzziness as well, either with a fuzzy-state system (which works very well in RTS games; effectively allowing you controllers for the separate yet parallel goals of offense, defense, resource gathering, research, etc.), or with a full fuzzy logic system that is trying to make heads or tails of the limited amount of information it has received as reconnaissance data from the field to plan out what it needs to do to win the game.

Location-Based Information Layer

Sometimes, there's just too many things going on in the environment to really make it feasible to have each object in the world keep scanning the nearby area for things that it is interested in, and recording various perceptions about. Instead, we turn the tables. LBI systems create a more centralized, decoupled means by which data about the world can be accessed by individual game characters. Smart terrain and objects broadcast their existence to game characters (and to each other) so that what looks like complex interaction with the environment can be simplified and optimized. Influence maps allow for a wealth of information to be stored, sorted, computed, and analyzed for a whole suite of AI systems. LBI layer information is a separate system that you can think of as a large blackboard architecture for creating AI engines. An LBI layer can help the LT decision-making layer with terrain analysis that can find weak points in the AI's defenses or identify key areas of military interest in the map (areas that have high enemy foot traffic, or where many individual units have died), and discover valuable ambush locations in an FTPS deathmatch map. Alerting units to high concentrations of nearby enemies (acting as a rough "sound" perception, perhaps) could affect the ST layer's decisions. Pathfinding can use influence map information to avoid potentially deadly portions of the map, either by design, or because of a human laid trap.

One last way of using an LBI layer within your game is to embed triggers directly within your game world that are triggered by some game state, or merely by proximity. These triggers can set off events, messages, or entire game scripts that cause any number of things to occur. In this way, designers can tag specific areas of the world itself with "intelligence," which can help simplify other AI systems that might incur unnecessary baggage if they were to try to encapsulate these types of event data. Placement of these triggers is almost universally done within some kind of a level editor, although other systems have been used, like in game placement, or even simple text scripting of specific locations, although this last method isn't very friendly.

BROOKS SUBSUMPTION

The distributed AI approach is somewhat close to Brooks subsumption architecture, referenced in the first chapter. However, where Brooks designs are primarily directed toward modeling creatures with low intelligence (he hopes his robot designs to someday be as robust as natural insects: not that intelligent, but reliable in the face of sometimes an overwhelming number of adverse factors), the distributed method goes a few steps further upward in the chain, to bestow upon our AI systems a level of strategic planning and intelligence that goes above ant intelligence. The subsumption goal, of being able to achieve all of your goals in the face of a dynamic number and type of impairments, is definitely our goal when building our game AI engine. We would like our systems, as well as the behavior of the AI-controlled characters within it, to competently handle whatever the human player does within the game, as well as hopefully being able to recover within context to the adverse affects that might come up during any game situation.

Also, Brooks's notions of "reaching for insect intelligence first, and then on," is very poignant in the world of game AI. Our systems don't need to be super intelligent. The smarts of a housefly may just be enough to give the average gamer a run for his money in the twitch gaming world of the deathmatch arena. Ant colony-level mentality may be all that's required to impart an RTS AI system with all it needs to build up, defend itself, and occasionally conquer. In the end, we're not trying to win; rather, we strive to engage the player. Most people feel a twinge of pride, and relief, after finally smacking the fly that has been harassing them for a half hour, rather than feeling stupid for not being able to kill something with a speck for a brain. The fly engaged the player, performed surprising and mostly useful behavior which caused the human to focus his attention and energy, but in the end was not a match for the player; a nice lesson in perfect game AI behavior.

GAME BREAKDOWN

Goals

In this section, we will first break a popular commercial game into the distributed method, giving several examples at each level of how each AI layer could be implemented. Note, that this section is not describing how they should be implemented, for either efficiency or entertainment value. Rather, this is meant as an exercise into thinking about game AI problems from all possible angles, so that you can begin to truly understand the vast array of solutions that are available to any given game AI situation. So, we'll strive for the "most effective" AI, meaning that it stops the player the most, while confining ourselves to the spirit of the game by not adding any a-

TeamLRN

dditional animations or behavior types to the theoretical game. Otherwise, we would just say, "And then the guy whips out his rocket launcher, and fires . . ."

Secondly, we will discuss how an AI-controlled player might be created for each game. In this scenario, we are not trying to change the enemy behavior in any way, or affect gameplay. We are trying to create an AI system that plays the game well, and less importantly, like a human would. The examples given in this section are also not an indication as to the best way in which to implement the AI player, but are rather a broad spectrum view as to the types of solutions that are possible given the inputs and outputs.

Distributed *Super Mario Bros.*

The first ever side scrolling platformer (technically, other earlier games like *Pitfall* had many of the same elements, but didn't actually scroll); *Super Mario Bros.* is a true classic. It gave us the phrase "1-UP" (for a powerup that gives you an extra life) and took hidden game elements to an entirely new level, by making it a cornerstone of the game. It had 32 real levels, and 20ish hidden ones. The entire game was 32 K of code, and 8 K of graphics. That 32 K of code wasn't all code, either, because a good size chunk of it was taken up by the tile information necessary to construct the levels. Let's just say, space was tight. Almost every element of the design was space related. Mario wears a hat, to not have to animate his hair. He has a mustache so that we get the notion of a mouth, without having to display one.

Some familiarity with the game is required to make the below discussion worthwhile. The monster types that will be used for illustration later include:

- Regular creatures like the mushrooms (called Goombas), which could be squished by jumping on their head.
- The various kinds of turtles (or Koopas) could be disabled with one bounce, and then kicked, which would send them zooming along the ground as a projectile. In projectile form, they would harm both Mario and other enemies, and bounced off walls. Some Koopas in later levels had wings, and could fly. These took three bounces to kill, the first took their wings and turned them into a normal Koopa.
- The Hammer Brothers were special Koopas. They were taller than average, and always came in pairs. They would try to jump to the same level of the screen as you, all the while tossing dozens of harmful spinning hammers in a parabolic arc toward Mario.
- Lakitu is the name of a special enemy Koopa that actually rides on a small cloud along a line at the top of the screen during certain levels. He ducks if you get near him, and will somewhat try to hover around you. If you speed up and

try to loose him, he'll speed up as well. While he's around, he throws out special spiked turtles that you can't bounce on to kill. He's actually one of the smarter enemies in the game, besides the Hammer Brothers.

- Bowser, the end boss of each level, was a huge Koopa that breathed out large blasts of fire, and jumped up and down. Occasionally, he would also throw hammers like the brothers. Like every other creature in the game, he can be killed with a fireball from Fire-powered Mario, or by dropping him into the lava by releasing the drawbridge he stands on.

AI Enemies Implementation

As released, this game's enemies used almost no AI at all. The enemies were algorithmic (they traveled in circles or patterns, or performed a set behavior every so often). The most common behavior used by the enemies was that they would simply walk in the direction they were facing. An enemy of this type would be placed in contained areas and would walk forward until they hit a wall, and would then turn around and walk the other way. There were some uncommon elements that reacted to Mario's position (like the guy riding on the cloud, or the Hammer Brothers, and to some extent, Bowser the boss monster), but only to somewhat affect facing and movement. Overall, they were all but oblivious of Mario. They were merely well-placed obstacles to the advancing player as he navigated each level.

Perception and Event Layer

Mario couldn't do much within the game. As for movement, he could go left or right, jump, and walk or run. There were also a few "water based" levels where he used the same controls (jump meant "swim," but it was just an underwater jump), and the only thing that was different was the gravity on Mario. If you jumped in air, gravity pulled you back down pretty quickly. Underwater, you sluggishly drifted back down to the bottom, allowing your right or left movement to make it seem like you were swimming. Real running could only be accomplished when standing on the floor or a platform. Mario's status was one of four values: small, big, fired up, or invincible.

With this kind of limited action palette, the game's perception system can keep track of pretty much everything having to do with Mario, and do so quite frequently. Other perceptions that our AI might use include: the location of the current left-side border of the screen (the game always scrolled to the right, and because you can't scroll backward, we'll know how far the human can see knowing this, as well as what the player can reach on the left side), a calculation of how far away from Mario I am that is signed so that negative values mean I'm to his left, the location of any other nearby creatures, and maybe some additional complex calculations (like the distance from the platform I'm on to the one before it in the level;

TeamLRN

given this value, I can predict where an incoming Mario might land or the path he might take).

The vast amount of applicable perception data is about Mario, rather than the individual AI character, so it makes a lot of sense to centralize these perceptions so that all the creatures in the game can share the calculations.

Behavior Layer

Mario's behaviors may be limited in *Super Mario Bros.*, but the average enemy's set of behaviors is even more so. The behaviors that any given enemy can perform are almost the simplest imaginable: they involve either playing an animation or performing some simple movement. A few of the enemies can throw a projectile.

An adequate behavior layer for this game would probably be to create three behaviors: `PlayAnimation` (taking parameters for the animation name, as well as starting frame, the time scale, etc.), `Move` (taking parameters for the offsets, and a left/right direction), and `SpawnProjectile` (parameters would include projectile to use, speed, and path type of projectile: arc or straight line). There are only a few, so there's really no reason not to write them directly in code as a state that can be used within an FSM (which is probably what we'll use as our ST decision-making solution). Then, you could create inherited behaviors for the specific actions that game enemies use, like combination animations (the man-eating plants only behavior is to first play "Rise out of pot," followed by "Chomp several times"), specific types of `Move` behaviors (straight line, circle, bouncing, etc.), and projectile attacks (hammer, breath of fire, bullets, etc.). These inherited behaviors could also be coded within the game, or written using a simple script system (the configuration script system from Chapter 18, "Scripting Systems," would work fine) to define sets of parameters that would be linked to a specific behavior.

Animation Layer

There really isn't any real animation in the enemies that could be made better by modern animation selection routines. At most, they have two frames, which they oscillate back and forth between. Many have only one frame, and repeatedly mirror the sprite to give the appearance of movement. So we're out of luck here.

Motion Layer

Behaviors for *Super Mario Bros.* creatures are pretty much just setting movements, most of this layer will be sparse. We could do some limited pathfinding for enemies that are able to jump from platform to platform (as well as the flying creatures), so that they could hunt down Mario a bit better. Enemies are almost always localized (meaning tied to a particular area of a map world), so path networks could be local

as well, not requiring connectivity throughout the level. Lakitu on his cloud could use a special "path" network that represented the line in the sky where his cloud could travel. Potential fields could be used to do the pathfinding, but the overall movement required from the system is digital, and potential field systems tend to give movement a more smooth, organic look and feel.

The levels aren't completely static (Mario can smash almost any brick within his jumping range), so any pathfinding data that linked blocks together for connectivity would need to be updated as Mario broke connected blocks, but only if you are using this same pathfinding information for enemies that can jump from block to block.

Short-Term Decision Making

Considering the limited number of behaviors and perceptions available to the enemies, the decisions required of individual characters are relatively simple. Take the flowers that pop out of the flowerpots (which, in coding terms, means they play an animation). Then can only come up if Mario isn't standing on the pot, and usually follow a set time schedule. If we relax the timed interval restrictions of certain enemies for our theoretical game, then the game could be made almost devilishly hard by setting off these timed creature actions at the worst possible time: when Mario is in the wrong place, is vulnerable because of a bad jump, or has limited foot space to maneuver.

Because of the small action set that each AI enemy has (usually just `Walk`, and `TurnAround` if you collide with a wall; the most complex enemy has movement, jumping, and limited projectile attack capability), it's safe to say that we could use FSMs easily here. Most enemies would only have a couple of states, maybe three.

Another method might be to use scripting to describe the simple actions that the enemies use. A few sample scripts (written in a pseudo C style) for this hypothetical system are shown in Listing 23.2. Because the scripts would be so simple, the script writing tool could even be implemented within a larger level editing tool. The tool would allow you to construct the scripts, and then tag enemies placed within the maps with the scripts that they would use. You could allow the editor to run the game with your edited script, and immediately see the results of your changes. If your game had a console, you could even allow a designer to add or replace parts of an enemy behavior scripts from within the game. The last script in the listing is an "advanced" version of the original, where we've tried to use more perception data to make it especially brutal on poor Mario. This script tries to dodge incoming Mario jumps and harmful fireballs, and tries to keep the character from falling to his death by walking off ledges. It also tries not to clump up with other enemies, to create less space for Mario. Savage creatures indeed.

TeamLRN

LISTING 23.2 Sample Behavior Scripts for Enemies in *Super Mario Bros.*

```
//Simple Guy
Update:
    WalkForward;
End;

OnWallCollision:
    TurnAround;
End;
//--------------------
//Hammer Brothers
Update:
    If(MarioHeight > MyHeight)
        JumpUp;
    Elseif(MarioHeight<MyHeight)
        JumpDown;
    FaceMario;
    SpawnThrownHammer;
End;

//-Advanced Script-------------------
//Simple Guy
Update:
        If(MarioIsJumping)
        {
            //returns if Mario will land to my left or right
            dir = CalcMarioLandingSpot;
            //FindBestOffset finds the best spot right next to
            //where he'll land, so that he won't squish me on
            //the way down
            dir = FindBestOffset(dir);
            If(dir != MyDir)
                TurnAround;
            //make sure enemies don't pile up, spread them out
            //so its harder to land safely
            WalkForwardNoCrowding;
        }
        else
    {
        if(!FireBallNear)
        {
            FaceMario;
            If(!NearEdge)
                WalkForwardNoCrowding;
```

```
            Else
                TurnAround;
        }
        else
            DodgeFireBall;
    }
End;
```

Long-Term Decision Making

Usually, the LT layer handles, by definition, long-term problems. But, because of the quick, fly by nature of the game, an LT layer for this game would be concerned more with the other usage of LT systems, that of coordination of enemies toward a larger goal. The LT layer for *Super Mario Bros.* could be used to have multiple enemies work together to cut off all of Mario's methods of progressing. The LT system could monitor which powerups and hidden blocks are within the current scene, and allocate particular enemies to guard them, while placing other enemies at strategic locations, in places where Mario needs to land to navigate the level. Implementation could be another FSM working either in parallel, or one level up in the hierarchy from the ST state machine. In essence, the LT layer would assign tasks to particular enemies, and then their ST layer would decide upon the best way to perform that task. We're going to assume that individual enemies are only pawns of the system within the LT layer, and so a single enemy will never decide to not perform the LT assigned task, if certain conditions were true. This situation could come up in games where you allow individual characters, be they soldiers in a war, or bad guys in a fighting game like *Double Dragon,* to have morale or bravery statistics. If morale gets too low, or their assigned actions would lead to certain death, they might turn tail and run, instead of listening to the LT layer. But not here; in this game, everyone lives for the glory of the system, whose only goal is to stop Mario from getting the Princess.

Location-Based Information Layer

If we are creating an AI system for the game that learns over time, then an influence map could keep statistical track of the route that Mario takes through each level, which powerups he tries for, and the like, and this information could be used to position enemies for maximum detriment. Of course, this kind of information would only come after having Mario run through the level a few times. The nice thing about a system like this is that the AI systems would then work on somewhat arbitrary maps because it would be extracting strategic information from Mario's movements rather than from level designer–placed cues, like jump connectivity points and lists of placed powerup blocks. The system would also adapt over time:

TeamLRN

as Mario responded to the enemies' placements by changing his methods to get through, the enemies' placements would change also.

AI Player Implementation

For this part of the breakdown, we'll now assume that the enemies in the game are using the standard *Super Mario Bros.* "AI," and that we want to make an AI-controlled system for a Mario-like character that would allow him to perform all the things that a human player can do. A possible usage for this could be a game where you have another player that is controlled by the CPU, running through the level at the same time as you, in a competition to see who can get the highest score, or collect the most powerups, in a sort of thematic race.

Perception and Event Layer

The amount of input data that our character, Tony, has is limited. As he sees enemies, the perception layer would add them to a list (`m_nearbyEnemies`, possibly), and track their type (for predicting their movement) and position. A list of nearby visible (and known, but hidden) powerups would also be useful.

Behavior Layer

Tony has a few behaviors. He can walk, run, jump, and (if powered up) shoot a fireball. All of these behaviors can be constructed as very straightforward pieces of code except one: Jump. In *Super Mario Bros.*, as far as Mario is concerned, jumping IS the game. *Super Mario Bros.* is the game that introduced the concept of controllable jumps. You can hold down the jump button, for a higher jump (up to a certain max), but you can also *steer* your jump (using the direction pad) to a fairly large degree. These simple additions make the Jump behavior for an AI Tony a large undertaking.

Jump steering is accomplished by using two things: in air momentum, and the height of the jump. Figure 23.1 shows a diagram of the basic gameplay mechanic of Tony's steerable jump. When you push the jump button, the game takes into account your forward movement (initial momentum), and then starts adding to an accumulating value based on the direction you're pushing (positive value for forward direction, negative for reverse), and your height in the jump (the higher, the bigger the value), which is then used to affect the arc of the jump. To create a workable AI Tony, we have to teach him how to jump with this game mechanic.

Doing this can be done like any complex game behavior. You could find a series of joystick inputs that make Tony jump in useful ways, and use each jump specifically as a different behavior. So, you'd have a `LongestJumpPossible` behavior, and a `JumpToSingleBlockDirectlyAboveMe` behavior, as well as many more. You might even have specific jumps coded just for certain areas in the game. Certainly

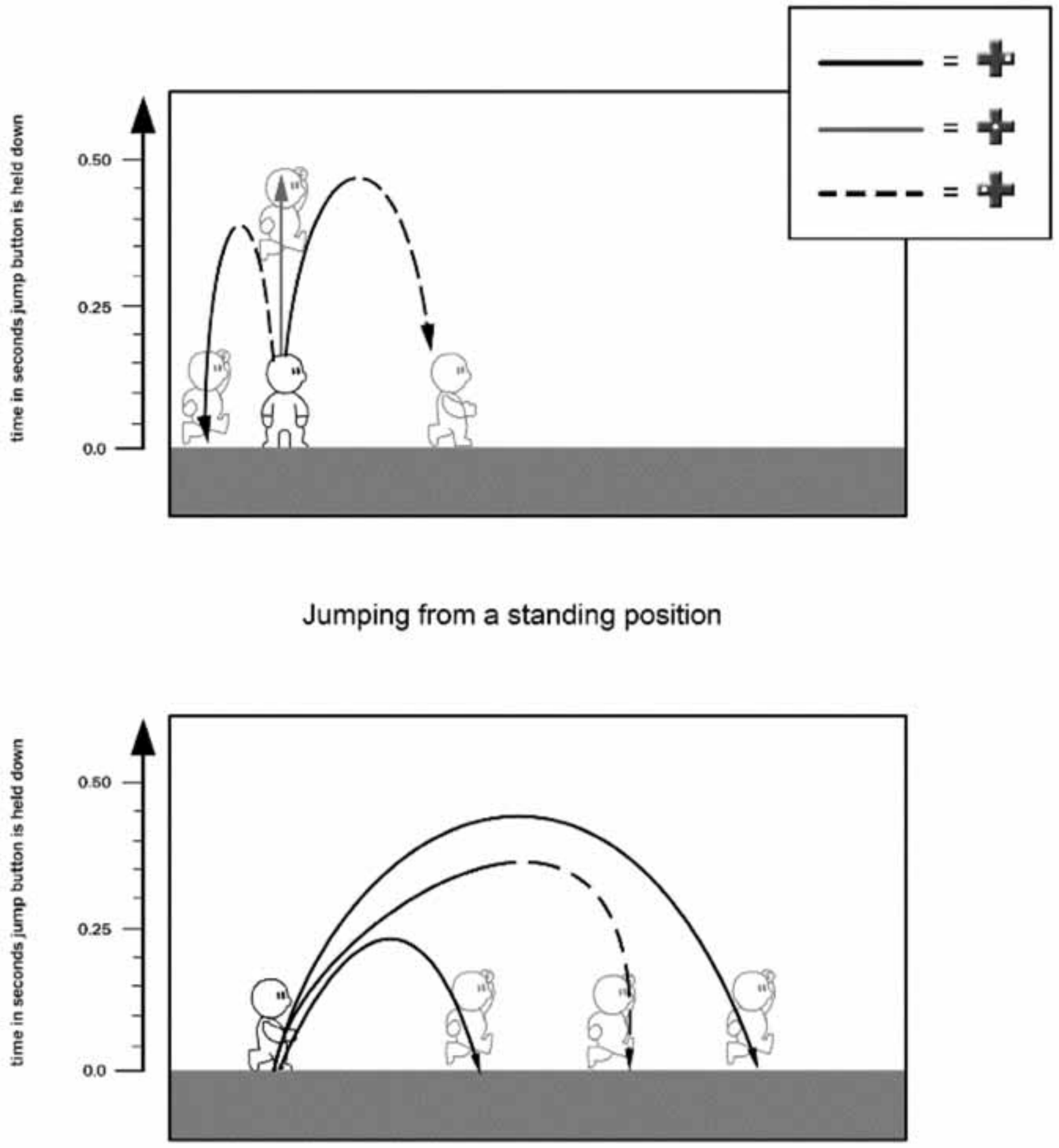


FIGURE 23.2 The mechanics of Tony's jumping.

this method would work, because there are only so many constructs within the game that Tony needs to get past. You could create 50 different types of jumps (number chosen arbitrarily), and then create a system for determining which jump to use based on the delta vector between where you are and where you want to land. You could even train a NN or GA to determine the rules for when to use each type of jump.

TeamLRN

Going further, you could also train a NN or GA to actually *find* the correct series of joystick inputs to navigate a specific jump vector. This might take a while, because you're talking about many rules (50 in our example), with each rule possibly taking 10–30 steps (assuming 30 frames per second, that a jump animation is a full second long, and that you can steer the jump during the entire animation). Of course these numbers might be reducible; you would need to experiment to determine exactly how many rules and steps for each rule were necessary.

Animation Layer

Like the enemies, the number of animations that Mario had didn't require the help of an animation selection system. The biggest variable in player movement was the jumping, so different jumps could have been animated for the numerous styles of jumps. But with the human controlling Mario, the game doesn't know when he's going to let go of the jump button or quit steering a jump. With an AI Tony, however, we would know ahead of time the exact jump we were going to attempt, and trigger specific jump animations that would look cool for the different types of jumps. If Tony had 20 different jump animations, 10 styles with 2 each, then a very simple animation system could compute which one to play based on the style of the jump, and then some random check to mix up which animation (out of the two equivalent animations for each style) is displayed. Other places for animation selection would include playing transition animations, so when you stop pushing forward suddenly, Tony would play the correct braking animation, or would react to repeated jumps with custom animations.

Motion Layer

Pathfinding would be necessary to give our Tony a sense of where to go, although the 2D nature of the game would drastically simplify the task. Paths would be mostly consistent of jump connectivity between platforms. We could even preprocess the necessary jump style to use to jump from one platform to another, and encode this data within the pathnodes themselves, to help optimize the game code from having to make these calculations during run time.

Pathfinding could even be performed in a sort of potential fields implementation, in which each platform location stored the force vector necessary to get to the next platform. If the platform was just a floor, connected to the next section, it would give a "walk" vector, otherwise it would give "jump" vectors. Sections that have both a walk and jump vector are near the edges of a platform, or have multiple successive places that can be reached. To find his way to a powerup block, or some higher up platform, Tony would search backwards through the potential fields graph (from his target to his current location) to find a series of movement vectors that would get him there.

Obstacle avoidance could be done a number of different ways, dependant on Tony's goals within the level. If he didn't mind skipping past some enemies, then the presence of enemies could invalidate certain paths, which would make him take alternate means to get through the screen. If he was a bit more bloodthirsty, we could note each opponent in turn, and if killable, he could plan a jump vector that would bounce him onto the nearest of them, and he could even try to compute bounces that would then daisy chain bounce him onto another enemy, giving him bonus points.

If Tony was using the potential fields method, then the enemies could also emit negative potential fields, making Tony jump whenever he got too close to them, automatically. Tony usually has to jump *before* he gets to a given creature, so the potential field effect would have to be offset from the actual creature's position. A motion layer done this way wouldn't really have to be aware of the enemies of the level, because they would just affect how he traversed from one platform to another. Care would have to be taken, though, that Tony wouldn't automatically jump over an enemy, and fall to his death in some cavern.

Short-Term Decision Making

Short-term concerns might be things that only deal with the screen of space that Tony is currently on. Things like getting powerups, bouncing on enemies in a specific order so that he minimizes his danger, and making his way to the next screen can all be handled by the ST decision layer. A simple FSM (with a few states, like `EliminateCreatures` and `GetPowerups`, would probably suffice), especially if we've put a lot of intelligence within the other layers of the system.

Long-Term Decision Making

LT decisions might include time-based considerations (forcing Tony to take a pipe shortcut through a level, to make the time limit), or specifically taking a certain path to get a large cache of coins and the extra life with them. Tony has one basic area in which he needs somewhat advanced "planning," that of determining the best way to break bricks to then get to certain well hidden secrets. There are points in *Super Mario Bros.* where Mario needs to break a brick, which gives him access to the one next to it as a step, and then break out three more that are up a level, followed by leaping across a small pit and breaking out one on that side, to *then* bump the secret block that sprouts a vine, taking him to the warp zone. That's quite a bit of planning for such a simple task. Although it could be done (as a classic planning algorithm, where you would have the state of the screen, and operators like `Jump`, `BreakBrick`, and so forth that would allow you to test different plans to find one that would work), this would be quite computationally expensive. So, because

TeamLRN

we don't have arbitrarily constructed levels, we can precompute how to achieve certain secret spots within the level, using small scripts of which blocks to break in what order.

Location-Based Information Layer

There are times when Tony pushes forward, and doesn't see any enemies. So, he jumps to the next platform, only to have the screen scroll, and land right next to an enemy that he didn't know was just off screen, killing him. If we implemented an influence map for all the floor space in the level, using occupance data that stored how many creatures, if anybody, had been on each patch of floor for some set time period. Thus, the occupance data would sort of *linger,* leaving a ghostly trail of occupance behind each creature that would gradually fade away. Having this kind of system, Tony would check were he was going to jump to, and note if anybody *had* been walking on the platform, but had just returned to the right of the screen. He could even, by watching the values decay over a few frames, determine the direction and speed of travel of the enemy. A system like this would virtually stop surprise run-ins like these, except for one-shot monsters that don't move until they get on screen.

Tony could definitely benefit from a smart terrain system. Powerups could tell him of their nearness, and even give him directions for how to get there. The different level elements (such as flower pipes, super jump pads, etc.) could also telegraph their location and use to Tony, so that he could use them "blind," as it were.

SUMMARY

Distributed AI design is a way of splitting up AI tasks into chunks that not only allow for a separation of functionality, but also as separate platforms for adding personality and individuality to AI-controlled characters. It helps AI systems add richness of behavior while providing reuse of code.

- Distributed AI design allows difficult AI problems to be broken into manageable chunks.
- The layers within the distributed method include perception, behavior, animation, motion, short-term decisions, long-term decisions, and location-based information layers.
- Distributed design adds to the Brooks's subsumption architecture with the inclusion of advanced strategic layers, as well as cooperation layers for dealing with multiple AI entities into a larger picture.

- The chapter fully dissected the game *Super Mario Bros.*, both from the perspective of trying to improve the enemy and Boss AI, as well as in the creation of an AI-controlled Tony character.
- As you can see by their sheer number, not all of the techniques talked about during the classic game breakdown are simultaneously needed or even compatible, but just goes to show that the number of ways that we could code up the enemies and an AI player for *Super Mario Bros.* is quite diverse indeed.

TeamLRN

24 Common AI Development Concerns

In this chapter, we will discuss many of the things that tend to come up as concerns in almost any AI engine development. Some of these are design considerations, others are entertainment related, and still others deal with production issues. In general, this chapter hopes to provide insights into some of the considerations that might flavor the implementation of your AI engine, as well as the AI-controlled entities themselves.

DESIGN CONSIDERATIONS

In this section, we'll look into a few areas of AI engine design and implementation that you should think about. These topics can not only shape the direction you might take in designing a particular game AI system, but also in the details of implementing a system.

Data-driven design considerations. Common snags to look out for if you're trying to decouple your game data or logic from your engine.

The one-track mind (OTM) syndrome. This disorder affects some AI engine designers, who think that they should find one AI method and stick with it, no matter what.

Level of detail (LOD) AI. Useful ideas for implementing an AI engine that allows for this optimization technique.

Support AI. Other uses for AI on a game project that you might not have thought of.

General AI design thinking. Just some general ways of thinking that you should give a try when you're going about AI engine design.

General implementation ideas. These are notions that are good to keep in mind during development, as well as general rules of thumb when coding AI systems.

TeamLRN

Concerns with Data-Driven AI Systems

As the content level of games continues to rise, it becomes more and more of an imperative to get additional hands into the internals of the AI systems. Designers are increasingly requiring an immense amount of content from a behavior standpoint (meaning, the definition of *possible actions* that the AI can perform) and are thus also increasing the size and complexity of the game AI logic (or *when* to perform the actions). Actual game behaviors are much more tightly coupled to game code (they might possibly have to access the animation system, deal with physics, spout game and sound effects, and link into any in game communication details like the messaging system), so it is much harder to get the definition of behaviors into a system that a layperson can use. Generally, data-driven design involves the logic side of AI development because of this. In a scripted AI engine, basic behaviors for the various characters are presented to the designers as keywords or functions that they can call from their scripts. In a visual system that defines state machines (or other structure), you might have individual action nodes (that are representations of your game behaviors) that designers link into a flow diagram.

On way to go about determining how to achieve the level of data-driven design within your system is to data drive the highest level of the system. Then monitor the usage of the tool, and if the designers are constantly bugging the programmers to add elements at the next level down, *then* data drive that layer, repeating this as necessary. This follows the same model as the most classically data-driven system within modern 3D games, the renderer.

A 3D renderer is a software system that knows how to draw a number of very low level, completely reusable *primitives.* These primitives come in layers: the lowest would be *polygons,* which combine to make *models,* which then combine to form *scenes.* At each level of the rendering pipeline, objects are completely constructed from objects of the level below it, and each level is pretty much independent of the others. Whenever making a system data driven, the evolution from "fully hard coded" to "fully data based" is always from the highest level primitive to the lowest level. To continue the analogy, some of the earliest 3D renderers were scene definition scripts. *POV-Ray,* a ray tracing engine from the 1980s, is a prime example of this. *POV-Ray* had a number of primitives (toroids, spheres, heightfields, and many other shapes that can be defined using a mathematic equation). Rendering things with *POV-Ray* involved writing a little scene script, and then letting it run. Many young graphics experimenters got their first taste of making something visually impressive come out of their computers by using POV-Ray. But you can't define a model in *POV-Ray*, because it only renders mathematic constructs. A figure that looks like Crash Bandicoot would be somewhat tough to mathematically model. So, our young experimenter likely found himself a mesh modeler at some point later on. These modelers use a different set of primitives

(three-sided polygons called triangles or *tris*, and four-siders called *quads*) to define models, which could then be rendered using a different system, a mesh renderer, instead of mathematical rendering. This is where we currently are in 3D modeling. The reason that the polygons themselves don't require a data-driven design is that they're so low level, *so* primitive, that there really isn't any need to go further.

In game AI, you could think of the analogous primitives to be *animations*, which combine into *behaviors*, which combine into *strategies*. The first step in data driving the system, then, is to encode a system for designing the game strategies. Here's where games of today mostly stop. Using (usually) either a script of some kind, or a table (depending on preference, really) we construct systems where designers can set up the game rules, the state machines, the fuzzy parameters for player attributes or the probability of behaviors, and the specific decisions to use for transitions, success or failure. But this is still all the first layer of the data driving of AI systems.

If you find that your designers are continually asking for additional behaviors, then the next step is to allow nonprogrammer definition of behaviors. What this requires is to come up with a set of primitives that all behaviors are built from. Commonly, these primitives are thought of as translation within the world (movement), some sort of physical action (animation), and game events (to send information, or launch a sound or graphical effect). If your game uses translation data within the animations to move characters (instead of just playing a run animation, and then sliding the model along the floor at a speed that mostly matches the movement, like *The Legend of Zelda* and *Quake*), then movement is also an animation. Even if you are sliding the characters physically, the movement speeds can be built into the animation that will be playing during the movement. Thus, we can data drive the behaviors in the same way that we did the strategies, using *animations* and *events* as our primitives. You can define a state machine that moves from one animation to another, occasionally spawning a game event, either from a script or a table, just like the strategic AI layer.

If you're finding that this *still* isn't enough, and that your designers are requesting constant animation changes, you might have to go still further, and give the designers the ability to build the animations themselves. While this isn't commonly done (because animation also includes art direction issues and the like), fighting games in particular have used this level of data-driven design. Animations are built out of the next primitive down, which is called a *frame* of animation. A frame represents a snapshot "pose" of the character's body position and movement for a very small slice of time. Not only do fighting game designers change parameters and send events keyed to specific frames, but they can actually construct combination animations by playing a few frames of this animation, then a few frames of that, and then some from here, blending between them if deemed necessary.

TeamLRN

Data-driven design is not a magic bullet; it won't universally improve any game or system. The level of organization needs to go up as the level of data driving goes lower. Because the size of the data necessary to define all the possibilities goes up dramatically the lower in the system you are, data bloat and other issues come to bay. Make sure that you are not data driving areas of your game that could be done better with a code-based solution. If your animation system is completely data driven, but all the animations are using the same script except for one, then you are being pretty wasteful, and you're much more likely going to be better off, in terms of both performance and data size, to use a code-based approach and special case the one exception to the system.

The lesson is, only support the amount of data-driven design that is required by your game, and even then, make sure that you're following the formula: create the reusable, simple primitives that allow users to build more complex objects, and no further. If creating those primitives becomes an issue, then go down a layer in complexity, and data drive that step as well.

The One-Track Mind Syndrome

A semicommon problem that affects some AI programmers is to get hung up on a particular technique, and to apply it by rote to every AI problem they come across, barely giving thought to its relevance. One of the central problems with AI tasks is that most are very context sensitive; they really only work for specific types of problems, given certain fields of input, and under particular game conditions. For an even moderately complex game, there aren't really *any* game AI techniques that will be the one and only you'll need to get everything done in a clean, scalable, and manageable way.

One of the most common traps that AI programmers (especially new AI people, or even senior staff in other game areas that start poking their heads into AI) fall into is the "State Machines are All You Need" trap. For good reason, a nice-sized swath of people find state machines easy to understand, and like the way that they break down problems. They then promptly lose their minds, buy a ring, and marry them. Now, FSMs can go a long way in the game industry. It's true, you probably *could* program 80% of games on the shelf with state machines, especially with programmers that don't mind 100 hour crunch weeks, dredging through pages of nearly indecipherable state machine logic stuck in huge switch statements with somewhat arbitrary priority systems, all because they once again forgot that state machines don't scale very well, and tend to get harder to maintain the more states you add to them. FSMs are like a hammer. You can't build a house without your hammer. But building a house with *just* a hammer is a whole lot of work indeed.

An AI engine doesn't have to be all encompassing. You can use an FSM for the basic state layout of your game (front end, introduction, gameplay, gameover), a

FuSM for the main short-term decision layer, a simple planner to run your path-finding and long-term decision layer, and a scripting system to data drive animation selection and run configuration scripts for the decision layers. Your perception system can be humming along in the background, sending out messages to all of these systems, and keeping the wheels turning. Sound complex? Well, as a whole, it is. But piece by piece, what you've really created is a straightforward, modular, scaleable system that can be the basis for any number of titles. It can handle all the problems thrown at it by the game, as well as last minute game ideas, and tuning crises that might come up because of focus testing. If something comes up that is completely unplanned, something that it can't handle, it's flexible enough to incorporate another module, without breaking the "tender, fragile balance of the house of cards" that many large scale completely FSM or completely rule-based AI systems become over time.

Level of Detail (LOD) AI

AI systems are generally strapped for resources. Polls at the Game Developer Conference over the years have seen a slow rise in CPU allocation for AI in games, from about 2% in the mid 1990s to around 10% now. Certain genres, like turn-based games, obviously blow this metric, but these numbers are considered averages.

Just like our graphics systems, one way in which to quickly tame rampant processor usage of AI systems involve using LOD AI. The different LODs are dependant, just like graphics, on the player not being able to see the shortcuts being used. A typical LOD list might entail:

- *Off screen and faraway.* Characters in this category are completely nonexistent to the player.
- *Off screen and close.* These AI characters cannot be seen, but the player might still hear them, notice doors closing from someone passing, etc. Many games don't use this determination, in that they continue to treat the character as close by.
- *Very far off.* A character in this LOD would be visible as a pixel or two.
- *Far off.* Characters are now visible as solid colors, and possibly shapes, but no real detail yet. You can tell the difference between a monster and a humanoid, and tell a truck from a car.
- *Medium.* This distance would be your true area of sight, determined more by the camera angles used in your game, as well as where the depth fog starts to clear. A good distance might be somewhere between 40 and 70 yards.
- *Close.* Anything closer than Medium is considered close.
- *Interaction.* This distance implies that the character is actually interacting with the player in some way.

TeamLRN

There are a few ways to handle the AI as it changes between the various levels. One is to actually run different AI routines. This is akin to the graphics practice of generating lower polygon models for LOD models that will only be seen from very far away. If you're scripting your AI, then you could just have different AI scripts for the LODs, or the functions buried *within* the AI engine that the scripts are running might perform LOD checking. A character using dynamic obstacle avoidance when the human around is great, but when he's off screen or very far away, we can forget about this sometimes costly step. That is, as long as we can include the caveat that players have to be able to fix themselves if they get into strange positions, and then the human player comes back. But from the character's AI system, he would always just be calling `DoAvoidance()`, and the avoidance function itself would query the LOD system to determine whether or not to perform real avoidance. The only real problem with this technique is also shared with its graphics analogy: you are usually required to write other versions of the script, or code, or database. In short, you multiply the amount of implementation and debugging work to create any behavior in a system using LODs, because you need to support the multiple routines.

Another way of handling LOD in AI systems involves varying the update frequency of the AI engine's `Update()` calls to specific areas. For characters within the human player's immediate area, AI decisions might be updated very frequently, upwards of 10–30 times per second. For off screen elements, this might fall to a figure more like 2–5 times per second, or even less. Nonessential behaviors (so called *window dressing*, because it serves no purpose other than looking pretty) can be reduced to not updating at all, if need be. When setting up these schedules, try and *load balance* your update calls, so that you're not updating all of a specific type of unit at LOD level 3 every 15 game loops (or ticks). Rather, offset their starting update time, so that each individual unit still only updates every 15 ticks, but only a *few* units of that type update each tick.

As an example, we'll discuss an NPC character in a 3D FTPS game. The NPC is a scientist, and is running a small AI script that has him continuously moving between one of three workstations, playing a different animation at each one, to appear that he is working. Here we have a character that is adding nothing to gameplay other than contextual movement within the world. The ways that the different levels of detail would affect his behavior would be:

Off Screen

When he's not seen, don't run his AI at all, because he's not influencing anything. You might want to continue playing his sounds occasionally if he's close to the player's location, but not if there's a nicely sealed lab door between the player and him.

Very Far Off

Stand still. The view distance is so far that even moving the character is unnecessary because it will likely only translate to a few pixels worth of movement anyway.

Far Off

Slide the character (don't animate, pathfind, or avoid: just slide) occasionally back and forth in a straight line between his equipment. You'll still give the illusion of activity within the world, without all of the work. If the scientist were an acrobat instead, and was supposed to be performing cartwheels and huge aerial leaps, we couldn't just slide him, but all this means is that we'd just have to use a different technique for an acrobat at this LOD.

Medium Distance and Closer

For this distance, just run his regular AI because this particular guy isn't performing expensive calculations. Which brings up a good rule of thumb: try to get your designers to give nonessential game characters very few advanced AI tasks. Our scientist shouldn't need to use pathfinding to get between his equipment, and shouldn't be running any unnecessary perceptions.

As another example, let us consider an enemy civilization in an RTS game. These games are a bit different, in that the position of the player isn't really a set place, and *by design* the human player isn't going to be seeing the majority of AI units for large chunks of time because of the limited amount of world that can be seen at once, as well as the Fog of War. A sample breakdown for the different AI LODs in an RTS game might be:

Off Screen and Far Away

All strategic AI continues to run (at all LOD levels), but at adjusted update rates. Another possible optimization might be that the AI has limited updates, but when a major event happens (for example, the AI reaches the next major upgrade in the technology tree), it gets a short time to update. However, tactical decisions and actions are dramatically simplified. Note that being under the Fog of War is equivalent to being off screen.

Movement can be done by just sliding the units along at some set speed in the direction of their target, and when within some range teleported to the exact final destination, or they could be left idle until the travel time is up, and then teleported the entire distance. Unit to unit collision could be simplified, or even ignored during this LOD (although, you would have to ensure that two guys aren't sitting on each other before they do become visible again).

Actions, especially combat, could be determined using statistics instead of actual fighting. Other actions, like building structures or mining resources, would

TeamLRN

also be statistically determined (so, the game wouldn't have to move peons back and forth between a gold mine and a resource center, but would instead determine the time it takes to get some amount of resource, and just apply a timer to a set resource increment).

Off Screen and Close

Because "close" means likely contact, this LOD should probably be close to full AI, except for some visual elements. Tactical units might still play sounds, if not under the Fog of War. Animation selection is obviously ignored.

Very Far Off to Close

These LODs would be roughly the same, AI wise. What is "on screen" in an RTS game is usually much more localized than other genres, and the camera angle is almost always limited to a restricted, semioverhead view (*Myst* used a more general system). Because of these two facts, you don't have the degree of perspective, where you see characters as specks far off in the distance, like you do in other games.

Interaction

At this level, everything is on, and the AI is updating at the highest frequency level that it needs to make intelligent decisions.

Support AI

Sometimes, you must design elements into your AI engine that don't have anything to do with the primary gameplay in the title you are developing. Other areas of the game can still benefit from AI techniques. Things to consider in terms of secondary AI systems include:

- *User interface.* Your game might have an intelligent inventory system, which stacks inventory items in such a way as to maximize space, or put crucial items into more accessible locations. Or, you might use mouse-based *gestures* (stylized movements that can be assigned a function; like side to side swipes, L shapes, or circles) to perform commands within the game. Both of these systems could be easily accomplished with a very simple offline trained neural net. Another big UI usage is "advisors" in a civ-style game (where you'd have a specific AI analysis, most likely the same one that an AI-controlled opponent would use, to look over the player's game, and give personalized advice as to what options the player has within different areas of gameplay, like research or diplomacy).
- *Tuning game parameters.* Any time you find that an AI system has lots of parameters within a specific system, ask your self these questions: Can I create a side

program that can replicate this system atomically, or at the very least run this system within my game over and over? Can I quantify the potential "goodness" of a set of parameters, within some normalized scale? Are there relations between my parameters that I haven't been able to find with just trial and error tuning? If you answered yes to these questions, then your system is a good candidate to try to use an AI method to tune the system for you. GAs are particularly good at tuning parameters, especially if the states of these variables can easily be translated to some kind of genetic representation. Because of the number of parameters that many games use to tune their gameplay settings, you might have to split up your game into special "states" (which may not be really any different other than the parameters are more related within these states, so you end up with a *number* of GA tuned sets of parameters that you use based on which state your game is in), or you may have to try and GA tune small portions of your parameter set, and hand tune other parts. Games have used these methods for tuning parameters on physics simulations, or tuning the parameters on transitions within a state machine running deathmatch bots.

- *Automated testing.* As games increase in complexity, it becomes harder and harder to test every single possible combination of factors within the software to find potential crash bugs. One way of doing this is to use something that other software companies have used for years, an automated testing system. By constructing your software up front with this system in mind, you can have parts of your game be bug tested by an autonomous tool, giving you more time to spend on gameplay tuning instead of bug identification and replication. Basically, the key to allowing for automated testing is to have a control system that is very generic, open, and spoofable (which means that you could have another program *spoof* this input without actually needing a person to create the input). Games are generally good candidates, because the interface to the game is the generic controller, keyboard, joystick, and mouse that the player would use, and it's usually an easy thing to generate this input, and feed it into the system instead of using actual input. There are several types of testing along this line. Limits testing, where you specifically use directed inputs that are around the limits of the system's capabilities, in combinations that might lead to bugs. Random testing would use completely random input to the system. Smart testing would actually be a system that tried to employ real game playing techniques to play the game, but might possibly then switch to one of the other methods at key points. So, a smart system would play competently in a racing game; but when it finds itself surrounded by other cars on a bridge, might start sending random input to test the robustness of the physics in handling cars on the bridge under collisions and different levels of control. Another factor in automated testing is the question of testing products out-of-the-box vs. in development. Testing during development allows for specific testing scenarios

TeamLRN

to be run on smaller portions of the code, and also fixes bugs before other systems are implemented that might be affected by bad code behavior. All of these testing systems might use different AI systems to be implemented. You might use anything from GAs to random test a particular section of a game, or actually use the AI-controlled opponent of a game to test out the system (if the AI in your game has been executed by having it output controller data to interface with the game), and have it specifically test limits or sketchy moments within the game.

General AI Design Thinking

When you're designing an AI engine, you're standing on the edge of a large sea of possibility. You're also at the near top of the development totem pole, as far as dependency. AI requires hooks into almost every other system of the game to make rational decisions, and make them quick. When dealing with the sometimes daunting array of functionality that an AI system will require, you should take into account a few ideas:

- During design, brainstorm like there's no tomorrow. Once your problem set has been laid out (meaning, you know what it is for the most part that the game is going to require from your AI engine), spend some quality time going over *as many ways as you can think of* for solving each of those problems. Don't think *about* them, however. Not yet. Just think them up, and write them down. Brainstorming is about keeping as much electricity flowing through as many brain pathways as you can, for as long as you can. Coming up with outlandish ideas is not pointless, or a waste of time. Because some times, stupid ideas are just the seeds of really great ideas.
- Follow up your brainstorming sessions by having serious talks with the rest of the AI staff about each idea in your list. Again, don't throw away "stupid" ideas just yet. Put things on the table, and cut them apart as a team. Dissect them, and find out if they're stupid to the core, or if there's a golden egg buried in there. Getting additional brains involved in a large undertaking like AI engine design can help uncover ideas that were hazy or even completely blank in even the best plan, and merely talking about the issues will get your brain in a state where it will be working things out in much more tangible ways, rather than the "I'm pretty sure how to do that" mentality that sometimes leads to giant holes in a design.
- If there's time in the schedule, try quick prototyping small-scale AI problems in a laboratory example, like the AIsteroids test bed that was used in this book. Advanced techniques can be worked up in a matter of hours or days, and given real world testing without having to spend weeks only to find out that it's not

suitable to your game. Moreover, implementation usually uncovers things you didn't think of in the design chair. Don't feel like you failed as a designer. There's *always* going to be too many variables to see everything. Don't try to predict the future. As soon as you have 80% of your solution, dive into the code, and *discover* the other 20% in a month of prototyping. Contrast this with spending months more in design, scratching and clawing to try and predict more of the "possible shortcomings and pitfalls," finding another 10% of your solution, having no code to show for it, and then *still* finding design holes when you start coding.

- Finally, just be open. Don't take other's ideas as attacks on your own ideas. Use a lesson from Fuzzy Logic when dealing with other people, especially programmers. If you are right, it doesn't mean that everyone else is wrong, and vice versa. You being 50% right, and the other guy being 50% right, still equals 100% right. Allow for fuzzy states of rationality in your dealings with others, and instead of arguing over semantics to prove that you're right, you'll instead be incorporating the factors of both ideas that are correct *together* into a better solution.

ENTERTAINMENT CONSIDERATIONS

Unlike some other software industries, our programs have two goals: to perform their stated functionality, and to give the player an entertaining experience. These two goals, while not exclusive, are rarely friendly, and how you create your game is almost always a careful balance between good programming and programming for the sake of goodness.

- *The Fun Factor.* Points to consider when tap dancing around the one thing that we are *really* in the business for: to make fun games.
- *Perceived randomness.* The question of randomness, and it's perceived inequities on gameplay.
- *Difficulty concerns.* The level of difficulty that you create within your game is definitely an entertainment issue, and there are design and AI design considerations.
- *Some things that make an AI system look stupid.*

The All Important Fun Factor

When you ask somebody about a new game, what do you really want to know? As an AI programmer, you might want to know how intelligent the AI systems seem, or maybe you're a jaded purist, and really just want to know how the gameplay in *this* game is deviating from the norm for the genre.

TeamLRN

But for the typical Joe User, you want to know one thing: is it fun? Yeah, sure, it's pretty, because there's hyperspatial megatextures on the collinear mapped pseudomonkeys. But once again, is it fun?

What really makes a game fun? Psychology has several theories. One is that simple tasks, which combine quick visual identification skills with motor reflexes, awaken old, hunter-gatherer instincts within our brains. We are built by nature to discern movement, far more so than color or shape. Video games might provide these deeply seeded centers of the brain with the kind of stimulus that they haven't had since we left the African plains and started hiding in caves in France. Another theory lies in classical conditioning, which states that any sufficiently repeated task that is also given periodic positive reinforcement will cause physiological changes within our brains that will make us want to do it *more*. Still another talks about the fact that most people derive their deepest sense of "pleasure" when engaging in a task that they have a high degree of skill in, and are also using to just shy of their limitations.

So, a fun game is one that gives the player rewards, but not too many, because it also needs to be challenging. How can we model our AI systems to best aid us in this endeavor? We need to maintain reaction speed of the system, to provide the right amount of drama and reactivity. We also need to make sure that the AI isn't too hard (because humans will give up easily if they feel like there's no chance of victory, to try and save face; "I didn't want to win anyways..."), but we must also strive to not be too easy, because the fun metric is for a game to be on the edge of your abilities. But, the "edge of your abilities" is different for every player, now isn't it. This is one of the driving forces behind *adaptive* AI difficulty determination. The system is supposed to monitor gameplay, and adjust the level of AI opponent difficulty based on the actions and performance of the human.

While this adaptive element is the Holy Grail of difficulty level problems, there are major hurdles to success with this method that have been discussed elsewhere in this book (most notable is purposefully poor human performance, to lessen the difficulty of the game).

Malicious exploitation of adaptive systems may one day be overcome, however, then we shall be able to deal with the question of difficulty level. One solution might be to model the player over time as to skill, and try to discern false negatives. Tuned correctly, the player should not be able to find a pattern of oscillating good and bad behavior that result in an overall *massive* advantage over the AI opponent. Again, the goal is for a slight advantage, to keep the player "At 40% health," meaning just on the verge of being in trouble, but still firmly in the game.

The other element of fun is novelty. Novelty allows us to try, and enjoy, experiences that might not be fun otherwise. We're willing to put up with the atrocious difficulty level of a game like *Defender* because it was new and unique. If somebody put out roughly the same game nowadays, it wouldn't do well, mostly because the

novelty is gone. Now all that's left is an unbalanced game with a difficult control scheme, nonexistent AI and grainy graphics. Many people heralded the AI enemies in *Medal of Honor* to be truly special, in many cases for one simple reason: that they would *pick up grenades you threw at them, and throw them back.* A simple addition to the AI scripts for an FTPS enemy, to be sure. But nobody had thought to put that element into a game before, and the *novelty* was instantly rewarded with praise.

Now, the AI exhibited by the entities in your game do not represent the sum total of either the difficulty level or the novelty within a game. Many other elements, including gameplay mechanics, control scheme, amount of powerups, time limits, etc. have plenty to do with this. So, there is room for all kinds of experimentation within the game AI world. But, if all else fails, the AI must be able to bend to the needs of the great fun meter. Because if it isn't fun, then you have essentially failed, no matter how smart it is.

Perceived Randomness

Almost all games have an element of randomness inherent in their gameplay and AI. The reason is called *replayability,* which is the degree to which a user wants to play the game some more after he's either solved it, or has played it for a decently long period of time and has gained a level of proficiency. Two types of games have so far proved to be the most replayable: games with solid gameplay and multiplayer support (like *Quake,* or chess), and games with solid gameplay and *balanced* randomness (like *Tetris,* or poker). Note that in both cases, your success also depends on having a good, solid, game experience.

Balanced randomness means that an element of gameplay is random, but it doesn't grossly affect the game's outcome. Pretty much no matter what order the pieces fall in *Tetris,* if you can keep a level head and a good system, you can survive very far in the game. Even in poker, where the luck really is in the draw, a good player can turn bad card luck into a win. Unbalanced randomness *feels* random. It makes the player feel like he's no longer in control of the game, and that at any second a string of dice rolls can undo any effort he might have achieved. A game that unknowingly uses too much unbalanced randomness in its primary gameplay systems or AI systems is surely doomed.

The way in which unbalanced randomness is introduced into games is deceptively simple: it is to use the normal `randNorm()` call (which returns a random floating point number between 0.0f and 1.0f). When you let *actual* randomness dictate the behavior of your AI enemies, you get unbalanced randomness. The reason that this is wrong is simple. Human beings do not intuitively accept statistical random chance. If you ask someone the question "I've flipped this coin 30 times, and it's been heads *every time!* What do you think the next flip will be?" they will almost invariably say "Tails! It's due!" even though there isn't any more chance of it being

TeamLRN

tails than the last 30. Probability is just about the most alien idea possible to the normal human brain; this is why the lottery people make so much money. If the average person realized they were actually about 10 times more likely to get hit by lightning 600 times than they are to win a typical state run lottery, they might just ease up on the $50 they spend every Friday.

So, how do we allow for balanced randomness in our games? The answer is simple. Don't be random. Say you're coding an AI decision function that is only to respond true 70% of the time. If we were to use the expression "`randNorm() < 0.7f`," we've statistically solved the problem. Over the lifetime of this function, it will return true 70% of the time. But in the short run, say, a single game, it might actually *always* return false. Yes, the chances are small. But it could happen, statistically. What does this mean for our game? Unbalanced randomness is what it means. Instead, we need to create a series of outputs that are balanced, to assure a closer approximation of what we consider a random series of results. For this same function, a more balanced way would be to have it generate a string of numbers at game start, the length of which is approximately equal to the average number of times it gets called in a game. So, let's say that our little function gets called an average of 20 times per game. To create a balanced series of outputs, it would generate 20 Booleans, 14 of which were true, and then apply a balancing function to spread out the negative and positive responses as evenly as possible, pushing the final array onto a stack for quick usage. Then, each time you called the function, it would return the next value popped from the stack. Now, know this: you're still going to get some variation in statistics. It might be that this run through the game, the little function only gets called 18 times, or maybe 23 times (if you do go long, just generate another 10 number sequence, or however many you feel is appropriate). You might even want to add in a little variance into the initial population creation (so that the function will actually return a random range of 65–75%, say). But what this buys you is a series of "random" results, where you don't ever get too many failures in a row, and you basically ensure that the final statistics will always be fairly close to what you originally wanted. No reliance on "actual" randomness, and the result is that your players never feel cheated.

Some Things That Make an AI System Look Stupid

- *The standard enemy with the machine gun "rules."* The rules are: miss the first shot, try not to shoot first, use tracer bullets to give away your position, and use a large cone of aim so that you miss a lot. First of all, these are good rules, if used correctly. But if taken too literally, or abused, the opponent will look stupid, indeed. Sure, an AI enemy should miss quite frequently, but don't spray bullets like a fire hose. You can actually find targets that are decently close to the player, and still not hit him (amazing, isn't it?), which might even make the encounter more exciting for the player.

- *Bad pathfinding.* No one thing has contributed to the utterance of the phrase "Stupid Computer" then this problem. Bad pathnodes, characters without adequate dynamic obstacle avoidance, multiple friendly units piling up on a narrow bridge, a speedy unit orbiting around a spot because he hasn't gotten close enough to be "there" yet, a surrounded unit twitching wildly as it tries to move, and a fast unit repeatedly running into the back of a slow unit are all comically common examples of bad navigation gaffs. Other issues under this heading might be RTS peons that build a building from an angle that will trap them once built, units finding an alternate route that takes them right past the enemy's massive laser cannon array, units in formation that switch positions every time you click for them to move, and supersized creatures that can't fit through a doorway standing on the other side staring at you as you unload ammunition into them, without fear (instead of just running down the hall, out the large front door, around the side, and then punting the player into outer space).
- *Enemies that know you're standing around the corner with a chain gun, but walk calmly around the corner anyway.* Kamikaze tactics and zombies aside, intelligent enemies don't do this. You might even have intelligent behavior that ducks behind cover. Then you walk back out? No, dive out, toward another piece of cover, and clumsily throw a grenade in the player's direction. Nothing too lethal, but it sure beats "Jeez, nobody moved or shot for 5 seconds, coast must be clear."
- *Enemies that don't notice a pile of 35 dead bodies in one spot and think "Sniper?" or "Is there a Tower of Death nearby?" but instead walk right over their friends to perform a patrol.* A *small* amount of environmental consciousness can go a long way with enemies. Some of the stealth games make it a part of the game to hide bodies to not alert other guards, but there's definitely an "all or nothing" consensus on this issue in games.

PRODUCTION CONCERNS

There are also general concerns stemming from the actual production of the AI system itself. Games are being made bigger and more complex by larger teams on advanced hardware.

- *Coherent AI behavior.* Much like a lead artist needs to consider the "look" of the entire game when considering the quality and feel of each artists contribution to the product, so too do separate AI programmers working on a single product need to think about the overall feel of the game AI.
- *Thinking about tuning ahead of time.* Creating your AI systems with tuning in mind from the very beginning will help the process to happily chug along from start to finish.

TeamLRN

- *Idiot proof your AI.* By assuming that unknown things are going to happen to your AI entities, and allowing them ways out, you can help make your AI characters much smarter looking.
- *Consider designer used tools differently.* Designers are (generally) not programmers, and we shouldn't treat them as such. AI tools that will be given to designers to use have issues that need to be considered before implementation and release.

Coherent AI Behavior

Everyone has played a game where levels 1–3 were fun, well paced, and gradually ramped in difficulty. Then they got to levels 4 and 5, to find radically different difficulty leveling, super long levels that have only three spots of action spread thin through the levels, and were crippled by a frustrating gameplay mechanic that almost stopped you from playing the game. Most likely, this is because the game developer actually had multiple AI people working on the game, and these multiple people didn't really talk to each other, collaborating on technique or gameplay feel. While this problem has decreased dramatically in recent years due to the increasing importance of AI in our games, it still rears its ugly head from time to time. AI tasks are split among the available talent, and away they go to their separate rooms, coding away.

One way to fight this is to have the designers construct the game equivalent of a business "Mission Statement." For any given game, they should have a fairly clear, simple description of what they're shooting for with the gameplay and AI systems for the game. A sports title might be "To provide a fun, fast basketball game that uses statistics to simulate signature moves, shooting ability, and play calling, but provides a quick, arcade-style movement system with over the top special moves and quick defensive opportunities." A fighting game might have the mission of "To create a karate simulation where the player will be able to quickly set off complex combinations of moves and counter moves and not have to worry about lining up his attacks." Now, if you have different people implementing the different parts of your AI, they're still going to know what kind of game you want, and are going to be able to implement it with the designer's vision in mind. They're not going to have a strange notion as to which parts of the game you want to be a certain way, and which to code another.

Thinking about Tuning Ahead of Time

Tuning game AI is quite possibly the *most* important part of the process. Take notice of Blizzard's games: *Warcraft, Starcraft,* and their current online foray, *Worlds of Warcraft*. Almost all of their games enter a beta testing phase that generally ends up being almost as long as other companies spend on development in total. They routinely continue to polish games that *already* have achieved higher than current

standards of gameplay, and will even continue to address gameplay balance issues after the game has been released. Why? Because they desire to put out the best product they can, partially because they know their fanbase demands it, but partially because they have extreme pride in their creation. Yes, they spend a lot of money to do this, but they also sell millions upon millions of games because they do. Other companies, like Square and Nintendo, follow this same formula. Tune, tune, and tune some more, until there's no more tuning to be done.

Facilitating this level of polish requires an upfront commitment to AI design that allows for quick turnaround of tweaking and balance issues. Data-driven AI is a huge step in the right direction for allowing the tuning process to be streamlined. Getting programmers out of the way of massive parameter tweaks, as well as other data-driven issues like enemy placement with a level, and specific enemy behavior in response to player location or condition will go a long way in speeding up the process of tuning a content heavy game. Plus, when tuning the game is fast and easy, the designers are more likely to do much more of it, and as such the process is self reinforcing.

Another tip is to not put magic numbers in your AI decision-making systems. If your enemy has a line in one of his states that reads `if(m_nearestEnemy < 45)`, maybe you should change that 45 to some kind of variable, and expose that variable to whatever your game is using as a tuning system. Chapter 25, "Debugging," details a Widget system that allows you to expose any game variable to a bank of tuning controls, where they can be adjusted in game. This kind of system is almost imperative for AI systems that rely on heavy game testing and tuning to get balance and gameplay feel right.

Idiot Proof your AI

Always provide idiot proofing. If you think there's any way that an AI behavior can screw up, it probably will. When there is a doubt, there is no doubt. Software systems have a smirking way of finding the one open door you've accidentally left for them. Not to mention that dropping three human players into your game, each with their own notion of how to exploit the system, act as nothing short of a high-powered catalyst to AI decomposition. Prepare for this, by providing backdoors out of behaviors. Timers for behaviors that have gone on too long are the easiest to code, but simple exit conditions can really help stop an AI behavior that's making itself looks stupid. Just get into the habit of giving your AI a way out. Idiot proofing extends to designer provided data, as well. A chunk of code that will save you tons of time in development is an ironclad "checker" that either runs on the data from a command line, or at game load time (and can be removed from the project before release), and provides you with complete scanning of incoming AI data for inconsistencies, outright errors, overly complex or cyclical state diagrams, broken

TeamLRN

pathnode networks, missing elements, doubled elements, and the like. Yes, this is throwaway code. But it's better than the throwaway weeks you'll be spending debugging the pathfinder only to discover that a designer screwed with a file in an unexpected way, or that your version control corrupted a single byte in a script file that nobody has touched in four months, and it's causing odd behavior in a small wall switch on level 8 that miraculously, no tester will check during quality control until 12 minutes before you go gold.

Consider Designer Used Tools Differently

AI tools that will be used by designers need a special touch. If you're going to be exposing game logic to the designers, do so with some semblance of kid gloves. Don't put every bit of functionality *you'd* want in an editor in, put just enough to get the job done while remaining straightforward, and simple. Going to be building logical expressions? Consider only allowing ANDs, and not ORs, XORs, or anything else. Logic gymnastics aren't the strong point of many programmers, much less people who may have started out in the industry testing games. This is not to be insulting to designers, they have one of the hardest jobs in the industry (Great job on Game of the Year. Now get back to work on the *better* version for next year!), and everybody thinks they're a designer. Sort of like everybody thinks they can sing. Another don't includes command line tools with lots of parameters to set (if possible encapsulate this kind of thing as an Export button from within the editor, or at least make a batch file or six that they can run to do what you need of them). Provide lots of well-documented, functional examples with any tool or scripting system. Finally, be open to feedback from the designers on GUI issues and functionality problems or irritations.

SUMMARY

A modern day game AI engine is a hugely complex software system, and there are many common things to consider when coding up one. This chapter looked at specific concerns dealing with design issues, entertainment issues, and production issues.

- Some of the concerns to think about during the design phase of the AI engine include data-driven problems, one-track mind syndrome, level of detail AI, support AI, and other general AI design ideas.
- Entertainment concerns include the fun factor, perceived randomness, difficulty settings, and general things that make AI systems seem stupid.
- Production concerns involve coherent AI feel, tuning the game, idiot proofing your AI, and treating tools used by the designers differently.

25 Debugging

In this section, we'll discuss one more part of real AI development, debugging your game, from the start to the finish. We shall cover common debugging issues, and cover some ways in which to write your code in the first place to plan ahead for bugs. Included in this chapter will also be a Windows MFC implementation of a very useful runtime debugging and tuning tool called *widgets*.

GENERAL DEBUGGING OF AI SYSTEMS

Because of the nature of AI engines, debugging them can be a cumbersome affair. AI invariably touches a number of game systems, bridging the gap between control, physics, sound, gameplay mechanics, and input/output systems. So, many times bugs that appear to be AI based end up being deep within one of the support system's code, but doesn't come out until the AI system really starts taxing a particular chunk of game code. Get used to having to not only show other people that they need to fix something in their part of the code, but be ready to back it up by having either a test case set up that can replicate the problem, or have them come over to your work space and step them through directly. You'll save a lot of time and energy that will be wasted by sending off an e-mail saying "Fix your code," and then sitting and wondering when it's going to happen.

One benefit of using the Distributed AI design from Chapter 23, "Distributed AI Design," is that it allows for setting breakpoints at multiple levels, and stratifies the functionality of each subsystem to the point of being easier to identify where in the code a problem might be located. So, you debug specific systems, instead of having to trace through large combination systems or convoluted classes.

VISUAL DEBUGGING

Visual debugging means using viewable information from within the running game to show you information about what the system is doing to debug your programs.

TeamLRN

Having your AI characters display information while the game is running, including text about their current state, or lines showing intent, direction of travel, and thought processing, even visually watching influence map data change and move with the game to see problems. Game AI, more so than most systems, profit greatly from this kind of debugging information. The benefits include:

Provides a Variety of Information

Visual debugging includes writing text to the game screen, as well as other visual aids. You might want to draw lines pointing toward the targets that each AI character is interested in, or even highlight pathfinding traces, to find bugs in your navigation system. A visual representation of the influence map data is especially useful for debugging the game (in fact, you might want to try turning off drawing of the regular game characters, to just watch the influence data for anomalies), as is any abstract data organization method that can help you see a more simplified view of the game.

Helps with Both Debugging and Tuning

At each stage of your game, you should give yourself a good visual representation of what is going on, so that you can *assure* yourself that what you think is happening really is. If you're coding up a specific perception that deals with line of sight, put a visual system in the game, so that you can either see all the traces the AI is doing to determine line of sight, or have it "signal" in some way, to let you know exactly when it starts, and when it stops having line of sight. Then get in the game and actually stare down the barrel of the system, making sure that it's doing what you want, but also that you're getting the *feel* of the system that you're looking for. This is especially important with secondary characteristics, like reaction time. Put indicators in the game, and watch them happen a few hundred times. This kind of behavior will help you find strange math feedback bugs that create holes in behavior and perception, but it also helps you to tune systems for proper gameplay feel *much* quicker and more easily.

Timing Information

Frequently, it is hard to get events to happen within a debugger that only occur on one game loop, especially if your main game update loop is time based (rather than frame based) and you can't set it to use a constant delta during debugging. If your game timer is using the system clock of the processor that you're debugging in, stopping the code with a breakpoint, and then stepping through some code will give you a huge time delta because time continues passing when you're debugging. If you set your game to use a constant delta time for debug purposes, this problem

can be minimized. If this is not possible, you can use visual debugging information, to put up on-screen information while the game is running at full speed to try and determine the problem. Note that if you write too much text, you might slow down the game because of that, and again have problems getting your bug to repeat.

Watching for State Oscillation

When using a state based decision structure, you can watch for odd state switching by allowing your AI system to display state information visually in game. Have the game display this information directly on the character, or over his head, so that you can easily correlate the data with the character. Other useful state information might include hierarchy status, useful statistics like the time the character has been in the state, and transitional information.

Useful for Console Debugging

On consoles, you typically are developing in a Windows or Linux environment, and then upload the executable in some way to a test console, while running a debugger on the PC. Because of the remote debugging issue, many common debugging tricks can't be done, and so drawing text or graphics on the console's screen becomes a big source of debugging aid.

Debugging Scripting Languages

Unless you've taken the time to completely write up a debugging system for your scripting language (and not many game schedules allow for this), you might be left high and dry with only in-game tactics in which to find bugs. One trick is to give your scripters specific debugging commands to put into their scripts; you can then set up on screen text from within an AI behavior script as well.

Double Duty Influence Mapping

If your game has an influence map system, you can use it as a visual debugging tool as well. By either adding more space per influence cell (if you have room), or even taking over the system completely (if it's not something the part of your game you're debugging requires to run) to display additional, debug specific information is a great and easy usage of the technology. You could display terrain analysis happening on the fly, or avoidance code working on the various AI units in your game. Anything you can link to a specific location, that can be displayed by setting values in the map, can be shown visually by just allowing the system to display the contents of the IM while the game is running.

TeamLRN

WIDGETS

Sometimes (in fact most times), when coding up specific behaviors or perception systems, you come across in-game values that you wish you could not only see, but also change or tune while the game is running. Widgets are an implementation of this concept that you can easily add to your Windows games, or port over to non-MFC using applications and use wherever.

Basically, widgets allow you to put a "control knob" on many types of variables within your game. While the game is running, a small window will appear, called a Widget Bank, that stores all the widgets you've created. Opening the bank allows you to change the values of the variables you've linked to each widget in real time, while the game runs.

Implementation

The code is pretty simple, with only a few basic rules and things to do to get up and running. The entire system was written by Max Loeb, who, incidentally, also helped with most of the diagrams in the book. The basic system is as follows:

- `WidgetBank` is the highest level of the widget hierarchy. It is the "window" in which all the widget groups reside.
- `WidgetGroup` is the second level of organization. Each widget must be a part of a group; you cannot put widgets directly into a Bank. Groups can include sub-groups.
- The `Widget` class itself. A widget can be one of a few types: a `basic button` (for launching an event function of some kind), a `radio button` (for choosing between two labeled settings), an `OnOff button` (a special button for toggling Boolean values), a `Scrubber` (which allows you to *scroll* through values of a continuous variable), and a `Watcher`.
- The `EventHandler` class, which will allow us to use callbacklike functionality within our widgets.

The `Widget` class is an empty base class that has two functions in it: `Update` and `Draw`. It merely provides a way for the other widgets to have a common parenting, so that the widget bank and other classes can use any and all widgets.

The `WidgetBank` class (Listing 25.1 shows the header) is the main window for the system. It is the class that mostly takes care of the MFC functionality for the windows.

LISTING 25.1 WidgetBank Header File

```
/*****************************************************************************
***
*  WidgetBank: This is the window that houses all the widget windows,
ie.
*                camera widgets, bone widgets, light widgets, and so
on.
*
*****************************************************************************
*/
class WidgetBank : public CWnd
{
public:
    // constructors
    WidgetBank();
    virtual ~WidgetBank();

    // member methods
    BOOL Init();
    void RedrawWidgets();
    void UpdateWidgetBankSize();
    void Update();

    // widget creation methods
    int GetHeight();
    Group* AddGroup( char * label );
    afx_msg UINT OnNcHitTest(CPoint point);
    afx_msg void OnSize(UINT nType, int cx, int cy);
    afx_msg BOOL OnEraseBkgnd(CDC* pDC);

    // member variables
    Group * myWidgets[MAX_NUM_WIDGETS];

    int     m_numWidgets;
    int     m_totalWidgetHeight;
    CRect   m_ClientSize ;

    DECLARE_DYNCREATE(WidgetBank)

    DECLARE_MESSAGE_MAP()
```

TeamLRN

```
private:
    int m_id;
public:
    afx_msg void OnNcDestroy();
    virtual BOOL CreateEx(DWORD dwExStyle, LPCTSTR lpszClassName,
LPCTSTR
                          lpszWindowName, DWORD dwStyle,const RECT&
rect,
                          CWnd* pParentWnd,UINT nID,LPVOID
lpParam=NULL);
   afx_msg void OnVScroll(UINT nSBCode,UINT nPos,CScrollBar*
pScrollBar);
};
```

A Widget `Group` is an organizational method of setting up your widgets to be displayed hierarchically. Upon startup, all the groups will be minimized, and you open a group by clicking on its label, which will *open* the bank for viewing of the individual widgets inside. By using groups, only those widgets you want to see at any one time have to have their groups open, which can help a lot if you've put widgets on a class that has a lot of instantiations in your game. The group class is a bit more involved because this is where the brunt of the widget organization functionality resides. Listing 25.2 shows the header, as you can see, most of the important functions deal with adding the various types of widgets to the bank, drawing them, and updating any window movement, resizing, and so on.

LISTING 25.2 Group Header File

```
/*************************************************************************
*   Group:      A Group represents an entry in the widget bank, and
*               can house other groups or widgets. It contains a
*               header which can be expanded/contracted to show/hide
*               its child groups or child widgets, which are added with
*               subsequent calls to AddScrubber, AddOnOff, etc.
*
*************************************************************************/
class Group : public CWnd {
public:
    // constructors
    Group( char * label, CWnd * pWin, int pos, int height, int width,
           const int level );
    virtual ~Group();
```

```
// member methods
virtual void Update();
void OnClickHeader();

bool IsExpanded(){ return m_status; }
int  GetHeight();
int  GetClientHeight();
int  GetPrevPos(){ return m_prevPos; }
void SetPrevPos( int prevPos ){ m_prevPos = prevPos; }
void SetWidgetBank( WidgetBank * wb ){ m_widgetBank = wb; }

Group * AddGroupWidget( char * label );

ScrubberWidget<int> * AddScrubber( char * name, int & val );
ScrubberWidget<float> * AddScrubber( char * name, float & val );
ScrubberWidget<unsigned char> * AddScrubber( char * name, unsigned
                                             char & val );

OnOffButton * AddOnOff( char * name, bool & a, int ID1 = 0,
                        EventHandler * h = 0 );

void AddWatcher( char * caption, float & val );
void AddWatcher( char * caption, int & val );

void AddText( char * caption );
RadioButton * AddRadio( char * groupName, char * leftName, char *
                        rightName, int & val, int id1, int id2,
                        EventHandler * h = 0 );
BasicButton * AddBasicButton( char * filename, int id,
                              EventHandler * h = NULL );

    int Draw( int y_pos );

    // MFC Overrides
    DECLARE_MESSAGE_MAP()
    afx_msg void OnNcDestroy();
    afx_msg BOOL OnEraseBkgnd(CDC* pDC);

    // member variables
private:
    Group * m_childGroups[ MAX_CHILD_GROUPS ];
    Widget * m_childWidgets[ MAX_CHILD_WIDGETS ];
    WidgetBank * m_widgetBank;      // a pointer to the parent
widgetbank
```

TeamLRN

```
    CButton m_header;
    COLORREF m_color;                   // the color of this widget
    int m_top;                          // position of the top of this
widget
                                        // in widget bank
    int m_prevPos;                      // used for positioning groups when
                                        // drawing
    bool m_status;                      // is this widget currently
expanded?
    unsigned int m_numChildGroups;      // number of subgroups contained in
                                        // this group
    unsigned int m_numChildWidgets;     // number of child widgets in this
                                        // group
    int m_level;                        // how many levels deep is this
nested?
};
```

The `EventHandler` is a basic callback class, with a purely virtual function called `UIEvent()`. To use an event handler with a widget, you make a child class of `EventHandler` for the class that you need to use a callback from, instantiate a copy in your class, and then override the `UIEvent()` function to be your callback. Make sure you include a parent pointer back from your `EventHandler` child class, so the callback can has the access it needs. When you set up a widget button, you give it a button ID. When the button is pressed, it will call the `UIEvent()` function and pass in the button ID. Your overridden event function can then use the button ID to determine what it wants to do.

Moving right along, we come to the actual types of widgets themselves. Each one will be discussed in turn, and then a small sample file will be shown that that implements each type within a program.

BasicButton

The simplest of widgets, this allows you to put a label on a button, and use it to set off a callback event. Listing 25.3 shows its header. As you can see, it's really just a wrapper for an MFC `CButton` that a widget can access.

LISTING 25.3 `BasicButton` Header File

```
class BasicButton : public Widget
{
public:
    BasicButton( EventHandler * eventHandler = 0 );
    ~BasicButton(void);
```

```
    void Create( char * label, int id, CWnd* pWin, int pos );
    void Draw();
    EventHandler * m_eventHandler;

    DECLARE_MESSAGE_MAP()
protected:
    virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);

private:
    CButton m_button;
};
```

Watcher

A watcher is a templated widget (although right now it's only implemented for `int` and `float` variable types) that just shows you the value of a variable; you can't change it from the widget itself. Useful for constantly updating variables that it wouldn't make sense to try to change from the widget, because the game would immediately blow away your changes anyhow.

Listing 25.4 is the header file.

LISTING 25.4 `Watcher` header file

```
template <class T>
class Watcher: public Widget
{
public:
    Watcher( int & watch);
    Watcher( float & watch);
    ~Watcher(void);
    void Create( CString label, CRect r, CWnd* pWin );
    void Draw();
    void Update();

private:
    CStatic m_label;
    CStatic m_watch;
    T & m_val;
    int m_frameCount;       // frame counter
    int m_updateInterval;   // update every this many frames
};
```

TeamLRN

RadioButton

Radio buttons are standard windows controls, it allows you to choose exclusively between items. The current implementation only supports two choices, but it could be easily extended to an arbitrary number of choices. The header is in Listing 25.5.

LISTING 25.5 RadioButton Header File

```
class RadioButton : public Widget
{
public:
    // constructors
    RadioButton( char * groupName, char * leftName,
        char * rightName, int & val,
        CWnd * pWin, int yPos,
        int id1, int id2,
        EventHandler * h );
    ~RadioButton(void);
    void Draw();

protected:
    virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);

private:
    // member variables
    CButton m_GroupButton;
    CButton m_LeftButton;
    CButton m_RightButton;

    int & m_val;

    EventHandler * m_eventHandler;
};
```

OnOffButton

This widget is a special kind of button that toggles a Boolean value. It's drawn using the "check box" type of Windows control, or the standard push button, dependent on which style you set it as. Listing 25.6 shows its header information.

LISTING 25.6 OnOffButton Header File

```
class OnOffButton : public Widget {
public:
```

```
    // constructors
    OnOffButton( bool & state, EventHandler * eventHandler = 0 );

    // member methods
    void SetStyle( int style );
    void SetCheck( bool checked );
    int GetCheck();
    void Draw();
    CButton m_button;

    DECLARE_MESSAGE_MAP()
protected:
    virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);

private:
    // member variables
    bool & myState;

    EventHandler * m_eventHandler;
};
```

ScrubberWidget

A scrubber is one of the more useful widgets. It allows you to reference a `float`, `int`, or `char` type variable, the value of which is shown in the widget. However, if you click and hold the mouse cursor on the widget, you can drag the values left and right between minimum and maximum values that you set. You can also set the speed of the scrubbing (between slow, regular, and really slow). Listing 25.7 is the header.

LISTING 25.7 `ScrubberWidget` Header File

```
template <class T>
class ScrubberWidget : public Widget {
public:
    // constructors
    ScrubberWidget(T & var );
    ~ScrubberWidget();

    // member methods
    void Refresh();
    void Draw();
    void OnEditKillFocus();
```

TeamLRN

```
        void Create( char * label, CWnd* pWin, int pos, SCRUB_SPEED speed =
                     REGULAR_SPEED );
        void SetMin( T min ){ myHoverButton->m_minValue = min; }
        void SetMax( T max ){ myHoverButton->m_maxValue = max; }
        void SetMinMax( T min, T max ){myHoverButton->SetMinMax( min, max
    );}

        // Overrides
        afx_msg BOOL OnEraseBkgnd(CDC* pDC);

        // member variables
        CEdit myEdit;
        HoverButton@T: * myHoverButton;

        T &scrubVar;                    // the value being changed

        DECLARE_MESSAGE_MAP()
        afx_msg void OnNcDestroy();
    };
```

Integration within a Program

To use widgets in your program, follow these simple steps:

- First, include the `WidgetBank.h` file in any class that you want to put widgets onto.
- Then, add a function, called `AddWidgets()` to the class. If the class is to be a "main" class, which will spawn groups as well as widgets, then the function should take a `WidgetBank` pointer. If the class is a "secondary" class, which will instead only have values that you want to use widgets upon, then the function should take a `Group` pointer.
- Override the `AddWidgets()`call in your class to either add whatever bank, groups, or widgets you want, using the examples in Listing 25.8 as guidelines.
- Figure out how you want to update the `WidgetBank`; it has an `Update()`function that you should call every frame if you want completely updated widgets.

LISTING 25.8 Widget Use Guideline Example

```
// Our car's EventHandler class. Note that it must
// be created with a pointer to the car so that
// we can interact with it in our UIEvent's switch
// statement. Alternatively, we could also
// simply derive our car class directly from
```

```
// an EventHandler, and remove the need for
// a Car pointer.
class CarEventHandler : public EventHandler
{
public:
    CarEventHandler( Car * car ){ m_car = car; }
    virtual void UIEvent ( WPARAM id );
private:
    Car * m_car;
};

void CarEventHandler::UIEvent(WPARAM id )
{
    switch (id)
    {
    case Car::IGNITION_KEY:
        m_car->StartCar();
        break;
    case Car::WIPERS_CONTROL:
        m_car->StartWipers();
        break;
    case Car::AIR_COND:
        m_car->ToggleAirCond();
        break;
    }
}

class RacingGame
{
public:
    RacingGame(){};
    void AddWidgets( WidgetBank wb );

private:
    Car     m_car;
    Track   m_track;
};

class Car
{
public:
    enum {
        IGNITION_KEY,
        WIPERS_CONTROL,
```

TeamLRN

```
            AIR_COND
        };
        void AddWidgets( Group * g );
        void StartCar(){ m_engine.Start() }
        void StartWipers(){ m_wipersOnOff = true; }
        void ToggleAirCond( m_air ? m_air = FALSE : m_air = TRUE );

    private:
        Engine  m_engine;
        bool    m_lightsOnOff;
        bool    m_transmission;
        bool    m_air;
        bool    m_wipersOnOff;

        CarEventHandler m_eventHandler;
    };

    /*************************************************************************
    *   Name:   AddWidgets
    *
    *   Info:   A typical AddWidgets function for a fictitious racing
    *           game. Because the RacingGame object is high level, it
    *           will be adding Groups directly to the widget bank. Actual
    *           Widgets will be added to these groups by the AddWidget
    *           functions of lesser, individual components of the game.
    *
    *   Args:   wb - A pointer to the WidgetBank, which is the top-level
    *           parent of all Widgets and Groups. Again, you don't add
    *           widgets
    *           directly to the widget bank—you only add Groups.
    *
    *************************************************************************/
    void RacingGame::AddWidgets( WidgetBank * wb )
    {
        Group * g;

        // Add our first group to the widget bank. AddGroup() returns a
        // pointer to the group it created. You can either use this pointer
        // to add widgets now, or, preferably, pass it to the AddWidgets()
        // function of a lower-level contained class.
        g = wb->AddGroup("Car Properties");
```

```
// Now that we have our group, we'll pass it to the AddWidgets
// function of our car class object, which is a member of a
// RacingGame object.
m_car.AddWidgets( g );

// That takes care of the car's widgets, so let's add widgets for
// the race track. We'll make another group, and reassign our
// group pointer to it
g = wb->AddGroup("Track Properties");

// Again, we pass the newly assigned pointer to the AddWidget()
// function of a lower level class, this time a RaceTrack object.
m_track.AddWidgets( g );
}


/*****************************************************************
 *  Name:    AddWidgets
 *
 *  Info:    A typical AddWidgets function for a fictitious car class
 *           to demonstrate the use of widgets. This example only
 *           covers a Car object, but remember that you have to write
 *           an AddWidgets function for any class that you want
 *           to have widgets. From here you might write AddWidgets
 *           functions for your racetrack class, your environmental
 *           class, your AI classes, etc.
 *
 *  Args:    wb - A pointer to a Group. We use a Group pointer
 *           to add the actual Widgets to our application.
 *
 *****************************************************************/
void Car::AddWidgets( Group * g )
{
    // We'll need a pointer to a group. We'll call it pg, for "parent
    // group"—be careful not to confuse it with the group pointer
    // that is being passed into this function.
    Group * pg;

    // Our car class contains an engine object. Let's give it its own
    // widget group. Groups can contain other groups, which gives
    // you a lot of flexibility to organize your widgets.
    pg = AddGroup("Engine Properties");
```

TeamLRN

```
// Our engine class has its own AddWidgets function, so we'll
// pass it our new group pointer

m_engine.AddWidgets( pg );

// Our car class has some member variables that would
// be fun to control while the game runs. We'll hook up
// some widgets to them now, using the group pointer
// that was passed in

// let's start by adding a widget to control the on/off state
// of the car's headlights
g->AddOnOff( "Headlights", m_lightsOnOff );

// it would be nice to monitor the car's fuel gauge—we'll
// add a Watcher widget.
g->AddWatcher( "Fuel Level", m_fuel);

// We want to tune the car's mass as it drives around, so
// we'll attach a ScrubberWidget. We're going to catch the
// ScrubberWidget pointer that this function returns, so that
// we can change a setting
ScrubberWidget * sw;
sw = g->AddScrubber( "Mass", m_mass );

// We don't want negative or absurdly huge values for the mass of
// this car during scrubbing, so we'll set some limits on the value
// using the pointer we got back from the AddScrubber function
sw->SetMinMax( 0, 10000);

// Car objects can have automatic or manual transmissions. We'll
use
// a radio button, which allows you to have a caption for the
overall
// control, as well as each actual button.
g->AddRadio( "Transmission:", "Automatic", "Manual", m_transmission
);

// For our final widget, we'll add a button that starts the car's
// engine and other systems. Because we want to attach some
// functionality to this button(it won't do anything if we don't),
we
// pass in an EventHandler object that we've written for Car
Objects.
```

```
    // We also pass in an enum name for the button. This enum value
will
    // become the id number of the widget. When the button is actually
    // pressed by a user, its id number is passed to the EventHandler's
    // UIEvent function, and is used in a switch statement to call that
    // widget's particular code.
    g->AddBasicButton("Start Car", IGNITION_KEY, m_eventHandler );

    // We'll add a few more widgets that use the same EventHandler.
    g->AddBasicButton("Windshield Wipers", WIPERS_CONTROL,
                      m_eventHandler);
    g->AddOnOff("Air Conditioning", m_air, AIR_COND, m_eventHandler );

}
```

So, now go forth and populate your game with widgets. You'll quickly find that they'll speed up both debugging and tuning of your game. Some ways that you might extend the widget system, to get even more out of them:

- Wrap all your `AddWidgets()` functions with `#ifdef __DEBUG`, or whatever you are using for conditionally compiled code in your project, and then use the preprocessor to conditionally remove all of your widget stuff when you go to release the game.
- Use a `BasicButton` to save or write a text file containing all your widget values. You could serialize all your widget values within a file, and when the game starts back up, it could then initialize all of your variables with the values from the file. In this way, you wouldn't spend an hour tuning a value, and then have to write them all down on paper to adjust your initialization values in game. Before you release the game, however, you would have to transfer all the initial values out of the file (or actually use the file as a configuration script).

SUMMARY

Debugging AI systems can be quite a chore, because they interface with a majority of other game systems, can be filled with specific case code or data, and require complex setups to replicate bugs within. This chapter discussed many issues for AI developers to watch for when debugging game AI engines.

- General AI debugging problems might appear in other people's code, and many concerns can be alleviated by using the Distributed AI method.

TeamLRN

- Visual debugging provides a variety of information, helps also with game tuning, can provide timing information, can help watch for state oscillation, is especially useful for console development, is useful when debugging scripting languages, and can dovetail nicely into influence mapping systems already in use within a game.
- The Widgets library introduced in this chapter provide the user with a general platform for exposing variables to a simple user interface that allows monitoring as well as changing of variable's values while the game is running.

26 Conclusions, and the Future

We've covered a lot of ground in this book, and the hope is that enough of it stuck that you're already brimming with ideas that you are going to implement, using a little skeletal code from the CD-ROM, as well as a serious amount of your own hard work and creativity. We've covered everything from the simple to the very complex, both in theory and practice, and along the way discussed an entire paradigm for approaching AI engine design.

During engine design, split up your AI engine tasks into a distributed, layer-based system, using any of the applicable layers:

- Perception/Event layer
- Behavior layer
- Animation layer
- Motion layer
- Short-term decision making
- Long-term decision making
- Location-based information layer

Then, for each layer you intend on implementing, try to consider the eight considerations when deciding upon the type of decision-making techniques you plan on using:

- Types of solutions
- Agent reactivity
- System realism
- Genre
- Content-specific requirements
- Platform
- Development limitations
- Entertainment limitations

TeamLRN

WHAT GAME AI WILL BE IN THE FUTURE

The push for better and better AI opponents will continue. Although online play connects more and more humans to play each other, many people still play only single-person games, or do not go online to search for opponents. These people are still the majority of game players, and they demand increasingly complex and compelling game agents to play against.

AI has, and will continue to become increasingly important to the public opinion of any particular game. Game reviews spend most of their time on the pros and cons of the AI exhibited by the game. The last 10 years were almost completely the realm of game graphics, and we can now see the fruition of that effort: huge polygon counts, texturing and lighting that is approaching photographic levels, and overall movie quality visuals are almost the norm. An equivalent push is now coming into play for the AI systems in games. We will see increasingly complex and creative AI in games, from enemies that learn the human player's style and react accordingly (learning, and opponent modeling), AI opponents that come up with novel solutions to gameplay problems (inference, emergent behavior, or even creativity), even opponents with humanlike moods (emotion).

Another thing that games in the past have suffered from is lack of personality. There are very few differences between opponents, or the differences are purely from a statistical point of view (enemy A is slightly stronger than B, but B is faster). The reason being, of course, is that enemies of the past have been more hard-coded (written in a very specific and code-based way), for balancing concerns as well as coding time. In stark contrast would be the enemies in a game written mostly around a learning system, with very basic knowledge of the game world, and who would make decisions based more on the game situations that they have been involved with over their lifetimes. *Black & White* used a system somewhat like this (although the description is overly simplified), and almost no two main creatures turn out the same, even if played by the same person. The personality of the creatures is being determined by such a large number of factors that emergent behaviors and personality traits are inevitable. Doing this leads to a much more personable view of the game's creatures, and an overall more satisfying outlook on the intelligence of the system. This is analogous to an experience of playing a pen and paper role-playing game being much more personable and intelligent than the experience of reading a book about the same game world. Playing the game is interactive, and thus awakens instinctual perceptions within ourselves that give life to the characters and elements that we come across, simply because we are a part of the process. We can interface with the world, change the world, and become a part of the world. Reading the book is merely taking in a story, and although it can seem compelling and to some lesser degree real, it will never be able to answer all the questions we have, or give us a look from another angle. There is a distancing from

the material that is created by the author's mode of writing, as well as his overall storytelling ability. Which is why wholly scripted AI systems will not completely satisfy us, since these are again limited by the scripter, and while adding richness to the gaming experience, will never be equal to the experience of dealing with another intelligent *person.*

AI will also incur the changes to gaming in general, and need to make strides to accommodate them. In the short run, a number of these changes might be in the area of human interfaces. A large number of games are beginning to incorporate voice commands from a headset or microphone. Games may one day require full speech recognition as well as translation-type abilities. Also, many games are becoming online, persistent world endeavors. The AI associated with enemies or NPCs in these games might have opportunities for long-view learning and personality building, simply because the game doesn't ever stop.

In the far-flung future of our games, we may one day have full featured intelligent systems that competently play our games with us, at the difficulty level chosen specifically for each player, with style, creativity, personality, and a degree of *humanity.* Oh what fun we'll have.

TeamLRN

Appendix About the CD-ROM

The CD-ROM included with this book contains all the source and demonstration programs referenced within the book, as well as some other useful materials. Also, you can refer to the main Charles River website (*www.charlesriver.com*) for updates and additional support information.

CONTENTS

SourceCode. All the source for the various topics is arranged in subdirectories by chapter of introduction. Each demonstration is compiled using Microsoft Visual C++ 6.0, and the compiled binaries can be found in the specific output directories.

Figures. All of the figures from the book are included in this directory, named the same as they are in the chapters.

Useful Web Bookmarks. Here are a few pages of links to various web resources, from general to very specific. The links are divided up into categories: ALife, fuzzy logic, general AI websites, genetic algorithms, location based information, neural nets, scripting, various AI links, Game source code, and various Game AI issues.

Libraries. Contains the currently newest available download of the two libraries used by the demonstration code: the GLUT wrapper for OpenGL, and the Lua language. Of course, you would want to check the internet for newer versions, but the demos in this book have only been tested with these versions.

TeamLRN

SYSTEM REQUIREMENTS

All the demonstration programs were tested on multiple machines, from a Pentium 4 3.2Ghz with a GeForce 3 graphics card, to a Pentium 3 1Ghz laptop with a GeForce Go card. Since GLUT works on pretty much all versions of Windows (ME, 2000, XP), the demos should compile and run on these setups. All the code was written and compiled using Microsoft Windows Visual C++ 6.0, and GLUT and OpenGL was installed on the machines.

Index

2002 Game Developers Conference, 90
2.5D games, 124
2D *Mario Bros.*-style games, 131
2D shooters, 145
3D games, 36, 42–43, 124
3D rendering and primitives, 542

A

Abuse, 131
academic AI, 9–11, 460
acting 'human,' game development and, 9–10
action potential, 454
activation function in neural nets, 459–460
activation levels in fuzzy systems, 281, 286
ad-lib machines, 82–83
AddMessageToSystem ( ) function, 315
AddWidgets ( ) function, 567, 575
adult content games, 179
adventure games
  areas that need improvement, 92–94
  common AI elements, 88–90
  introduction to, 87–88
  summary, 93–95
  useful AI techniques, 90–92
Age of Empires, 32, 98, 100, 103, 107, 108
agent reactivity
  and AI system design, 30–31
  in genetic algorithm solutions, 450
  in location-based information systems, 407
  in message-based systems, 334, 336
  in neural nets, 487–488
  of production systems, 503
agents
  cooperative, in FTPS games, 117–118
  reactivity of, 30–31, 276, 307, 368
AI (Artificial Intelligence), 3
  data-driven, in real-time strategy machines, 105
  engine, *see* AI engine
  game, *see* game AI
  helper, 109
  hierarchical, 104
  idiot proofing your, 556–558
  infusion of AI techniques in shooter games, 156
  micromanagement, 110
  system design and bounded optimality, 24–25
  track, *see* track AI
AI development
  common design concerns, 541–551
  general design thinking, 550–551
  production concerns, 555–558
  with AI elements, 108–109, 229
AI engine
  basic components and design, 11, 29
  decision making, inference, perception 30–40
  design steps, summary, 46–48
  distributed AI design, 515–516
  layers, 577
  navigation, pathfinding, 40–46
  techniques overview, 239
AI system hooks in games, 61–62
AI systems
  debugging, 559–561
  level of detail (LOD), 274
AIControl class, 61
AIsteroids test game, 49
  Control class, 60–62
  FuSM implementation (fig.), 301
  GameObj class, update function, 49–52
  GameSession class, 55–61
  implementing neural net within, 464–480
  implementing genetic algorithm system into, 428–442
  main game loop, 63
  message-based implementation, 319–327
  with occupance IM (fig.), 392, 396
  running GA implementation (fig.), 443
  scripting system implementation, 339–345
  ship object, 52–54
AIsteroids.cpp, 63
algorithms
  genetic, *see* genetic algorithms
  monothetic, 505
  planning, 496–500
  planning, in civilization games, 213
  and production systems, 502
  and scripting, 367
alleles, 414
alpha-beta searches, 191, 199–200, 202
animation frames and data-driven systems, 543–544
animation layer
  distributed AI design, 521–524
  in *Super Mario Bros.*, 536
animation picking, selection, 8, 160, 171, 175
anticipation in FTPS games, 127, 130
ApplyBehaviorRule( ) function, 435
arbitration, message, 332–333
'arcade AI,' 116
arcade-style games, 31, 33
artificial intelligence, *see* AI
Artificial Intelligence: A Modern Approach (Russel and Norvig), 4, 9

TeamLRN

artificial life (alife)
disciplines, 492–495
games, 235–237
pros and cons of using, 495
techniques in games, 510
using in games, 491–492, 495–496
artificial neural nets, 455–458, 489
Asteroids
and AIsteriods test bed, 49
and genetic algorithms, 428
Atari's *Gran Trak*, 177
Atari 2600, 33
attitude, basing game solutions on, 128
auto-retreating in real-time strategy games, 98
automated testing, 549–550
Avalon Hill and war games, 223
avoidance
obstacle, *see* obstacle avoidance
simple, potential fields method, 41–42
axons described, 453

B

backgammon, 200
background in fighting games, 206
backpropagation in neural net learning, 459
backward chaining inference, 501
Baldur's Gate, 71, 74, 76–79, 83, 347
bartering systems, 75
BasicButton class, 566–567, 575
basketball games, 157–159, 161, 163, 173, 522–524
Battle Chess, 201
battlefield planning and ground control method, 378
behavior layer, distributed AI design, 519–520
behaviors
AI and, 4
and animation selection, 160–161
blending with fuzzy-state machines, 104
coherent AI, 556
competition in racing games, 182
complex moves in fighting games, 208
emergent, in alife, 495
Game Theory and, 187
modulators' effect on, 14
range, with FuSMs, 303
robotics theory of mind, 26–27
steering, and obstacle avoidance, 45
Super Mario Bros. characters, 534–535
Big Blue (chess computer), 10
binary decision trees (BDTs)
see also decision trees
areas for exploitation within games, 506–507
described, 503–506
binary variable crossover operators (fig.), 423
bitwise IM, 381, 403
BitwiseInfluenceMap class, 397–405
Black & White, 32, 236–237, 415, 457, 492, 578
Blade Runner, 92
bonus lives, GameSession update loop, 60
boss enemies
in fighting games, 206, 210
in FTPS games, 116
in platform games, 137, 142
in role-playing games, 73–74
in shooter games, 151, 155
bots
deathmatch, 116
personalities of, 126
QuakeC sample, 113–115
bounded optimality (BO) in AI systems, 24–25, 212
boxing games, 203
brain, human
modulators, 14
nerves, neural networks, 453–455, 489
organization of, and game AI systems, 11–12
breeding programs and genetic algorithms, 237
Bridge, 198
Brooks, Rodney, 26
Brooks subsumption and distributed AI design, 527
brute force searches
and artificial intelligence, 188
in chess programs, 190
and genetic algorithm solutions, 452
Buffy the Vampire Slayer, 204, 209
build orders and resource management, 102
buildObjective ( ) function, 223
bullets, 51–52, 145, 265
bytecode scripts, 371

C

CAAIControl class, 441–442
Cabal, 154, 155
CalculateActivation ( ) function, 308
Call to Power II, 219
callbacks, 318–319, 340, 566
camera angles, systems
in 3D games, 36
in adventure, stealth games, 90
in fighting games, 206, 210
in platform games, 138–139, 141–143
in sports games, 161, 162
Capcom'sStreet Fighter 2, 203
Carmack, John, 119
Carmageddon, 183
cart racing game, 178
CD-ROM, contents and system requirements, 581–582
Cellular Automata (CA), 492–494
Centipede, 152
character-based behavioral intelligence, 4
characters
AI, and distributed design, 516
enemies, *see* enemies
fuzzily controlled, 304
in God games, 221–222
multiple intracharacter FSM interactions, 270
nonplayer (NPCs), 74
off screen, 546–548
in platform games, 131
reactivity, *see* agent reactivity
in squad combat games, 115
cheating
and breaking rules, 31
in character-based games, 6
in sports games, 157–158
checkers games, 188, 200
CheckTransitions ( ) function, 255, 269
chess games, 10, 188–198, 200, 201
choke points in maps, 379
chromosomes, 414
circle strafing, 30
civilization games, 211–214
Civilization screenshots (figs.), 212, 213
Clank, 37
classic strategy games

common AI elements, 198
described, 200–201
Game Theory and, 187–191
use of search ( ) and think ( ) functions, 191–198
useful AI techniques, 199–200
classification in neural nets, 461, 463, 505
client handlers, 312, 318–319
cloning, 417
coach-level elements in sports games, 158–159, 174
cockpit controls in flight simulators, 229
coding, dynamic hard, 13
coevolution, 449
cognition described, 15–17
coherent AI behavior, 556
Coleco-Vision, 33
collision avoidance systems
in game development, 7
GameObj class, 49–51
with neural nets, 464–465
collision checking in AIsteroids test game, 57
collision state calculation diagram (fig.), 429
collision systems in fighting games, 206
collisions
bullets, 54
highly scaled gametime problem (fig.), 444
combat games
AI-controlled intelligent system, 23–24
racing hybrid, 181
squad combat games (SCGs), 115
vs. role-playing games, 81
combinatorial explosion, 283
Command and Conquer, 107
comments in script files, 340
committee of machines technique, 485
common AI elements
in classic strategy games, 198
in fighting games, 205–207
FTPS games, 116–119
in platform games, 137–139
in racing games, 179–182
in shooter games, 150–151
in sports games, 158–162
communication
messaging, *see* messaging
between objects, 47
compact disk contents, system requirements, 581–582
comparison evaluators, using in scripted FSMs, 271
competition
learning in self-organized maps, 486
in racing games, 182
compiling scripts, 365, 371
computation costs
see also CPU time, allocation
in chess programs, 190
and game design, 38–40
and load balancing, 273–274
and racing games, 185
scripting, 362–363
of terrain analysis, 406
and using genetic algorithms, 445
Computer Graphics Technology Group, 346
configuration script systems, 339–340, 346
confirmation messages, 333
connectivity of neural nets, 457
consoles
debugging, 561
as game platform, 33
Sega *Genesis*, 97
control-based IM, 381
Control class, AIsteroids test game, 61
ControlInfluenceMap class, 393–396
conversation engines in FTPS games, 127
conversion engines and grammar machines, 82
cooperative agents, 117–118, 129
cooperative elements
in platform games, 138, 142
in shooter games, 152, 156
CopyEliteInto ( ) function, 438
costs, computation, *see* computation costs
CPU time, allocation
see also computation costs
and brain processing, 455
enemies and, 7
fuzzy logic and, 511
and game design, 338
high-level graphics and, 6–7
and level-of-detail (LOD) AI, 545
load balancing, 38
crack.com, 131
Crash Bandicoot, 37
crashing console platforms, 33
creativity in FTPS games, 127
Creatures, 492
crossover operators in individual reproduction, 422–427
CrossUniform ( ) function, 439–440
crowds in sports games, 161

D

danger signification, 15, 378, 404
data perceptions and cognition, 15
data-driven systems
cameras for 3D platformers, 140
in civilization games, 213
common AI development concerns, 542–544
in fighting games, 207
FSMs, 271, 279
FuSMs, 305–306, 310
in rhythm games, 234
in shooter games, 153
in sports games, 170–171
in war games, 228
using in real-time strategy machines, 105
deathmatch AI
bots, 129
and finite-state machines, 119
in FTPS games, 116, 124
debug data fields, 61
debug draw function, 398
debugging, 365
AI systems generally, 559
built-in, scripting systems, 365–366
controlling variables with widgets, 562–575
debug draw function, 398
determining when AI element is stuck, 108–109
FSM methods, 266
fuzzy systems, 302
games, 8
influence mapping, 381, 561
LBI systems, 405
messages, 333, 335
neural networks, 461–464
scripting systems, 363–367
and state-based systems, 35–36
summary, 575–576
with system hooks for, 61
Decent 3, 502
decision game types, 32

TeamLRN

decision-making systems
AI engine design, 30–37, 577
in chess programs, 200
decision trees, 503–507
distributed AI design, *see* distributed AI design
and game design, 23, 25
in God games, 222
human brain frontal lobe, 12
planning and, 496
production systems, 500–502
realism, and FSM-based, 276–277
robotics, 27
in sports games, 159–160
and state systems, 241
decision trees
see also binary decision trees (BDTs)
described, using in games, 503–507, 510
decisions, short- and long-term, 525–526
Defender, 40, 145
Defense Advanced Research Project Agency (DARPA), 502
defuzzification, 508
Descartes, Rene, 491
design
AI engine, steps for, 46–47
distributed AI, *see* distributed AI design
general AI design thinking, 550–551
lessons from robotics, 26
designing
AI engines, 46–48, 541–551
distributed AI, *see* distributed AI design
FSMs, considerations, 276–278
FuSMs, 306–308
genes, 428
genetic algorithms, 450–451
location-based information systems, 407–408
message-based systems, 334–335
neural networks, 487–488
scripting systems, 337–339, 368
Deus Ex, 93
Diablo games, 71, 73–74
diagrams
artificial neuron, neural network (fig.), 456
collision state calculation (fig.), 429
FSM, *see* FSM diagrams
neuron (fig.), 454
Soar architecture (fig.), 502
difficulty levels, 36–37, 208
Digenetics' *Checkers with an Attitude*, 200
diplomacy systems, 103, 213, 450
distributed AI design
AI engine layers, 577
distributed layers, 517–526
overview of, 515–516
summary, 538–539
documenting FSM diagrams, 266
Donkey Kong, 40, 131, 140
driving simulations, 178
Dune: Building a Dynasty, 107
Dungeon, 87
Dungeon Siege, 84–85
Dungeons and Dragons, 81
dynamic hard coding, 13
dynamic objects, storing in Standard Template Library list structure, 57
dynamic skill level adjustments, 36

E

economic AI, 112
economic individual units, 98
Elder Scrolls games, 80
elitism in genetic reproduction, 417, 421–422
Eliza (program sample code), 18–22
embedding Lua in game environment, 346–355
emergent behavior in alife, 495
emergent properties, 492
Empire Interactive, 223
enemies
see also opponents
AI implementation, *Super Mario Bros.*, 529
in character-based games, 6–7
designing, 554–555
in fighting games, 205–206, 210
in FTPS games, 116, 129
making better, 84
more intelligent, in racing games, 184
in platform games, 142
in racing games, 181
in RPGs, 72–73
in shooter games, 150–151, 155
in stealth games, 88–89
ENIAC computer, 188
entertainment
common AI development concerns, 551–555, 558
FSM limitations on, 278
FuSM, limitations on, 308
game development limitations, 36–37
genetic algorithm solution limitations 451
location-based information systems limitations 408
message-based system limitations 335
neural nets limitations 488
scripting system limitations, 370
value of zero-sum games, 198
Evade state, 396–397
EventHandler class, 562, 566
events
layer, distributed AI design, 519
vs. polling as perception register updating, 39
Everquest, 82
evolution
in games, 415–416
and genetic algorithm solutions, 413–415, 452
time-consuming, 446
evolution application (EA), implementing, 431–442
execution flow, classis vs. modular FSM (fig.), 242
Exit ( ) function, 255, 259
expert systems, 500–502
explosions of objects, 54
exploits and sports games, 172

F

Faile chess program, 191
False Belief Task, 17
feature creep, 448
feed forward (FF) networks, 455–456
fighting games
areas that need improvement, 209
common AI elements, 205–207
examples, summary, 208–210
introduction to, 203–204
pseudocode for *BigPunch* (listing), 520
useful AI techniques, 207–208
war games, 223–228
figures in book, on CD-ROM, 581
filters and AI systems, 16, 28

Final Fantasy, 67, 73–74, 76
FindError ( ) function, NeuralNet class, 466
finite-state machines (FSMs), 78
in adventure, stealth games, 90
AIsteroids test game performance, 264–267
in classic strategy games, 199, 202
design considerations, 276–278
extensions to, 268–273, 268–273
diagrams, *see* FSM diagrams
in fighting games, 207, 210
in FTPS games, 119
framework, classes, 279
implementing ship in AIsteroids program, 250–264
optimizations, 273–276
overview of, 241–244
Pac-Man free-form implementation (listing), 244–245
in platform games, 139, 142
pros and cons of using, 266–267
in racing games, 180, 182
in real-time strategy games, 103
in shooter games, 152, 156
skeletal code, 245–250
in sports games, 163, 175
summary, 278–279
and transition systems, 241
using FuSM as single state within, 305
firing bullets in Alsteroids test game, 52–53
first-person shooters/third-person shooters (FTPS), *see* FTPS games
fitness function, 416–417, 420–421, 452
flight simulators (SIMs), 229–233
FlightGear, 229–233
flocking, and self-organizing behavior, 494
fluid gameplay sports, 157–158, 174
'fog of war' line-of-sight system, 376
football games, 160
forward chaining inference, 501
Freecell, 234
FreeCiv, 213
FSM diagrams
for asteroids (fig.), 250
documenting, 266
Pac-Man's Blinky (fig.), 243
Robotron player (fig.), 284
FSMAIControl class, 249–250
FSMMachine class, 247–249
FSMs, *see* finite-state machines (FSMs)
FSMState class, 245–247
FStateApproach implementation (listing), 296–297
FStateAttack implementation (listing), 297–298
FStateEvade implementation (listing), 298–299
FStateGetPowerup implementation (listing), 299–300
FTPS (first-person shooters/third-person shooters) games
areas that need improvement, 125–128
common AI elements, 116–119
examples, 124–125
introduction to, 88, 113
summary, 128–130
useful AI techniques, 119–124
war games, 223–238
Full Throttle, 92, 93
fun factor in game design, 551–553
functions, *see specific functions*
FuSMAIControl class, 289–290
FuSMMachine class, 288–289
FuSMs, *see* fuzzy-state machines (FuSMs)
FuSMState class, 286–287
fuzzy logic
see also fuzzy-state machines (FuSMs)
in adventure, stealth games, 91–92
described, using in games, 507–510
in sports games, 163
systems, and FuSms, 283
fuzzy-state machines (FuSMs)
see also fuzzy logic
coding control class, 293–300
described, 309
diagram for asteroids game (fig.), 292
extensions to paradigm, 304–306
implementing FSM within, 284
implementing FuSM-controlled ship, 290–293
in FTPS games, 123
in real-time strategy games, 103–104
in sports games, 163, 175
optimizations, design considerations, 306–308
overview of, 281
performance of implementation, 300–304
skeletal code, classes, 285–290
summary, 308–310
using as single state within FSM, 305
using in alife games, 237
variants of, 282–283

G

GAAIControl class, 452
GAMachine class, 431–442
game AI
and academic AI, 9–11
AI engine, *see* AI engine
conclusions, 577–579
cooperative elements, 89
secondary systems, 7
summary of definitions and concepts, 28–29
system, and the brain (fig.), 12
term defined, 1, 3
timeline (fig.), 5
game balance
in FSMs, 278
in sports games, 173
game development, 4
AI engine design steps, 46–47
limitations on, 34–37
game solutions
local vs. global, 446
message-based systems, 334
planning algorithms, 499
scripting systems, 368
types of, 30
using FSMs for, 276
using FuSM types, 306–307
using genetic algorithms, 445
using location-based information systems, 407
using neural nets for, 487
Game Theory, principles and concepts of, 187–191
Gameboy, Nintendo, handheld platform for game design, 34
GameObj class, 49–52
GameOver ( ) function, 60–61
gameplay
diagrams for classic role-playing games, 79–80
programming, 5
testing for FTPS games, 118
games
see also specific game

TeamLRN

adventure, 87–95
artificial life usage in, 492, 495–496
breakdown of, 527–538
evolution in, 415–416
fighting, *see* fighting games
first-person shooters/third-person shooters (FTPS), *see* FTPS games
future of, 578–579
genres, *see* genres of games
high- and low-level character interactions, 374–375
integrating scripts into, 350–355, 366
miscellaneous genres, 211–237
platform, *see* platform games
reviews of, 578
rhythm, 233–234
role-playing, *see* role-playing games (RPGs)
shooter, *see* shooter games
'solved' by researchers (fig.), 190
sports, *see* sports games
squad combat (SCGs), 115
'state space,' 16
stealth, 40
strategy, *see* classic strategy games
tokens, 340–345
traffic systems, 179–180
'twitch,' 31
voice commands, 579
zero-sum, 188
GameSession class, AIsteroids test game, 55–61
Gauntlet, 525
Gene class, 430–431
General Problem Solver (GPS), 500
generational reproduction, 417, 421
genes, 414, 417–418, 449
genetic algorithms (GAs)
and artificial life, 494
basic genetic method, 416–417
and breeding programs, 237
in classic strategy games, 202
design considerations, 450–451
extensions to paradigm, 448–449
implementing into AIsteroids test bed, 428–448
overview of, 413–416, 445–448
representing the problem, 417–427
in strategy games, 200
summary, 452
genetic programming, 449
genetic recombination, 414
genome, 415, 436
Genome class, 430–431
genres of games
see also specific game genre
described, 31–32, 65
first-person shooters/third-person shooters (FTPS), 88
and FSMs, 277
and FuSMs, 307–308
and genetic algorithm solutions, 451
location-based information systems, 408
and message-based systems, 335
miscellaneous, 211–237
neural nets and, 488
GetInfluenceValue ( ) function, 383
GLUT (OpenGL Utility Toolkit), 49, 63, 582
goals
in game breakdown, 527–538
planning, in real-time strategy games, 104–105
God games, 220–223
governing principles, 491–492
GPS (General Problem Solver), 500
grammar machines (GMs) in role-playing games, 82
Gran Trak, 177
Gran Turismo, 183, 413
Grand Theft Auto, 179, 181, 184
grid-based systems, navigation in, 40–41
ground control method and influence maps, 377
guards, nonplayer characters (NPCs), 89

H

Half-Life, 116, 117, 119
handheld platform and game design, 34
handlers, input, *see* input handlers
head-to-head fighting games, 206, 209
Help systems in platform games, 141–143
helper AI
in classic strategy games, 198, 202
described, using in real-time strategy games, 109
in real-time strategy games, 112
helper bots, 117–118
Heretic, 125
Herzog Zwei, 97, 107, 109
Hiatt, Adam, 145
hierarchical AI in real-time strategy games, 104, 112
hierarchical FSMs, 268, 279
hierarchical FuSms, 305, 310
high-level strategic AI, 99
hockey games, 157
Hopfield networks, 485
HumanControl class, 61
HumanControl.cpp, 63
HumanTestControl class, 431
hyperspace, human-controlled ship in, 49

I

IDE, providing for scripting systems, 366
if...then statements and fuzzy logic, 283
immersion in role-playing games, 71, 73
improving
AIsteroids test game, 264–265
fighting games, 209
racing games, 183–185
sports games, 172–174
strategy games, 201
IMs, *see* influence maps (IMs)
individual-unit AI, 110–112
inertia, using to prevent state oscillation, 272
inference and decision making in AI engines, 30–37
influence maps (IMs), 14–15, 373–375, 409, 561
InfluenceMap class, implementation, 382–405
Init ( ) function, 61, 466
initializing starting population, genetic algorithms, 416
input game types, 31–32
input handlers
and AI system design, 37–40
and deathmatch-style FTPSs, 124
input layer in neural nets, 455
integrating
scripts into games, 366
widgets within programs, 567–575
intelligence
and academic AI, 10
appearance of, and game development, 17

Brooks subsumption, 527–538
defined, 3
Interact ( ) function, 405
interactive fiction games, 87, 93–94

J
Java
as scripting language, 337
version of Eliza (sample code), 18–22
Joust, 35, 503–504
jump steering, 534
Jurassic Park: The Lost World, 151

K
Karate Champ, 203
Kasparov, Gary, 189
keyword systems, 82
kill zones, 14–15
knowledge base learning, and human memory, 12–14

L
Laird, John, 501, 502
language and theory of mind, 22
layers
designing AI engine, 577
distributed AI, 517–526
in neural nets, 455, 459, 464
LBI systems, *see* location-based information (LBI) systems
learning
and game AI, 14
human memory, and knowledge base, 12–14
in neural nets, 459–460, 463–464
opportunities in real-time strategy games, 108
LED football games, 157–158
Legend of Zelda, The, 78, 543
level of detail (LOD) AI systems, 274, 279, 545–546
'Lex & Yaac' scripting, 365
libraries on CD-ROM, 581
light gun games, 154
load balancing
in AI game design, 38
both FSMs and perceptions, 273–274
location-based information layer
distributed AI design, 526
in *Super Mario Bros.*, 538
location-based information systems
design considerations, 405–409
extensions to paradigm, optimizations, 406–407
influence maps, *see* influence maps
overview of, 373–376
LOD (level of detail) AI systems, 274, 279, 545–546
Loeb, Max, 562
logic
Boolean, and fuzzy, 281
fuzzy, *see* fuzzy logic
spatial reasoning, 119
long-term decision making (LT)
in distributed AI design, 525–526
in *Super Mario Bros.*, 537–538
long-term vs. short-term memory, 13
LookingGlass Studios, 90
loops
AIsteroids main game, 63
debugging useless, 108–109
Lord of the Rings (movies), 418
Lua programming language, 346–355, 371
LucasArt's SCUMM system, 93

M
Madden, 172, 173
map node and grid systems, 42–43
maps
see also location-based information systems
and camera paths in platform games, 139
finding choke points, 379
grid squares and, 41–42
routes though in FTPS games, 126
self-organizing (SOM), 486
with several strategic elements (fig.), 380
Mario Kart, 178
Mario64, 136, 140
marketing and sports games, 173–174
Markov models and FuSms, 282
Mars, robotics on, 26
massively multiplayer online role-playing games (MMORPGs), 81, 184, 494
Maximo, 139
MCell output (fig.), 493
MEA (means-end analysis), 498
Mealy machine model, 243
means-end analysis (MEA), 498
Mega Man screenshot (fig.), 132
Meier, Sid, 211
memory
associative, and Hopfield networks, 485
human vs. AI systems of, 12–14
RAM in arcade platforms, 33
reinforcement and degradation, 13
message arbitration, 332–333
message-based FSMs, 268–269, 279, 345–346, 355
message-based systems
client handlers, 318–319
coding the states, 327–329
design considerations, 330–331, 334–336
example implementations, 319–327
extensions to paradigm, optimizations, 332–334
messaging skeletal code, classes, 313–318
overview of, 311–312
performance of implementation, 329–332
message object, 313–314
message pumps, 312, 332
messagepump class, 314–318
messaging systems
in adventure, stealth games, 91
example of (fig.), 312
in FTPS games, 129
in platform games, 139–140, 142
in real-time strategy games, 103, 112
in racing games, 182–183
in role-playing games, 78–79
in sports games, 171, 175
in squad combat games (SCGs), 124
MessAIControl class, 321–327
MessMachine class, MessState class, 320–321
Microsoft Flight Simulator, 229
Midtown Madness, 185
mind science, 11–12, 17–18, 24–25
minimax
as planning algorithm, 498
search trees, methods, 199–200
mistakes and 'realistic' game behavior, 31
MMORPGs (massively multiplayer online role-playing games), 81, 184, 494
models of AI, 11–12, 282
mods (user-made modifications), 113, 130, 282

TeamLRN

Molyneux, Peter, 220, 221
Monopoly, 188
Monster Rancher, 236
Moore model, state machine, 242–243, 266
motion capture, animation selection in sports games, 160
motion layer, 524–525, 536
MStateIdle header, implementation (listing), 327–329
MutateOffset ( ) function, 440–441
mutations in evolution, 414–415
Myst, 87

N

Napoleon, 223
navigation
  AI engine design, 40–46
  mesh systems, 43–44
  in sports games, 161
  systems paradigms, 48
NBA Shootout 2004, FSM behaviors (listing), 163–170
need-based systems and towns, 86
neighborhoods in neural nets, 486
NES (Nintendo Entertainment System), 153
NetHack, 68–71
networks
  early stopping, 481
  feed forward (FF), 455–456
  Hopfield, 485
  map node, 42–43
neural nets (NNs)
  artificial, overview of, 455–458
  described, 453–455, 482–484, 489
  extensions to paradigm, 484–487
  game characters and, 236
  implementing into AIsteroids test bed, 464–481
  neural network activity, 461–464
  optimizing, 481–482
  and perceptrons, 237
  in strategy games, 200
  training, 460
  using in games, 458–460
  vs. decision trees, 504–505
NeuralNet class, 465–469
neurons
  artificial (fig.), 456
  described, 453, 454
  in neural nets, 489
Neverwinter Nights, 81
Newell, Alan, 500, 501
Newton, Isaac, 491
Nintendo Entertainment System (NES), 153
Nintendo Gameboy, 34, 234
NLayer class, 470–473
NNAIControl class, 473–480
NNs, *see* neural nets (NNs)
nonplayer characters (NPCs)
  in adventure and stealth games, 74, 89, 94
  helper bots, 117–118
  NPC AI, *see* NPC AI
  off screen characters, 546–548
  in racing and sports games, 161, 181
  in towns, 85–86
Norvig (*Artificial Intelligence: A Modern Approach*), 4, 9

O

objects
  see also specific objects
  avoiding, *see* obstacle avoidance
  communication between, 47
  explosions, 54
  and game object classes, 63
  message, described, 312
obstacle avoidance
  dodging and, 45
  improving *AIsteroids*, 265
  and navigation, 40
  pathfinding and, 118–119
obstacles in RPGs, 72
occupance-based IM, 381, 392
occupance data, using influence map to capture, 376
OccupanceInfluenceMap class, 387–392
Oddworld games, 137
off screen characters, 546–548
one-track mind syndrome, 544–545
online games, massively multiplayer (MMORPGs), 81, 184, 494
OnOffButton class, 568–569
OpenGL Utility Toolkit (GLUT), *see* GLUT
opponent AI, 186, 198
opponents
  see also enemies
  in classic strategy games, 201–202
  deathmatch, 116
  and fuzzy-state machines, 123
  modeling, in real-time strategy games, 101–102
optimality, bounded, 24–25, 212
optimizing
  collision checking, 57–58
  FSMs, 273–275
  FuSMs, 306
  influence map functions, 409
  location-based information systems, 406–407
  message-based systems, 333–334
  neural nets (NNs), 481–482
  scripting systems, 367–368, 371
order queues in real-time strategy games, 98
organic creature movement, 45
oscillations
  behavior, with FuSM, 303
  common state-based problem (fig.), 267
  state, 267, 272, 561
output game types, 32
output layer in neural nets, 455
overfitting neural nets, 482

P

Pac-Man, 39, 419
  encoding genes, 418
  FSM implementation, 243–245
PaRappa, 234
party AI, extending, modifying, 83–84
party members, nonplayer characters (NPCs), 75–76
pathfinding
  and AI navigation, 8, 40, 555
  and bounded optimality, 25
  for FTPS games, 118–119, 129
  and load balancing, 274
  more information on, 46
  and motion layer in distributed AI design, 524
  in real-time strategy games, 100–101, 112
  and robotics, 26
  in sports games, 160–161, 175
  system helper data, and influence maps, 377–378
  using influence maps, 526
pattern recognition
  fuzzy logic in, 508
  in neural nets, 457, 488
payoffs in Game Theory, 187
PCs as game platform, 32–33
perception game types, 31–32, 48
perception registers, updating, 39
perception systems, 90

perceptions
and AI engine design, 37–40, 519
and cognition, 15
fuzziness in sports games, 163
and fuzzy logic, 91–92
including inertia, 272–273
and perceptrons, 237, 455
types of, 37
perceptrons, 237, 455
personal computers (PCs) game platform, 32–33
personalities
in FTPS games, 126
lack of, in games, 578
opposing, 109–110
Pac-Man's Blinky, 244
perturbation learning, 460
physics models and racing games, 178, 185
planning
algorithms, 496–500
goals, in real-time strategy games, 104–105
preprocessed, 496
use in games, 510
plasticity in memory systems, 13
platform games
areas that need improvement, 141–142
cameras in, 207
common AI elements, 137–139
introduction to, 131–132
useful AI techniques, 139–141
platforms
console and computer RPGs, 67
FSMs and, 277
for FuSMs, 308
and game design, 32–34
for genetic algorithm solutions, 451
for message-based systems, 335
and scripting systems, 369
playbooks for sports games, 171
player-level AI in sports games, 159
player statistics, programming, 171
players
in civilization games, 211
in God games, 220
keeping on 'rails, 151
off screen characters, 546–548
in real-time strategy games, 97–98
state-based model, state oscillation, 267
playfield conditions, 157–158, 187
PlayStation, 9
Point Blank screenshot (fig.), 154
poker, 101, 188
Pole Position, 177, 183
Polge, Steve, 113
polish in games, 448, 556–557
polygons, 542
polythetic neural nets, 505
Pong-type games, 153, 157
Pontifical Catholic University, Brazil, 346
Populous screenshot (fig.), 220
potential fields and obstacle avoidance, 42, 45
POV-Ray, 542
powerup objects, 54, 59–60, 123, 404
predictability and state-based systems, 35
premature convergence (PMC), preventing, 420
preprocessed planning, 496
primitives, and 3D rendering, 542–543
Prisoner's Dilemma, 187
probabilistic FSMs, 282
processes, breaking down for AI design, 48
ProcessMessage ( ) function, 319, 336
production systems
common AI development concerns, 555–558
history of, using, 500–503
Propagate ( ) function, NeuralNet class, 466
prototyping AI engines, systems, 47, 48, 360–361
psychology theory
brain organization, 11–12
and the fun factor, 551–553
and game theory, 11
puzzle games, systems, 79, 87, 136–137, 234–235
Python programming language, 347

Q

Quake, 30, 113–115, 125, 502, 543
quest generators in role-playing games, 82–83
quests, scripted, 80

R

racing games
areas that need improvement, 183–185
common AI elements, examples, 179–183
generally, 177–179, 185–186
rubber banding in, 6
useful AI techniques, 182–183
RadioButton class, 568
rails, player, 151
randNorm ( ) callback, 553
randomness, perceived, 553–554
rank scaling, fitness values, 421
rapid prototyping scripting systems, 360–361
Ratchet, 37
reaction time
agents in FuSM solutions, 307
AI game engine programming, 38
reactivity of agents, *see* agent reactivity
real game AI development, 515
real-time strategy (RTS) games
areas that need improvement, 108–111
cheating and, 6
coach- and team-level AI, 158
common AI elements, 97–103
examples of, 107–108
'fog of war' line-of-sight system, 376
generally, 97, 111–112
and influence maps, 374
input handlers and perception, 37
and multiple layered decision architecture, 27
planning algorithms in, 497
societal evolution in, 449
tactical and strategic support systems, 101–102
terrain analysis, simple and advanced, 378–380
useful AI techniques, 103–106
realism
and the AI engine, 31
and FSM-based decision making, 276–277
in genetic algorithm solutions, 450
in influence map-enabled games, 408
and message-based systems, 334
with neural nets, 488
and scripting systems, 369
reason, thought, and intelligence, 3
recurrent networks, 457, 485
Regimbald, Adrien M., 191–198
RegisterForMessage ( ) function, 315
regression in neural nets, 461–464

TeamLRN

regression trees, 510
reinforcement learning, 459, 486
reliability models, 282
reproduction, generational and steady state, 421–427
resetting gameplay in sports games, 158, 163, 174
RETE algorithm, 503
reviews of games, 578
Reynolds, Craig, 45, 494
rhythm games, 233–234
robotics, 25–28, 494
Robotron, 32, 37, 155, 270, 284
role-playing games (RPGs)
finite-state machines (FSMs), 78
genres of games, 86
and influence maps, 376
overview, enemies to, 67–74
massively multiplayer online (MMORPGs), 79
scripting, 77
turn-based, 75–76
vs. combat games, 81
Rosenbloom, Paul, 501
roulette wheel selection, 422
RPGs, *see* role-playing games (RPGs)
RTS games, *see* real-time strategy (RTS) games
rubber banding in racing games, 6
rules
breaking, and 'cheating,' 31
for cellular automata, 492–493
in God games, 223
Russel (*Artificial Intelligence: A Modern Approach*), 4, 9

S

Sargon, 201
Saucer game implementation, 289–293
scaling
BDTs, 506
fitness values of individuals, 421
FuSMs, 302
neural nets (NNs), 484
scripting systems, 360, 361
Script Creation Utility for Maniac Mansion (SCUMM), 93
scripting
see also scripting systems
in fighting games, 205
in FTPS games, 129
overview of, 337–339
in platform games, 143
in racing games, 186
in rhythm games, 234
role-playing games, 77–80, 86
using in real-time strategy machines, 105
in war games, 228
scripting systems
in adventure, stealth games, 91
built-in debugging tools, 365–366
in *Call to Power II*, 219
debugging, 561
FSMs, 271
costs, considerations, 360–364
design considerations, 368–370
embedding Lua in game environment, 346–364
extensions to, 364–367
in fighting games, 208
in FTPS games, 124
optimizations, 367—368
in platform games, 140
in racing games, 182
scaling, 360, 361
in shooter games, 152
summary, 370–371
test game performance, 345–346
scripts
behavior, in *Super Mario Bros* (listing), 532–533
Lua, running in test bed, 355–360
self-modifying, 367
scrolling platform games, 131
scrubber widgets, 562
ScrubberWidget class, 568–569
Seaman, 236
search ( ) function, 191
searches
alpha-beta, 191
brute force searches, *see* brute force searches
stochastic methods, 416
Sega Genesis console, 97
selection functions and fitness functions, 422
SelectRouleteWheel ( ) function, 438–439
self-organizing behavior, maps, 486, 494
SetUpNextGeneration ( ) function, 434, 438
Shannon, Claude, 188
sharing scaling, fitness values, 421
ships
controlling with FuSm, 290–304
controlling with neural net, 464
implementing FSM-controlled, 250–264
object (AIsteroids), 52–53
sample Lua scripts to control (listing), 357–358
saucer-type, 290
shooter games
common AI elements, techniques, 150–153
examples, exceptions, summary, 153–156
introduction to, 145–146
short-term decision making (ST), 525, 535
short-term memory vs. long-term, 13
side scrolling platformer, 528
sigma truncation, scaling fitness values, 421
SimCity, 83, 203, 211, 220, 221, 222
Simon, 234
Simon, Herbert, 500
Simpsons, The: Hit and Run, 183
Sims, The, 236, 375
simulated annealing, 460
smart terrain, 373, 375, 404, 409
Soar project, 501–502
soccer games, 157
societal evolution, 449
source code on CD-ROM, 581
spatial reasoning in FTPS games, 119, 129
speed
of execution, and scripting, 362
for role-playing games, 71
and scripts, 367
in strategy games, 201, 202
sports games
camera styles in (fig.), 162
generally, 171–175
introduction, common AI elements, 158–162
racing, *see* racing games
useful AI techniques, 163–171
squad AI described, 129
squad combat games (SCGs), 115, 118, 125, 128
stack-based FSMs, 279
stacks in Lua environment, 348
stalling messages, 333
StampInfluence ( ) function, 383

Standard Template Library (STL), 57
state-based systems
  finite-state machines (FSMs), 78
  and game topology, 16–17
  and predictability in games, 35
state diagrams, *see* FSM diagrams
state machines
  described, 241, 278
  and transitions, 242
  vs. messaging, 311
state oscillation, 267, 272, 561
'state space' of game described, 16
StateApproach class functions (listing), 255–257
StateAttach class functions, 257–259
StateEvade class functions, 259–261
StateGetPowerup class functions, 261–262
StateIdle class functions, 263–264
StateNNEvade class, 475, 480
states
  activation levels in fuzzy systems, 281
  coding in messaging systems, 327–329
  collision, diagram of calculation (fig.), 429
  transitions, *see* transitions
  and wave attacks, 151
steady state reproduction, 421
stealth games, 40, 88–90, 92–93
steering behaviors and obstacle avoidance, 45
stochastic
  search methods, 416
  universal selection, 422
storing
  memory in brains, 455
  messages, 313–314
strafing circle, 30
strategy games, 187–202
Street Fighter 2, 203, 204, 207
subsumption architectures, 27
SumInfluence ( ) function, 383
Super Mario Bros., 521, 528–538
support AI, 548–550
synaptic gap, 453
system realism, *see* realism
system requirements, CD-ROM, 582

T

TA, 407
tactical game solutions, 30
Tamagotchi, 236
team-level elements
  in shooter games, 159–160
  in sports games, 174
teammate AI, 172
Tekken Tag Tournament screenshot (fig.), 205
terrain analysis (TA)
  cover, visibility, height, 378–379
  described, 375–376, 379–380, 409
  as location-based information system, 373
  in real-time strategy games, 101
testing
  automated, 549–550
  of CD-ROM demos, 582
TestSession class, 431
Tetris, 234, 235
Theory of Mind (ToM), 17, 24, 26–28
Thief games, 88, 89, 90, 92
think ( ) function, 191
third-person shooters, *see* FTPS games
thought, reason, and intelligence, 3
thresholds
  and AI game design, 38
  in human nervous system, 453
tic-tack-toe, 188
time limits
  an chess programs, 189
  messages, 315
  for occurrence data, 404
timeline, game AI (fig.), 5
tokens, game, 340–345
Tomb Raider, 88, 92
topology and game design, 16–17
Total Annihilation, 99
'total conversion' of game described, 113
tournament selection, 422
town building, 85–86, 100, 112, 378–380
track AI in racing games, 179–180, 185
training neural networks, 460
transition systems, 241, 269
transitions, handling with Lua scripts, 358
Traveling Salesman Problem (TSP), 188, 418–419
TRETE algorithm, 503
tuning AI behavior
  artificial life, 495
  beta testing, 548–549, 556–557
  and scripting systems, 361
  and visual debugging, 559–561
Turing, Alan, 22, 188
Turing test, 188
turn-based RPGs, 75–76, 211
Twisted Metal, 178
'twitch' games, 31, 89
twitchy character artifacts, 273

U

Ultima 7, *SimCity* screenshot (fig.), 83
Ultima Online, 81
unit grouping and better per-unit behavior, 110
Unreal, 113, 125
Update ( ) function
  controlling AIsteroids ship, 252
  described, 51–52
  and FuSMAIControl class, 291–296
  and FuSMState class, 286
  and GAMachine class, 433–434
  in genetic algorithms, 434
  in messaging system, 315
  and StateNNEvade class, 480
update functions
  FuSM Robotron player (fig.), 285
  GameObj class, 51–52
  UpdatePerceptions ( ) function, 62, 252
UpdateMachine ( ) function, 288–289
UpdatePerceptions ( ) function, 62, 252, 440–442
updating perceptions, 37–39
user extensibility of scripting systems, 361
user-made modifications (mods), 113, 130, 282

V

vehicular simulation and racing games, 178
violence in videogames, 184
visual debugging AI systems, 559–561, 576
voice commands, 579
von Neumann, John, 187, 188

W

walls built to hold back AI attackers (fig.), 380
war games, 223–228
Warcraft, 100, 107
Wargamer: Napoleon 1813, 223

TeamLRN