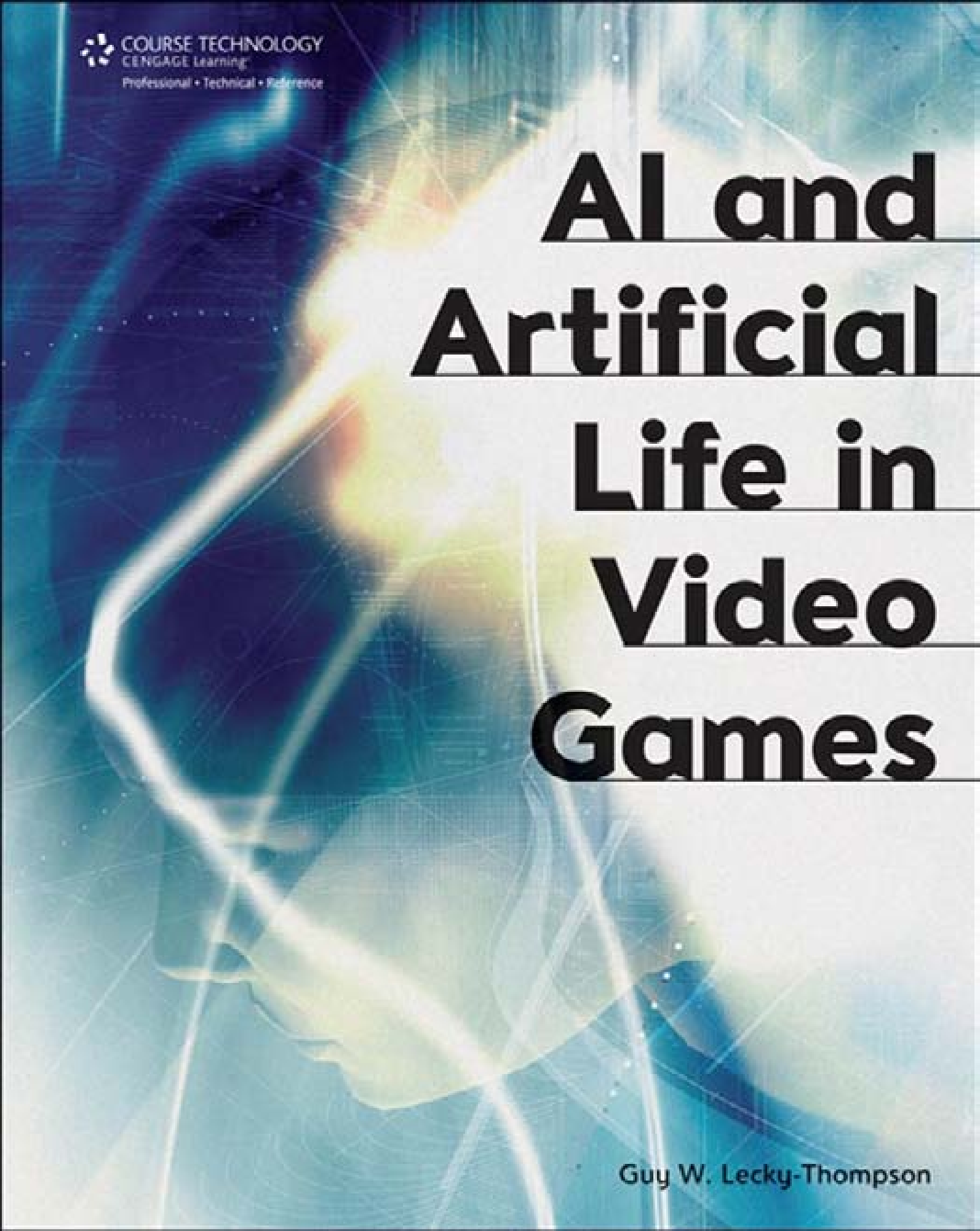
COURSE TECHNOLOGY
CENGAGE Learning
Professional • Technical • Reference
AI and Artificial Life in Video Games
Guy W. Lecky-Thompson

AI and Artificial Life in Video Games

Guy W. Lecky-Thompson

Charles River Media
A part of Course Technology, Cengage Learning

COURSE TECHNOLOGY
CENGAGE Learning

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

COURSE TECHNOLOGY
CENGAGE Learning™

AI and Artificial Life in Video Games

Guy W. Lecky-Thompson

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

Associate Director of Marketing: Sarah Panella

Manager of Editorial Services: Heather Talbot

Marketing Manager: Jordan Casey

Senior Acquisitions Editor: Emi Smith

Project Editor: Karen A. Gill

Technical Reviewer: Mark Morris

CRM Editorial Services Coordinator: Jen Blaney

Copy Editor: Barbara Florant

Interior Layout: Jill Flores

Cover Designer: Mike Tanamachi

Indexer: Jerilyn Sprotson

Proofreader: Gene Redding

Library of Congress Control Number: 2007939377

ISBN-13: 978-1-58450-558-7

ISBN-10: 1-58450-558-3

eISBN-10: 1-58450-616-4

Course Technology
25 Thomson Place
Boston, MA 02210
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at: **international.cengage.com/region**

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

For your lifelong learning solutions, visit **courseptr.com.**

Visit our corporate Web site at **cengage.com.**

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

Dedication

This book is for my wife Nicole and my children, Emma and William.

Acknowledgments

Once again, I owe a debt of thanks to the skills and persistence of the editing team. Karen Gill deserves special mention for her work managing to keep the balance between my initial prose and the finished product (as well as making sure we all did what we were supposed to!). A big thank-you also to Barb Florant for actually editing the prose and making sure that I made sense all the way through.

Mark Morris deserves credit for helping keep me as technically correct as possible. He gave some very valuable insight into some of the techniques and technologies presented in this book.

The behind-the-scenes team also gets a mention for work in preparing the book and making it look as good as it does, especially Gene Redding, for proofreading; Mike Tanamachi, for sharpening my slightly amateur illustrations; Jill Flores, for laying out the final copy; and Jerilyn Sprotson, who gave the book the all-important index.

Finally, a big thank you to Emi Smith for helping deliver the project from inception to publication.

My family, once again, has stuck with me through the process of creating the book. My wife Nicole and children, William and Emma, all helped me, either knowingly or unknowingly, one way or the other, to write it.

About the Author

Guy W. Lecky-Thompson is an experienced author in the field of video game design and software development, whose articles have been published in various places, including Gamasutra and the seminal G*ame Programming Gems*. He is also the author of *Infinite Game Universe: Mathematical Techniques*; *Infinite Game Universe, Volume 2: Level Design, Terrain, and Sound*; and *Video Game Design Revealed*.

Contents

PREFACE

Welcome to *AI and Artificial Life in Video Games*, a book intended to challenge as well as inform designers/programmers who want to add artificial intelligence (AI) and artificial life (A-Life) to their next video game creation. This is not a textbook on AI, nor is it an attempt to write a definitive work on A-Life. Both fields are continually expanding, but some solid ground rules are already in place, which we can leverage in creating better games. AI helps build better games. It can increase difficulty while also helping the player with the actual mechanics of the game. In other words, AI helps the machine to act in a way that will challenge the player, as well as help him by reacting intelligently to his actions.

By a similar token, A-Life can add an extra dimension of playability and immersion, while also providing some unpredictability to the nature of standard AI behavioral models. AI deals with the game's actual thinking; A-Life deals with making the environment more lifelike.

Some games are simply not possible without AI—chess, for example—whereas other games are built upon the premise of A-Life—*Creatures* being perhaps the first and most well known. Either way, any game, at many phases of video game development, can use the techniques in this book.

Be warned, however, that this is not a book about AI in video games, as used in the industry today. Instead, it is a look at techniques that can be used to add AI and A-Life easily and cheaply to video games, without extending the current AI techniques beyond their current capacities too much. As Mark Morris, of UK games developer Introversion, states,

"So why aren't we seeing all these wonderful techniques in games now? Is it the fault of the men in T-shirts or those in white coats, and what do we need to do to ease the passage of research from the lab to production-level video games?"
[http://forums.introversion.co.uk/introversion/viewtopic.php?t=1216]

Despite the huge advances made in processor power and associated resources in both the PC and console gaming world, over the past two decades, AI in games has not advanced one iota. 99 percent of commercial games on the market in 2008 use a combination of the basic AI techniques from games of the 1980s—namely, state machines, rule-based systems, and planning techniques.

Surprisingly, AI and A-Life can be used not only to model behaviors *inside* a game, but they can also be used to help create tools, generate code, and test the game under development. Any real-time system that has to control multiple complex variables can benefit from developmental AI and A-Life, as well as use AI and A-Life techniques to properly and completely test the systems.

This book does introduce the basic AI building blocks, which are entirely necessary in the creation of video games, but it also delves into more advanced techniques, which have only been exploited in a few games to date. What I want to make clear right now is that techniques in this book could be applied to any of the mainstream games and *would* probably enhance them immeasurably.

The industry, however, has been slow to adopt the best AI and A-Life techniques, something that I hope can be influenced in part by my writing this book. As Mark Morris says,

> *"We constantly criticize the academics and say that their techniques would not work on real games, yet when they ask for some source code, we tell them that there are "IPR" issues or that we do not have time to work with them."*

This points to one of the problems that this book seeks to address: AI and A-Life do not get a look in because of the protectionist nature of the industry. For the record, Introversion is active in bridging the gap between the academic and speculative side and the mainstream "real games" side of the industry. This studio always seems to have time to help out.

To address this issue, this book has been constructed to try to show the current state of the art (which has not changed for 20 years, in some cases) and then add to those techniques in a way that produces something that bridges that gap. This book is organized into chapters that give the reader a path through the techniques, covering things that might be familiar, and using them as building blocks for those things that almost certainly will not be.

Chapter 1, "Introduction," delves into the possibilities of using AI and A-Life, while also presenting a quick refresher course on what they actually are. The intention is to establish some high-level definitions of the concepts that we will work with specifically in video games, rather than attempt to discuss all-encompassing scientific definitions that can be used in any situation.

Having digested this, Chapter 2, "Using Artificial Intelligence in Video Games," covers the various techniques and genres in which AI has been used, as well as points the way toward possible future uses. Some of the successes and failures of AI implementation are documented; we will look at what has worked in the past and discuss how the past can be leveraged in the future.

AI is treated as an entry point to A-Life; there is a place for both in many gaming genres, but discussing A-Life without AI is bound to leave out some useful techniques. In "Uses for Artificial Life in Video Games" (Chapter 3), you will realize that most games will necessarily encompass a mixture of AI and A-Life. This chapter also shows where A-Life can be implemented in the video game development paradigm, and to what ends. The key is to understand its potential before trying to decide where A-Life can be used.

"The A-Life Programming Paradigm" (Chapter 4) dissects A-Life and shows how the various, existing theories can be applied to implement A-Life in video games. This requires a slight change in the way that behavioral modeling and decision making are approached. Traditional AI prescribes where A-Life will allow behavior to emerge.

Part of this understanding requires that the essential A-Life building blocks are present in the video game. Chapter 5, "Building Blocks," will take you through some key fundamentals relating to implementation of A-Life algorithms coupled with traditional AI. In this way, we can marshal the power that A-Life offers.

Part of this power is brought to bear in Chapter 6, "The Power of Emergent Behavior." In-game characters are given the freedom to have their behavior emerge, rather than be restricted to prescribed patterns of behavior that can be easily spotted and subverted. Emergent behavior is more than just the sum of its parts, however. It can be used at many different levels—from single-entity modeling to entire collections of in-game entities.

These collections can be used for the game proper but also have a role in "Testing with Artificial Life" (Chapter 7). Theoretical and practical advice is offered for testing systems using A-Life techniques. Video games, like all real-time systems, need extensive testing through modeling players' behaviors, and A-Life is an excellent substitute for real players when used appropriately. Development time is not wasted, either. In many cases, A-Life is used to model a real player that can be used as an in-game adversary. Chapter 7 will address how this can be achieved and give some specific examples.

We then extend these testing examples in Chapter 8, "Several A-Life Examples," which dissects some of the actual technology that has been used in games. The hope is that designers can borrow some of these techniques for their own games while discovering how the theories could evolve further.

"Multiplayer AI and A-Life" (Chapter 9) uses the previous material to show how AI and A-Life techniques can be employed in multiplayer gaming development. With the explosion of the Internet and addition of high-speed access, multiplayer and massively multiplayer game universes are rapidly becoming the norm. AI and A-Life can make multiplayer games run better and be easier to maintain, as well as offer some great in-game features. In some cases, their use can also help reduce the need for real humans to perform some of the high-level tasks associated with running a multiplayer game online.

However, many cutting-edge techniques remain in the informal laboratory. Even games such as *Creatures* were not considered mainstream. In Chapter 10, "The Application of A-Life Outside the Lab," we will examine how these novel, processor-sapping AI and A-Life technologies can be designed in, and not just bolting onto games.

After all, the goal of adding AI and A-Life is to make the game better. Exactly what "better" entails depends entirely on your point of view. AI and A-Life can help immersion, making a game more accessible. They can help make better enemies that are more capable, harder to combat, less fallible, more unpredictable, and generally more lifelike. For example, AI can be added to a car to help the driver (player) control it. The player's weapons can have AI built in, and A-Life can be used to manage the minions (entities) controlled by the player. An entire army can be governed by algorithms, making A-Life a formidable tool that the player can use to win the game. The same algorithms can be used to enable an army to learn from the player and adapt itself to the player's style.

There is also ample scope to get the A-Life painfully wrong. It does go wrong, and the results tend toward the disastrous. Part of the problem is that we are trying to give the game as much flexibility as the player has and, in doing so, we cannot be certain that it will not misbehave.

Much like the computer Joshua in David Bischoff's novel *WarGames*, some AI and A-Life applications in video game environments will lose their entities' leashes, enabling them to run a little wild. The danger is that once let off the leash, we cannot get them back under control. We can only trust that the necessary safeguards are in place to marshal deviant behavior.

For example, there is nothing worse for a player than believing the game is against them—or worse, cheating—and that their own trusted helpers (the car, army, or soccer team) are deliberately trying to turn the tide to their own advantage. The result of poor AI is unquestionably simple: the game will fail to convince the player but might feasibly be a commercial success thanks to other aspects of its design or IP.

Using the techniques presented in this book does not necessarily safeguard against getting AI and A-Life wrong on occasion, but it should help you understand where the pitfalls are, learn how to build in those safeguards, and develop correct AI and A-Life, as well as be in a position to correct any errors. After all, "intelligent," self-correcting code cannot be far behind, can it?

This page intentionally left blank

CHAPTER

1

INTRODUCTION

In This Chapter

- Defining Artificial Intelligence
- Knowledge Management
- AI Put Simply
- Defining Artificial Life
- Top-Down versus Bottom-Up Intelligence

This chapter will broadly introduce the concepts of artificial intelligence and artificial life, in particular, as applied to video games. There are some general concepts to be aware of before trying to tackle the rest of the book, and we cover these in this opening chapter.

AI and A-Life can be embodied by a quote from David Bischoff's novel *WarGames*, in which one of the characters attempts to explain how AI leads to a decision-making process.

"If you put your foot in the fire, and it got burned, would you put your hand in?"

It illustrates the point that, in a system of connected parts, such as the human body, we can use experience in one area to predict the outcome of a similar event in another area. In the case of the character in the novel, a connection between two specific games is being explained—neither of which can be won.

The inevitable outcome of one is stalemate, and the logical conclusion that Bischoff is trying to get across is that, in a similar game with higher stakes, stalemate is still inevitable. Another choice quote from the same book reads:

"The only winning move is not to play."

The context of these two quotes gives us the basis for both AI and A-Life. On the one hand, we use AI in games to direct behavior, while on the other hand, sometimes A-Life can produce solutions that AI, by itself, cannot achieve.

Traditional AI systems (state machines, planning systems, and so on) direct behavior in an explicit way. It is only when we augment these static systems with holistic and heuristic ones that A-Life begins to replace pure AI. This results in emergent solutions that often cannot be achieved with pure AI techniques as used in video games currently on the market and in development.

It is possible to avoid A-Life completely and tie many AI systems together, as can be seen in many "modern" video games, but the result is a limited emergence, which can be every bit as predictable, over time, as traditional AI solutions.

It is only when we look at each in turn that we can decide which one is most appropriate for a given game universe. This chapter and the next two should help you begin to make "intelligent" choices for a game project you might have in mind.

Defining Artificial Intelligence

For our purposes, a loose, informal definition of artificial intelligence will suffice. After all, this book is a study of the application of AI and A-Life in video game design, development, and testing—not a discourse on AI and A-Life as scientific fields. However, remember that they are just that—scientific concepts that imply two things:

- They are incomplete, and
- Advances are still being made.

In other words, as this book is being written, researchers are uncovering new AI techniques and A-Life paradigms. The rudiments of AI may have been worked out and documented, but these are surely just the tip of the iceberg. A-Life is still a field in its infancy, and there are sure to be leaps and bounds made as machines become more capable.

Put all this together, and we begin to realize that not only is a formal scientific definition unnecessary for our purposes, it is also incomplete. Hence, an informal definition will be more than sufficient; we could label it an *applied* definition that will take into account how AI will be used, rather than being an all-encompassing theoretical explanation of what AI is.

Artificial Intelligence as a Reasoning System

One aspect of AI is that it makes decisions based on information from existing models. It is a decision-making mechanism that might result in action or inaction and will be based on the presence or lack of information of some kind.

So, we could look at an AI module as something that receives an input, analyzes that input, and delivers an output. The analysis of the input is the "intelligent" aspect of the system. Rather than delivering a prescribed output blindly, without regard to any input, we create a model that attempts to map one to the other (output to input) within the confines of the problem domain.

In simple video game terms, this is like creating a game in which the player shoots upward at marauding aliens. The aliens can also bomb back—and in one of two modes, via a:

- Prescribed pattern, or
- In-game knowledge-driven decision architectures.

In the prescribed pattern mode, the aliens drop bombs on the player at regular intervals—say, every 10 seconds—regardless of the player's relative location. The mode using in-game knowledge-driven decision making will drop a bomb at a regular interval *if* the player is about to move into a position where the bomb has a chance of hitting him.

The latter solution uses in-game information and applies it to supply a more natural solution.

The model for deciding whether a bomb should be dropped or not has to take many variables into consideration—the positions of other aliens, the player, other bombs, closeness of the player, self-preservation, and so on. The more sophisticated the model, the more "intelligent" we say that it is.

So, in this case, AI uses information in a specific model to arrive at a decision. The subsequent action results from the decision-making process that uses observation of the state of play as the input information. The model is prescribed, but flexibility is allowed in the decision-making process.

If we want to model a different behavioral process, we need to create a different model with its own input information and output actions. We can implement a

movement model, for example, that manages the position of the alien in relation to other aliens, the player's craft, and any other bombs/bullets in play.

In essence, the model creation is a shortcut in the AI process. We substitute a solution (model) of our own, rather than provide the system with the ability to work out a solution for itself—a system known as "universal AI," where information (data) is acquired and applied to different situations. Universal AI, therefore, is not within our grasp for applications such as video games, if at all. But certainly, with the processing power available in current equipment, intelligent models can be used to enable the system to exhibit intelligent behavior.

These models are therefore prescribed, and the final behavior will be deterministic, and therefore somewhat predictable, even if it presents a challenge to the player. In our aliens example, the player will quickly recognize the model being used and adapt his own model of "dive and fire" to counteract it while avoiding any bombs being dropped.

The (human) player is equipped with problem-solving capabilities that enable him to apply previous knowledge to new situations and come up with simple solutions.

A deterministic model, which is based on rules, is created by one person (the game designer) and is solved by another person. In other words, the AI implementation is the product of one mental model and can therefore be solved by another, once the key is found.

In this kind of approach to AI, people create models to mimic intelligent (in this case, human) behavior by analyzing a specific problem. Just as we took the alien-bombing model through several steps to arrive at a solution (drop a bomb when over the player), video game designers and programmers will similarly create models (algorithms) that process system information in order to solve in-game problems.

Each model is represented by an algorithm in the resulting AI implementation. Programmers may talk in terms of an algorithm that represents the model that has been arrived at as a description of behavior that should manifest itself in the game.

The transition of the human in the game (the designer) to the alien dropping bombs is what results in the model that is used as a template for the alien behavior. This is an example of solving an in-game problem (albeit a simple one) using human behavior that is mapped in to the game space. This can be extended by applying universal AI.

One of the features of universal AI that can make the behavior more complex is the ability to adapt those models. If the game is intelligent enough to notice that the player has found a way to counteract the model in use and change its behavior accordingly, we can call this "real intelligence." A model that represents this new behavior might be dubbed "real AI."

Our first model dropped a bomb only when the player was directly below the alien; but this, as was noted, is easily counteracted. If, though, the alien notes that the closer it is to the player, the greater the chance that the player is hit, it might adapt its reactive model to be more proactive. This might lead the alien to move closer to the player when the player's craft approaches the line of fire. To arrive at this new behavioral model, the alien needs to know that it can move downward as

well as from side to side. It also needs some information regarding range as well as location—if the player's craft is directly below or if it is moving toward the alien or away from it.

This information needs to be gathered together and a decision made—move left, right, down, fire, and so on—but these calculations are not strictly binary. The original model was based on a simple yes/no response. Is the player directly below? If yes, then fire. If not, then do not fire.

We can build in adaptability that considers data that is not based on binary decisions, and we can create algorithms to manage the models being using to exhibit intelligent behavior. Although it might look like universal AI, this will result in models and behaviors that are still applicable only in one problem domain—the video game.

The result is that the algorithms that represent the models become part of a layered system in which new algorithms must be found to manage the application of these lower-level behavioral abstractions. It becomes a hierarchical system that mimics the way that behavior has been observed in real organisms.

So, AI in this case will be the application of models to a given problem domain. To a certain extent it is still predictable, and the player will discover ways to subvert the so-called intelligent behavior of the aliens, but it is as close as we can often get to in-game intelligence.

As we shall see later in this book, A-Life might provide a quick answer to this problem. Some algorithms that we can devise will allow for this kind of model adaptability to present a more natural, less predictable, and sometimes more intelligent model that the player will find harder to combat or more natural to interact with.

Appearance versus Actual Intelligence

Whether adaptable or not, some models are so sophisticated that it becomes almost impossible to tell whether the object on the other side of the interface is intelligent or not. The Turing Test, devised by Alan Turing in 1950, is an experiment to test the capability of a machine to exhibit intelligence. One embodiment of the Turing Test requires that a series of judges interrogate, via a keyboard and screen, another party, without knowing if the other party is human or machine.

It is the job of each judge to try and work out, within a specific subject area, whether the other party is human or a program designed to act like a human. If the judges are fooled, then the computer is said to have passed the Turing Test; that is, the judges cannot tell for sure whether the other party was a machine or a human.

However, take that system and put it into a different domain, and it will quickly become apparent that it is not intelligent. The appearance of intelligence is often enough for us to decide that the behavior exhibited is intelligent. We cannot tell the difference; so we happily accept that the being is possibly human.

One such case in game design mythology is related by Richard Garriott, manager of the *Ultima Online* project. He was walking around the newly created *Ultima* world in his guise as Lord British, when he came across an NPC (non-player character) going about its prescribed business.

For a few moments, Garriott engaged the NPC in conversation and could not tell from the first few questions whether it was human or machine. The fact that they were both characters talking through a text interface, and could both have been human, helped create the illusion that the NPC was really another player or another member of the design team. It was only the NPC's inability to relate exactly what part of *Ultima* it was working on that led Richard Garriott to conclude he was conversing with an algorithm rather than a human. However, he was fooled, if only for a moment.

Similar implementations of conversation algorithms, such as the famous Eliza, just follow prescribed patterns of interaction. They take input from the user (player), check for key words that they can respond to or rearrange the user's input, and provide output. Eliza, for those readers who have not come across this piece of AI software, was a chatbot modeled after a therapist. Through a series of questions and clever reversal of the human players' responses, it was capable of carrying out a limited but reasonably authentic conversation. It failed, however, in the Turing Test, to convince the judges that it was not a machine because the AI mechanisms implemented followed a static model that could not deal with new situations and rigidly stuck to the preprogrammed game plan devised by its human creator.

Certain prepared phrases can also be used as an attempt to guide the user through the conversation in an intelligent way. For example, if Eliza encounters a phrase that it does not understand, it is likely to counter with a phrase that forestalls decision-making and solicits additional, perhaps understandable information this time:

"Can you tell me why you …?"

If the answer does not help, then Eliza might respond with:

"Please go on."

The mixture of phrase-spinning and prompt statements, combined with an ability to parse the input into the appropriate parts of speech, gives the appearance of intelligence. While it is unlikely that Eliza would pass the Turing Test, for video game use and within a restricted environment, this level of intelligence is more than suitable for providing a sense of immersion for a willing player.

The same kinds of abstract theoretical behavioral models can have multiple uses, as we shall see later on. An Eliza-style speech-processing algorithm is an implementation of a challenge-response model. This model can be adapted to, say, fighting games, where question and answer become attack and defense, combined with renewed attacks based on some of the player's own moves.

The algorithms will be different in each case, because they address very different problem spaces, but they share some commonalities in the abstract model that the algorithms seek to represent in concrete terms. In other words, starting with an abstract model, multiple uses can be foreseen, in multiple problem spaces. The

reuse of the theory and models can only help a game studio create better AI and A-Life for future titles.

As in the Turing Test, the goal is to fool all the people all of the time. Video game players are easy marks—they want to believe. Only the rudiments of intelligence are sometimes required for them to feel as if they really are up against a clever opponent.

Video game designers must remember at all times that the limits on hardware mean that we need to leverage as little intelligence as we can get away with. In other words, the amount of intelligence that we design into a game has to be directly related to the game genre. The amount of processing power available to AI routines is dependent on the needs of other game aspects, like graphics, sound, and game universe management routines.

However, note that some of these can be augmented by the use of AI and A-Life and use less processing power in the bargain. Actually, a mixture of AI and A-Life will probably be needed—something we will look into as we progress.

The reapplication of AI models to make adaptable musical modes that use feedback from the current game status representation and player's (or enemies') movement to alter the music is one example. Another might be to use A-Life techniques to "breed" visual elements that look natural and can be processed at a reduction in processing power.

In theory, at least, it ought to be possible to take an innovative shading engine like that used to render high polygon visual models from low polygon representations (as in the CryEngine) and augment it to provide consistently natural results. The basic low resolution polygon model becomes just a template onto which the features of in-game characters can be rendered, and AI/A-Life techniques used to create the actual rendering deviations from some standard model.

To recap, our first tentative definition of intelligence is the application of knowledge within a system, using a model to arrive at decisions that exhibit intelligent behavior. This is generally good enough for video game applications. The Eliza program and Turing Test operate on the principle of the application of knowledge within a system. This knowledge is acquired through interaction (the Eliza "conversation") or from part of a database of stored knowledge.

The game 20 Questions is another example of applying acquired knowledge through interaction and stored knowledge. Questions are asked, with the answers narrowing down the options, eventually revealing the target object. It is a simple enough game for a human, as long as we know what questions to ask. For a machine, we can build a model that helps in asking the right questions and store (temporarily) enough information to enable the computer to arrive at a possible answer. Both humans and machines stand a good chance of getting a right answer—and both can also make mistakes.

Knowledge in a video game can take many forms. There is the game universe and the objects contained within it, the game rules, the actions of other beings, and the value of objects within the game universe. Each being in the game universe has to be able to work with that knowledge in order to act and react within the confines of the

game rules. Storing and processing of information in this way can be termed "knowledge management." It is arguably the cornerstone of most AI and A-Life routines.

Knowledge Management

In the context of video game AI, knowledge management is the:

- Storage,
- Access, and
- Application of information.

This information is acquired through interaction or supplied from outside the system. Each area of the system that needs to store, access, and apply information will be provided with some basic starting knowledge, but may also need to add to that information store and adapt the knowledge base accordingly.

Additional information can be acquired through sensory systems, and I tend to use the term "interaction" in one of two ways: in-game interaction or interaction with the gaming system via sensory means (a controller, for example). This is because I tend not to make the separation between the human controlling an in-game extension of himself and that in-game avatar.

Although it should be reasonably clear from the context which of the two interaction modes I am addressing, where ambiguity could exist, it shall be resolved by talking in terms of in-game interaction and sensory interaction.

The first stage in the process is to know what information should be stored—that is, what knowledge needs to be applied in order to function intelligently. Having decided what that information is, we then need somewhere to store the knowledge. Entities within the system will store knowledge units in different ways and for different reasons.

Solving this processing and storage of information problem can be done in a variety of ways. Generic neural networks contain interconnected nodes that can be trained to react to input stimuli and change their internal state so that new behaviors can be induced by varying the input data.

This capability means that a neural network has the capacity to both process information and store it according to the requirements of the system. At the beginning of this chapter, the modeling approach to AI took a shortcut in behavioral terms. Developing a neural network tailored to a specific information domain provides a useful shortcut to creating a knowledge management system for use with video game AI.

Neural networks are modeled on living beings. A human, for example, is able to use a neural network to solve multiple problems, store many different pieces of information, and process knowledge in various ways. To mimic this in a constrained environment such as a video game would not be feasible.

This book concerns itself with *computational* neural networks, which are modeled on our understanding of *biological* neural networks. In other words, a computational neural network is an abstraction, a model, or a simulation of what we believe goes on inside a human brain. Limits on computing power and the inherent complexity of

the human brain (which contains extensions such as emotion) mean that we cannot really get close in simulation terms to the complexity of the human brain, but we can, at least, mimic some of its reasoning power in a computational neural network.

In the same way that we created models such as Eliza to imitate intelligent behavior, we can also create knowledge models to mimic the storage and application of information, which will have lower overhead than universal knowledge management routines. An application of an AI technique such as neural networks can be used to manage the knowledge model and its application, but we do not have the resources to create a universal neural network.

A neural network that works in this way can be viewed as a collection of objects (neurons) that communicate (signals) to store or process information. That information is introduced to the system through sensory signals (interfaces) and exits as signals to create actions. Some input does not result in action, but creates an "almost path" through the system.

Actions are the sum of the signals present in the system and are influenced by each signal that is processed. These actions will be carried out by external interfaces between the network and the virtual system that represents the game. So, each action is influenced by the signals, but the signals do not directly create actions in themselves.

The "almost path" only becomes a "definite path" once a threshold has been reached—that is, the tipping point for the final link in the chain has been achieved. In other words, the gradual buildup of knowledge leads to an action at a certain point, whereas a single sensory signal, or input, does not.

So, our design choice is whether to treat knowledge as an information store or to treat knowledge as part of the AI itself, and as a collection of interacting objects. Prescribed information (knowledge) can be represented as a prepared neural network (learned in advance) or as an information store with no possibility of applying itself (it needs algorithms to access and use the knowledge stored within).

Part of this knowledge can be built into the model of the game universe. Each unit can have the possibility to know certain things about itself and the effect it might have on the environment. A heavy object, for example, might "know" that it will crush (destroy) things that it is placed on.

An object that generates light might know that it can illuminate the corner of a room. This approach was taken in *The Sims*, where each object was given properties that related to the effect that they have on the immediate game universe. There were objects, for example, that would incite content or discontent in Sims that encountered them.

It is the combination of effects—on the one hand, in *The Sims*, the reflection of inherent information simulating a transfer of knowledge, and the acquisition and processing of information in Eliza on the other—that produces the illusion of rudimentary intelligence. Reflection of inherent information and acquisition of new information are two techniques for mimicking knowledge storage, processing, and transferal, but both give the impression of intelligence.

A step up from this kind of AI is the computational neural network that is at the heart of the AI processes present in the god game *Creatures*. These are called Norns,

and they exist in the game world as autonomous entities capable of both simulated biochemistry and emotions, as well as refined thought patterns.

When a Norn encounters an object that makes it feel discontent, it learns this piece of information through a neural network-style reinforcement. Subsequently, the Norn might avoid those objects or the places where those objects are in order to avoid discontent, because it has been programmed to exhibit this behavior. A human, then, could marshal these two observed phenomena and use them to manipulate the Norns in the environment. This is, of course, the goal of the game—manipulate the environment to keep the inhabitants (be they Sims or Norns) happy.

So, along with the means to store the information, we need processes that can act on the knowledge before it can be of any use. Simple assimilation of knowledge serves no real purpose within video game design and development. For example, we might have built-in processes that offer information to in-game objects through interfaces between objects, as well as those that offer information to the in-game objects on the universe and interfaces themselves.

We humans actually learn much of what we know by observing others; hence the "allowable" trick that lets objects tell actors how they can be used. The terminology for this trick is *affordances*. These affordances are permitted because, in the context of a video game, there are not enough resources to provide learning for every possible action of every object within the game universe. So information is provided that should be obviously conveyed through the interface between the object and the actor (NPC or player). When an actor comes across the object, he knows how to interact with it because the object tells him. This is analogous to the player having read instructions on how to use a given object in the game.

We can apply this approach to virtually any kind of game object and allow an applied intelligence to shine through. Knowledge is stored through properties and acted upon via processes that transfer information, such as to affect the behavior of objects. Will Wright, creator of *The Sims* and *SimCity* (among others) and the chief game designer at Maxis, calls this method "smart terrain."

Internal information, such as pathfinding data, needs other, different ways to work on it. Pathfinding is one of those processes that often combines a sense of targets and goal-setting with a latent value or cost associated with each stage in achieving goals. This algorithm is part of something called the A* algorithm, which is a commonly used pathfinding algorithm that we will discuss in more detail later in the book. For example, when an entity calculates the best path through the environment, it will record its progress against the target location (spatial knowledge) and the cost of each available option, thus allowing it to choose the "best" option—the shortest and/or cheapest in terms of environmental variables. These goals and internal states are more pieces of information that need to be managed in the system. Processes that contain rudimentary intelligence will be required to make sense of these. Again, these processes can be represented in terms of classic AI paradigms—the A* algorithm for pathfinding or a neural network for pattern recognition—but the implementations will be different for each situation.

We need to be able to translate the effect of the processes on the problem domain; in this case—the state of the video game universe—and each process must have an effect, or it is a waste of processor resources. Available cycles will always be in short supply.

The translation of process into action will alter information within the problem domain, perhaps embodied as player-centric changes through a variety of means, but the prime purpose is to change information stored within the system. As a video game designer/programmer, remember that a process will cause something to be altered—information local to the actor, an object in the game universe, or the game universe itself. The actor may become hurt (reduction in health), it might cause another actor to be hurt, or it might damage/augment the game universe itself.

Each of these actions is just a state change in a piece of information, representing the state of the game world at a given moment in time; we can model these interactions as exchanges managed by processes created specifically to do so. In other words, when the player sees his onscreen character exchange goods for money, our underlying process simply deducts money from his purse and adds the items to the player-character's inventory.

The embodiment of an action (the discrete movement on the screen that reflects the movement of the agent within the system) will likely be a predefined animation, but in actuality, it is very simple. For example, a pathfinding algorithm may take many steps to work out the optimal route, and the player only sees a single route taken by the opponent, but the information inside the local actor—such as speed, direction, and so on—changes over time.

Once the path has been generated, the agent will move through the game space following that path. That movement from one point to another in game space can be represented inside the actor as a collection of directional information, some of which might dictate the visual representation of that action or movement.

Once an action has been taken, we need some way to observe the effect that has been made on the problem domain. This can serve many different purposes in AI programming—from learning through reinforcement to triggering other processes within the model used to create the behavior.

The actor (or other actors within range) should be able to observe this effect, which then becomes part of the actor's internal knowledge. Whether or not this new information has an effect on future actions is what separates a purely reactive stimulus-response algorithm from an AI algorithm. The AI algorithm, while somewhat prescribed, actually learns from experience.

For example, the effect on the environment could be another state-altering process, and this, in turn, could have some effect on either the environment or the actor. Chains of effects can be built up from simple actions—either *micro chains* that have small behavioral effects or *macro chains* in which pieces of the game universe are altered by the chain of effects and reach out over several time slices in the game session.

Information Feedback

Part of an agent based around AI algorithms' ability to learn is reinforcement through information feedback. Each model that is designed to make a decision based on incoming data should incorporate some form of feedback management. A neural network, for example, can be created with weighted feedback loops in which incoming state-change data causes a particular neuron to fire more (or less) frequently, depending on the weight of the data.

If this sounds somewhat abstract, imagine a neural network in which inhibitors are attached to a decision process. All that an inhibitor does is reduce the strength with which a neuron will fire. Sensory information can be fed back to the inhibitor to increase or decrease its strength relative to other neurons in the system. For example, three neurons have action outputs to move forward, left, or right. An inhibitor is attached to the forward neuron, with its input coming from a sensor in front of it. In that way, forward movement is restricted based on the feedback from that sensor. In other words, when choosing between forward, left, or right movement, the weight of the forward decision would be governed by whether or not an object is in front of the actor.

This is an example of reactive feedback. It is not knowledge based. However, remember that a neural network can learn, and clearly the previous example does not strictly use a neural network at its most powerful.

We could call it "instinct" as opposed to "learning"—simple reaction instead of being able to determine the likely outcome of a new action from experience. In order for the system to learn, a new inhibitor must be added (an activator) that would choose between left or right, depending on inputs from range, speed, and bump sensors.

Upon the instantiation of a specific actor, the activator is set to zero. The system has no preference as to whether it turns or not when it approaches a wall. Now, we could build a wall-avoidance routine to model reasonably intelligent behavior and implement that in the actor, which would be the prescribed behavioral model.

However, we could also place the aforementioned neurons and activator/inhibitors in the system and let the neural network adapt itself. In essence, we build a trained network that causes left or right movement based on the prediction of a bump, depending on speed and range. The situation that the network needs to build up into is one where continually bumping the sensor on the left side leads to a tipping point where the action is changed and the actor decides to move to the right instead of the left. If this action is then reinforced, we would expect that future bumps on the left side would lead to immediate movement to the right.

The system has learned that if it gets bumped on the left, it should move to the right, and the role of the activators and inhibitors in the system is to make sure that this decision takes place on a consistent basis, based on experience.

We could add to the collection of choices by allowing the system some freedom to choose between left and right movement (or reduce forward movement), depending on range input from sensors (connected to inhibitors) on the left and right sides of the actor.

Over time, a reasonably sensible avoidance network would be created that is able to deal with many different object-strewn playing areas. The network has learned how to best avoid objects based on anticipated damage, speed, range, and direction in much the same way that a living being would build up a similar set of choices—in other words, artificial life.

A-Life is the subject of the next section and most of the rest of this book. However, the theory introduced here needs to be summarized to ensure that key points are understood.

AI Put Simply

As was mentioned, this book is not a scientific study of artificial intelligence; it is a guide to implementing enough AI to imbue a video game with appropriate behavior. So, the field of artificial intelligence has been broken down into those parts that are relevant to video games and those that are not.

Put simply, we need some building blocks, but we do not intend to create true universal AI routines that can adapt to any circumstance. We are limited by processor power and the need to create a game with AI as a very important part of the whole; often, AI gets extremely short shrift, but the application of some simple principles can instill AI into the game without unduly burdening the system or development team.

When this concept is understood, as well as the building blocks that have been presented, then it is time to move on to A-Life, the focal subject matter of this book. Chapter 2 will flesh out how AI and the algorithms employed are currently used in video games. For now, our discussion will remain non-specific.

Thus far, our approach to AI has centered on rules-based modeling, albeit a step above the simple reactive stimulus-response. We derive algorithms based on observation of intelligent behavior, often our own. Even a rudimentary pathfinding algorithm is based on our own trial-and-error experience and a look-ahead to solve, for example, a simple paper and pen maze. The result is that the pathfinding algorithms are not themselves intelligent, but they have been created by a programmer, who is.

Each decision can be backtracked (in memory) and a new decision made as a result, but the behavior is governed by applied information within a rules-based system.

These rules can lead to strict binary yes/no decisions, such as in the 20 Questions game. Others can be weighted, which adds a level of sophistication to the questions that can be asked. For example, in 20 Questions, this equates to selecting one yes/no type from a collection of questions, depending on the weight of evidence that is building up from previous answers.

We call this "fuzzy logic"—choosing between possibilities based on a weighted system. It is at the core of most neural network implementations described in this book.

Neural networks generally in the field of AI may be based on fuzzy or crisp principles, where crisp decision-making mechanisms make explicit decisions that do not entail any of the information regarding the extent of those decisions, as in a weighted system.

Another kind of architecture consists of finite state machines in which each component has a collection of possible states and stimuli that can move it from one state to another. For example, smart terrain offers the possibility to move an actor's state from one state to another as a result of the interaction between the terrain and the in-game actor. This state change is then reflected through the appearance of the actor, the terrain, or both.

A Sim in *The Sims* universe knows how to turn on a TV because the TV object broadcasts that information. The interaction between the Sim and the TV causes a state change from "off" to "on." The "on" state then causes a state change in the Sim (depending on whether or not the Sim likes TV).

This feedback from the environment, however it is modeled, is vital to using AI in video games, whether in the slightly novel application presented here or in the more serious conquest management AI in strategy games. AI enables the system to constantly try to alter its behavior (for the better) based on information gleaned from the environment. That information can be broken down into two types: instinct, in which algorithms model simple involuntary behavior; and learning, which comprises more complex, weighted, decision-making networks akin to neural networks that learn how to make the best decision based on complex input signals.

Learning implies knowledge, which in turn means that the required information in the video game system (to be used by the actor under development) be stored and processed. Persistent information (knowledge) is a particular problem because, in video game terms, it needs to be saved and restored between play sessions. That is, an adaptive AI system is useless (except in certain circumstances) if it becomes dumb the next time gameplay is resumed. If a play session has taught the underlying AI some of the player's tricks, the next play session should restore that knowledge.

All of these issues hinge on inter-object communication within the problem domain—in this case, a video game. If game developers do not provide for these features from the ground up, then adding AI will be a very difficult task. Hence, this illustrates the overriding need to get AI routines defined at the start of a game development cycle—and if not implemented, then at least acknowledged.

Defining Artificial Life

Much of the rest of the book will discuss actual uses for AI and A-Life in video games, and an emphasis will be put on augmenting AI with A-Life. Now it is worth looking into how we might define A-Life for that purpose. Various scientific studies on A-Life are in progress in this evolving field that try to differentiate between:

- Strong A-Life (truly alive, synthesized life), and
- Weak A-Life (not truly alive, but modeling lifelike behavior).

In this book, we consider "weak" A-Life for use in video games—in other words, modeling the behavior of life rather than trying to create real life in an artificial

form. Therefore, we can allow ourselves a reasonably loose definition of A-Life, as was done with our definition of AI.

A-Life is a young discipline and still under development. Some programmers think of A-Life as AI deployed within a lifelike system, but others will have their own definitions. The A-Life domain is, however, changing all the time and is subject to well-researched and well-understood key theories. It is an applied science that does not deviate from the two basic views: that strong A-Life can make synthetic life possible and that weak A-Life only models lifelike behavior. Both views are the subjects of well-founded theories, and whether we seek to create life in a video game (like *Creatures*) or just exploit life-like behavioral models (like *The Sims*), we need to define how it will be done.

One mechanism at the forefront of A-Life theory involves including evolution through mutation and natural selection. This is an application of simple Darwinian science—only the strong survive long enough to pass on their genes, but there is always the possibility that offspring will be weaker, stronger, or just different.

In video game terms, we can apply this to behavior models as well as entire beings that are governed by simple AI routines. There are several approaches to implementing evolution, which are defined under the umbrella term "evolutionary computation." The A-Life discussed in this book falls into this category; AI theory is used as building blocks. Using evolutionary computation can be as benign as refining specific techniques during a play session, or it can be as active as creating and destroying life forms while culling out only those with specific behavioral traits.

The two main types of evolutionary computation that we shall look at are "genetic algorithms" and "genetic programming." They are similar and related yet subtly different approaches to breeding solutions and behavioral models. Genetic algorithms (GAs) take a best-fit solution to a specific problem, whereas genetic programming (GP) modifies the solution to fit the problem by changing the actual algorithm.

Both use algorithms borrowed from our observation of life—mutation, inheritance, and genetic crossover—combined with some selection criteria. Those selection criteria may differ according to what the designer is trying to achieve, but they are commonly linked to the success of the GA/GP implementation with respect to a given problem domain.

In our case, the algorithms are derived from classic AI, as was mentioned. Lifelike behavior is being synthesized within a video game using AI building blocks and A-Life theory to create more natural in-game behavior models.

Remember, strong A-Lifers believe that something truly alive can be created, and weak A-Lifers believe that the result is not truly alive. Our attempts will fall into the latter category.

The key selling point that A-Life has over strict AI is that the result is more than just a few AI routines put together. If that were all, then there would be no reason to move outside the reasonably well-understood domain of AI. An A-Life approach can be seen as the pinnacle of achievement for AI—emergent behavior. This is due to its nonprescriptive, bottom-up approach, which differs from an AI, top-down solution. In other words, we do not *just* make rules derived from human observation and try to

fit a problem around them; we build a system that should be capable of finding a solution within a given problem domain. It might not be the best solution, but it will be more natural, less predictable, and possibly even more efficient.

One final note: When using A-Life in video games, we do not generally seek a discrete solution to a specific problem. AI provides that by itself. Instead, a new layer of complexity is introduced that delivers additional value to the player for very little additional cost.

To illustrate this, consider an AI pathfinding algorithm. Assume that it will, via a series of binary decisions, find the best path through a terrain, based on the relative success of each decision with respect to possible subsequent decisions. (We will come back to how these are implemented later on.) Now, add to that a GA whose purpose is to "try" various paths and come up with one that is more or less doable by combining parents to produce offspring. In this way, we breed a path solution that is unpredictable and hopefully never the same, but always logical at each step.

GP can be used to modify the evaluation of what constitutes a best path, based on the terrain that would have to be negotiated, the position of the actor, and other key factors. Algorithms that do not fit within the selected criteria would be discarded, and variations of the others would be bred. In this way, we aim to produce behavior that is more lifelike. This adds value to the game in the way of a richer playing experience. We have taken AI, modified it with some standard, easy-to-implement GA and GP techniques, and the result should be closer to the goal of becoming artificially lifelike.

In the sense that life and beings in life are less deterministic, more varied in their responses (until conditioned), and less predictable, their behavior becomes a little closer to being lifelike and a little further away from feeling entirely artificial. Truly lifelike behavior might need the addition of a few more ingredients, but we are moving in the right direction.

Modeling Through Observation

Much of video game–applied A-Life is based on models that are derived from observation, as was described. In fact, the video game itself is based on this premise; only the most abstract of games (such as *Rez*) do not attempt to model actual/perceived, lifelike behaviors.

(*Rez* is played as a representation of a computer network and is entirely abstract in premise and representation. The graphics are the pinnacle of abstraction, some barely recognizable as abstractions of cities and valleys, and the backing music is industrial hardcore, similarly abstracted from what purists might call real music.)

However, unlike our use of applied AI, the A-Life application will not attempt to create an over-reaching model that is designed to capture all of the possible chains of events that might stem from a decision process. Rather, it is a less-prescribed approach that provides a framework for the building blocks (small AI routines) of A-Life, which are combined to create a larger, more complex model.

In other words, we want to enable complex behavior to grow from very simple behaviors. Think of a swarm of insects or a flock of birds. Each one takes cues from its neighbors, either based on following a "leader" (like ants) or based on the approximate distance, speed, and direction of the neighbors (like fish and birds). This behavior has been modeled by Craig Reynolds in creating flocks of "boids" that exhibit emergent behaviors by each following a set of discrete rules.

The swarm behavior is based on observing changes in the actor's immediate environment—as with AI. Each individual actor tweaks its own behavior based on a variety of inputs, and each input is weighted according to specific experiences. For example, one actor could decide that it is more important to avoid an obstacle than remain a set distance from its neighbors and risk injury. Another might decide that the injury is a worthwhile sacrifice and is willing to accept the consequences.

Where this modeling approach differs from classic AI is that changes in the environment can happen at much lower levels (the building blocks) in order to produce a larger, more complex, result. The behavior is allowed to emerge, rather than being prescribed by a static algorithm, even when that algorithm could be changed by events.

So, there is no over-reaching model to guide a single action, except where that level of guidance is necessary. This is one reason that A-Life cannot exist in isolation in the vast majority of video game implementations. We still need to direct the natural evolution of the solution—in other words, A-Life guided by AI, and built on AI foundations.

The model is allowed to evolve from the observed behavior of specific automata (abstract machines) whose behavior is either guided by themselves (based on external or internal stimuli) or guided by controlling forces. These forces also need to be intelligent enough to cope with changing circumstances and resort to strict, prescribed limitations only when the evolved behavior oversteps the allowed thresholds of freedom.

Our new model, therefore, evolves from a soup of *possible* entity routines, all trying to solve particular problems within their own problem domains. These entities are either testing possible solutions (GA) or modifying the algorithm that leads to them (GP), or even performing a combination of the two.

In essence, known processes are being synthesized in order to provide building blocks for the natural development of new processes. These known processes are derived from observation of real-world (or presumed world) behavior and are employed by other clever algorithms within the game universe.

These A-Life management routines may themselves be AI or A-Life based, or they may define strict limits (boundaries) that restrict the allowable freedom given to the A-Life/AI implementation. Luckily, this all sounds harder than it is to implement.

Calculated A-Life

The first kind of A-Life implementation that we will consider is driven by calculations used to derive the behavior of the system. The result of these calculations feeds

the A-Life and AI models, which then become an embellished embodiment of those calculations.

A-Life in this case is used to enhance the visibility of calculated results that reflect the true measure of the player's success in the video game universe. The behavior itself is not causal to the outcome; it just provides an intelligent (natural) interface to the information in order to guide the player.

A game such as *SimCity*, for example, can be modeled as a series of calculations, with the result of each feeding into others, and the behavior of the system derived from that. In other words, the original *SimCity* had three key real estate indicators—Residential, Commercial, and Industrial (otherwise known as the RCI indicators)—which fed back the status of the population's needs to the player. Calculations were performed at each decision point (in time) for each square in the game universe to determine the status of that square. The embodiment of these calculations was in the interface—growing shopping malls, decline or increase in demand in the RCI indicators, traffic on the roads, and so forth.

Each calculation of population size, needs, and satisfaction is fed into the universe, which comes alive as a result. However, the game universe behavior is also based on calculations, not the synthesized decision-making of individuals—either the in-game actors (Sims) or the squares depicting the status of the grid location.

Part of the reason was probably due to issues with machine resources at the time the original game was created. Subsequent versions have probably worked in some of the more A-Life–style emergent behavior, which is based on algorithms rather than calculations.

One of the *SimCity 4* innovations was the ability to follow a Sim around as it went about its business and be kept abreast of its overall status. The Sim can report any sickness, its employment status, and an overall opinion of the neighborhood, allowing the player to make judgments.

Whether this is A-Life or just clever feedback in the gaming environment is moot. (Do these opinions shape the universe, or are they merely a product of it?) But it is an indication that the series was becoming more lifelike with each new version. We can quite easily envisage a game in which each Sim is polled and its opinion then used to feed the behavior of the system.

The Sims series of games appears to operate similarly. From the system's observed behavior, there is at least a partial feedback loop. However, it could still be considered calculation-based A-Life; there is none of the innate synthesis of life that exists in games such as *Creatures*.

In *The Sims*, rules and roles are applied to observed individual parts of the system, modeling each one as an entity, rather than as a collection of smaller units. The game universe becomes the A-Life entity, whose behavior is governed by the actions of the actors within it.

So we still need to model a system. The fact that A-Life is to a certain extent being allowed to manifest itself is simply a representational issue, rather than an actual part of the underlying game model. Each part will have rules that govern its behavior and interaction with other parts of the same system. The calculations that go

on behind the scenes are the controlling/managing, routines that dictate the parameters in which A-Life is allowed to evolve.

The rules and roles, in turn, synthesize the action and reaction parts of the system, based on a predefined model. That synthesis, derived from calculations, mimics life, and the player is encouraged to treat it as such.

This is as true for Sim-style entities as it is for racing vehicles, football players, or gun-toting swarms of aliens. The fact that there are many calculations and AI routines driving their *general* behavior does not detract from the embodiment of their lifelike behavior toward the player. They are calculation-based A-Life representations of a system acting and reacting based on a series of models.

So which parts of a given system need to be modeled? The choices will be dictated by many different variables. More often than not, the defining issues are quite concrete:

- Processor (and/or memory) resources,
- Development (and/or design) resources, and
- Testing time available.

It is unlikely that the game designer/developer lacks the vision needed to create the AI/A-Life that adds value to the game. Usually, it is the real-world pressures of delivering a high-quality game to market within a deadline that dictates the amount of A-Life that can be incorporated, and the level to which it is allowed to exist.

The disadvantage with A-Life is that is easy to go too deep into the A-Life paradigm in an effort to allow it as much freedom as possible. If we wish, the whole game universe can be micromanaged, just so we can watch it evolve. But remember, our goal is not a scientific experiment. We are creating video games, and with the possible exception of *Creatures*-style games, we do not want the AI or A-Life to soak up all of the resources. This means that we have to be fairly thrifty with processor and development time requirements.

As we shall see, however, A-Life does have other uses besides being deployed within the game itself—uses that can be beneficial to the development time. We can re-use much of the code created to manage the A-Life in areas other than straight video game development, such as design, testing, and so on. These uses, however, lie more in the domain of synthesized A-Life, rather than calculated A-Life.

Synthesized A-Life

The next implementation style that we will discuss is called synthesized A-Life. Synthesized A-Life is different from but related to calculated A-Life in that the evolved behavior is based not on calculations, but on a sum of parts that create the behavior in an unrestricted fashion. In other words, we allow the building blocks of AI and A-Life to take the behavior of each system outside of its initial bounds, and within the rules of the game universe, it evolves unconditionally. If the player chooses to influence the game universe somehow, we allow it to react to that interaction from the ground up, whatever the consequences.

That is one facet. Another is the fact that when using synthesized A-Life, we model at a lower level and replace the calculations with a direct outcome by fuzzy logic, which arrives at a weighted conclusion, rather than a yes/no answer.

Take, for example, a tile in the game *SimCity*, which is designated as Residential, meaning that it can be used for houses. If the population is already housed, then that tile will remain empty until a calculation indicates that there are more bodies than buildings—at which point, the tile will be built upon and the population satisfied.

This is the calculated A-Life approach. The synthesized A-Life approach might allow a Sim, walking along the road, to see a virtual For Sale sign, which begins a series of decisions that lead it to build on that square or not. Another visiting Sim might see the first Sim's old house for sale, note that there are good jobs in the area and a nice shopping mall, and decide to buy it. The result is the same. On the one hand, we have dictated the outcome, and on the other hand, we have allowed it to take place. Clumsy though the example might be, this is actually the kind of approach used in the game *Creatures*.

Creatures models in-game A-Life entities from the ground up. Each one has behaviors that are modeled on real life and at a detail level that makes the creatures themselves the actual game. It is the interaction with a single creature that is entertaining, rather like the Sims series of games. Each creature is modeled to have needs (food, water, sleep) as well as emotions (pain, pleasure), and a synthesized brain and body function drive the creature along. This is very close to strong A-Life, and *Creatures* is a ground-breaking game.

Implementing strong A-Life is beyond the scope of this book; instead, we will examine more general AI and A-Life synthesis. But strong A-Life will be used as a guiding paradigm for the sake of comparison because it is helpful. The question we need to keep asking ourselves is, what granularity best suits the game and platform?

Granularity

The term *granularity* here means the level at which we are going to model the processes that give rise to A-Life (be it synthesized or calculated). Granularity really needs to be designed in from the very beginning; it is not easy to start at one level and move up (coarser) or down (finer) during the actual implementation. It is, though, possible to layer different levels of A-Life granularity (grain) as well as allow them to occupy the same game space. Some facets of the game universe will be based on coarse granularity, while others will have fine granularity.

A good example of fine-grain A-Life is, of course, *Creatures*. Almost every aspect of the creature is modeled finely, but the environment is modeled at a coarser level. The balls and blocks that are used as toys are not modeled with the same granularity; they do not need to be.

A coarse-grain equivalent would be *The Sims*, in which each Sim is modeled with emotions, but without biological attributes. Even coarser grains are possible, but the granularity chosen will dictate the effectiveness of A-Life in the system. The choice can be quite objective and practical:

- If too coarse, A-Life will not work properly.
- If too fine, A-Life will take too much development time, as well as hog the processor resources.

A soccer simulation is a good example of game in which the choice can become somewhat sticky. The trick is to choose the right behavioral model for lifelike behavior to emerge while sticking to AI routines that enable each entity to perform well enough to benefit the team. In terms of the control system at least, there are some decisions to be made.

At the finer-grain end of the spectrum, each entity (soccer player) can be modeled with a map in its head that stores the entity's intentions and expectations of the other entities. If the various emotional and physical models are bypassed—such as pain, pleasure, anger, tiredness, and so forth—the behavior of each entity can be modeled as a collection of AI routines at a very low level. The team then becomes an A-Life system whose behavior is dictated by the sum of its parts and that learns and evolves over time as each entity mutates its own AI routines with respect to self-judged success. Goal keepers will learn anticipation and movement, attackers will learn striking position plays, mid-fielders will learn passing, and defenders will get better at tackling.

At the other end of the scale, the coarse-grain version has an AI control system that tells each player precisely where to go and allows some freedom of movement within that mandate. This is almost like a manager who directs the team from the sidelines but allows them some freedom of choice.

The A-Life routines will govern the entities at the individual level, within reason, and also allow the over-reaching AI to adjust its own play so that the strategy (score a goal) is realized. The tactical level decisions can then be left to each player.

The strategy may change with time as the entities learn from the human opponent's moves, but it will start out as standard play (possibly with some adjustments based on the opposition). This coarse-grain control is then used to direct the fine-grain control given to each entity as it tries to master its own destiny.

Top-Down versus Bottom-Up Intelligence

Quite often, the debate between AI and A-Life is rephrased as top-down versus bottom-up intelligence. Top-down is prescribed—a single algorithm derived from human calculation or design—whereas bottom-up evolves from smaller patterns of behavior that are chained together to create a solution.

AI is a top-down intelligence paradigm. High-level solutions are written as knowledge storage and manipulation routines that are linked to actions within the game universe. The solution is a design issue, rather than a run-time issue; and while routines may be implemented in order to choose among solutions, each one is more or less prescribed.

On the other hand, A-Life is a bottom-up intelligence paradigm. Interactions are modeled between small units of behavior that might make use of AI paradigms

to operate and allow the general behavior to emerge naturally. Along the way, we might need to select and breed some additional facets of that behavior in order to get the optimal mix, but that is part of the A-Life way. We are allowing the behavior of the parts to drive the whole. However, the result is often more than just a sum of parts.

Now, unless the game is based purely on A-Life, the behavior will also need AI control. The only other possibility is to allow the behavior to be modeled in true bottom-up fashion using A-Life. AI, however, has many uses that A-Life just doesn't have within the confines of a video game. Sometimes it is not practical to allow behavior to evolve and emerge by itself; AI is needed alongside A-Life. The high-level AI will manage the universe and control the game from a central point, giving us a combination of bottom-up and top-down intelligence.

SUMMARY

In a nutshell, this is what A-Life is about when applied to video games. There are some additional nuances along the way, but within very broad lines, we are trying to achieve lifelike behavior by applying AI routines collectively with GA and GP to allow behavior to emerge, rather than be dictated.

A few points should be reinforced before we go on to see (in detail) how AI and A-Life can be used in video game design and development. First, remember that when the term A-Life is used, we are only talking about a small part of a very deep, interesting, and scientifically important field of study that is not as mature as its AI counterpart.

Specific techniques are used to synthesize lifelike behavior and (hopefully) create phenomena that enhance the quality of the video game experience for the player, as well as provide some useful tools for the designer and developer. Very few games implement A-Life for A-Life's sake; it is just a piece of a larger system.

Any AI techniques used are there to accurately model the pieces of the system that synthesize lifelike behavior. This is what makes A-Life different from pure AI. AI will guide the system and direct the general flow of play while allowing A-Life behavioral patterns to emerge naturally—complete freedom being inappropriate for a goal-based gaming system.

However, when choosing A-Life over AI, we dispense with the strict rule-based system of finite choice or weighted decisions in state machines in favor of (at the finest grain) modeling an entire system. That could mean a single automaton within the game universe, or it could mean an environmental object/entity.

The choice of granularity will eventually dictate how successful we are. A mixture of pure A-Life and applied AI is needed to implement a multigrained system for use in video games. Some areas will be A-Life–oriented (flocks of birds, for example), while others will use prescribed AI (such as pathfinding) augmented with A-Life in the form of genetic algorithms or genetic programming.

Part of the key is to allow evolution within the system. This can happen over the short term (within a play session) or over longer periods. The shorter the time

frame, the less chance we have of arriving at optimal solutions, since solutions (GA) and algorithms (GP) must evolve in order to arrive at their final answers.

Even with a good starting point, the makers of *Creatures* still put the environment into an AI-controlled box. This was helped in part by their choice of granularity for the objects and entities within the *Creatures* game universe.

So before we go any further, this chapter will close with a statement of intent, of sorts: The implementation of AI and A-Life in video games takes the form of a compromise between prescribed behavior, application of information, and true evolving behavior, which all give rise to a system that is more intelligent than the sum of its parts. With that in mind, let us look at some traditional applications of AI in gaming and some video games that have already made good use of AI.

This page intentionally left blank

CHAPTER

2

USING ARTIFICIAL INTELLIGENCE IN VIDEO GAMES

In This Chapter

- AI in Video Games
- Applying the Theories

This chapter will look at the various uses for artificial intelligence in video games and also where AI contributes to the field of artificial life. Areas covered include traditional AI applications for a variety of game genres, such as motor racing, first-person-shooters, and fighting and puzzle games.

We also tackle the notion that AI-driven behavior is reasonably prescribed in video games, a fact that is lamented by reviewers, but necessary in many cases where emergent pattern, learned by the player through trial and error, reveals a way to win. At its worst, prescribed intelligent behavior can lead to undesirable behavior when the rules fail to cope with unintended situations.

The term *prescribed*, as used in this book, covers all the situations that are attached to the more common terminology such as scripted, deterministic, hard-coded, or hand-coded. All of these are examples of AI behavior that is prescribed, although in different ways each time. For example, scripted AI behavior is prescribed, as is hard coded; however, the script might be open to customization by a game designer or even a player, whereas the hard-coded AI behavior might not be.

Where a specific kind of prescribed AI is being used, it will, of course, be duly identified as belonging to one of the aforementioned subgroups of what I call *prescribed* AI.

If strict AI is applied to its fullest extent, then there is the possibility that the game might be too challenging. Achieving the correct AI game balance between "too tough" and "too predictable" is very difficult. The topic is presented in a way that will illustrate obvious solutions for most genres.

A-Life provides an alternative; the behavior patterns are predictable, not because they are always the same, but because they respect the same kind of flexible logic that governs behavior that arises from a combination of instinct and reapplication of knowledge. In real life, we can sometimes predict how people will react, but that does not make it any less of a challenge to deal with them.

First, AI in video games must be introduced, and some of the ways that it can be applied in various genres. We will also cover some traditional AI techniques that are based on well-understood paradigms.

Since conventional AI techniques require understanding some basic concepts as well as paradigms, they will be covered when their use is encountered. What this book's scope will not do is provide a catalog of AI science—only how AI can be applied. Many paradigms are just too resource consuming to be included in video games. Some of the more common paradigms that will be covered are the following:

- Finite state machines
- Fuzzy logic
- Daemon AI
- Neural networks
- The A* algorithm and other weighted decision-making algorithms

Many of the items covered will provide the basis for our discussion of A-Life and AI and their deployment in video games. It is vital that the general theories be understood at this stage, if not the actual implementations of those theories.

AI IN VIDEO GAMES

Artificial intelligence has been used in almost all video games, from the first attempts to modern day epics. These efforts have not always been successful. A leading U.K. games magazine, *EDGE*, has often lamented the lack of good AI in games. The behavior of one in particular has been referred to as reminiscent of "demented steeplejacks" encountering a ladder. Even as recently as 2007, *EDGE's* review of *Transformers: The Game* had this to say:

> "Transformers' *AI makes* GTA*'s barking mad pedestrians seem like HAL 9000 by comparison; stand on a hillside and you'll see cars ploughing into each other at every junction, completely irrespective of your actions." [EDGE01]*

So, although AI has been around for a while, its deployment has not always been "intelligent." There might be a number of reasons for this. Maybe it was bolted on as a last afterthought, or perhaps there are simply not enough resources in the gaming system to allow for sophisticated AI implementation—sophisticated, that is, in relation to the rest of the game. Even today's most rudimentary game AI would have been beyond belief 20 years ago. Perhaps developers deserve the tongue-lashing they get when AI performs under par; after all, more power is at our fingertips than ever before.

As more features and options are piled into the rest of the game, however (vehicles, weapons, additional intelligent behavioral models for the AI to follow, interactive environment, and so on), the AI itself gets left behind. It can even run the risk of becoming no more than a slightly augmented script-following engine.

This partially explains why some AI, such as soccer simulations, tends to be better implemented than AI in more open environments. Games such as soccer are different because the developers have the advantage of being able to repeatedly apply and reapply the same AI for essentially the *same* game, in a restricted environment, over and over again.

As John Anderson, one of three AI developers at Atomic Games dedicated to the development of AI for Microsoft's *Close Combat 2*, notes in an e-mail to the GameAI.com Web site:

> *"AI will almost always be shirked by the software developers/producers, at least in the initial release. This I feel is because most AI cannot be written to be effective until late in the development cycle as it needs a functional game environment to see the effects of the AI." [GAMEAI01]*

Nonetheless, many aspects the elements in our paradigms list will find their way into video game control at a variety of levels, often depending on whether the in-game actor, entity, or object is on the player's side or not. Enemy AI and player co-operative AI are (and should be) often implemented differently, allowing the possibility for the enemy to be overcome, rather than providing an insurmountable barrier.

However, this kind of pseudo-AI is not always implemented intelligently—hence the criticism. Part of the issue here relates to the balance that must be struck between good AI and good gameplay. The two do not necessarily work together to provide value for the player. John Anderson explains this:

> *"Then the developer is faced with a choice of spending several more months to get the AI right, or release a fully functioning game with limited AI capability. Most choose the latter." [GAMEAI02]*

There are, therefore, instances where the developer's initial intent for a certain kind/level of game AI simply has to be shelved for the sake of the development cycle. Of course, the AI must be replaced by something, and more often than not, simple hierarchical rules-based systems are implemented, which use game universe knowledge that they should not logically have. This removes the necessity of sophisticated "guessing" AI, but in terms of the gameplay experience, it is not always to the player's benefit.

The choices appear stark. Either the AI has to be made sophisticated enough that, in the face of uncertainty, it can still reason (like a human can), or it has to be made into a system that can guess the probable best solution from a collection of possible solutions, or it is entirely stochastic. Underpinning these is the possibility that the AI can have access to in-game information that helps it to decide, providing a shortcut that allows the developers to remove some of that guesswork.

For example, sometimes knowledge of the environment is tantamount to cheating. Now, we are not suggesting for a moment that entities can see through walls, or that their vehicles do not obey the laws of physics, but it stands to reason that the virtual driver of a virtual car is more in tune with that car than the real player who has imprecise feedback from his own vehicle, via a control pad. If we coupled together the various algorithms that developers put in place to govern the physical movement of *all* vehicles in the game, as well as to provide thresholds for the virtual drivers, the net result could be a perfect opponent. But, it must also be balanced by the need to be fair and the mandate of providing an enjoyable experience.

Bearing this in mind, there are four general categories of AI that will be covered in this chapter, broken down roughly by genre. The hope is to produce a reasonable overview of AI as it is currently used in the industry and as building blocks for the development of the theme of this book: usable AI and A-Life techniques. These categories are:

- Movement: The manipulation of NPCs within the game universe;
- Planning: The high-level control of behavioral threads within the game universe;
- Interaction: Managing the interface between the game universe (and entities) and the player; and
- Environmental: The appearance and objects of the game universe as manifested toward the player.

Not every game will use all of these categories, as we shall see, and some rare games will *need* to employ at least one technique in each area in order to maintain the assumed reality of the game universe. However, AI should not be used gratuitously. It should either provide a challenge to the player or enhance the experience. AI for the sake of AI must be avoided.

If the developer is tempted to add something that seems like a clever bit of coding but does not advance the game or enhance the player's experience, it is unnecessary. It wastes CPU resources and development resources and could see the project abandoned entirely.

Movement AI

Any game in which the in-game agents (opposition, NPCs, co-operative characters, and so on) move needs some form of control so that movement can be farmed out to routines that provide intelligent behavior. This is true of practically all genres; movement is a key part of the video game experience.

Having said that, some games require more intelligence that others. In shoot-em-up games, for example, the incoming patterns of flight enhance the experience rather than detract from it. The player's ability to read and counteract these patterns is part of the genre's appeal.

We can mix in some clever in-flight decision-making—such as attack, evade, and so forth—but the actual movement of the entities should remain prescribed. The counterpoint to this is the AI needed to control loose flight, as in space combat games. Here, the movement AI is constantly looking for ways to position an enemy attack on the player's craft, while staying out of trouble.

Patterns or standard maneuvers can be employed, but they just form part of the AI, rather than being strictly prescribed. This is also common to racing games, such as *Wipeout*, *Burnout*, and *Formula 1*. The driver AI has to achieve goals, avoid traffic, and remain reasonably challenged, but not be superhuman.

When the movement AI takes on the management of a collection of common goal-centered entities—team-oriented sports simulations, for example—it becomes reliant on the planning AI. In other words, the movement AI implements low-level positioning on an entity-by-entity basis, according to the instructions of the planning AI.

Planning AI

Planning is a two-part affair. On the one hand, there are strategic engines designed to create an overall scenario in which the game tries to achieve a general goal. On the other hand, there is the planning associated with restricted movement. In *Civilization*, for example, the AI needs to plan the eventual overthrow of the player by an opposing force. Concurrently, however, instances might be necessary where movement or action must be planned at a lower level within a universe bound by hills, rivers, and time factors.

In soccer and other team-based sports simulations, planning can also work at two levels: the immediate play (which may change, depending on the player's countermoves) and the general strategy for the match being played. The immediate in-game planning occurs at two levels: each individual and groups of individuals, be that the team as a whole or subgroups within the team, such as defense, attack, forwards, and so on. This is real time, where the plan might have to shift, change, and adapt, depending on the actions of the player and his cooperatively controlled characters.

Even a soccer Sim such as *LMA Manager 2006* needs this kind of planning. The player can only direct, through actions of the manager agent within the game, and the agents that are under this superficial control are entirely autonomous. However, the player has given them tactics to follow and can call these into play during the game.

There may even be longer-term planning AI, which keeps abreast of the team's progression through a season and intelligently manages injuries, transfers, and training activities. These can be broken down into three discrete areas:

- Immediate planning: Reaction within a tactical plan.
- Tactical planning: The current goal as a subgoal of the main strategy.
- Strategic planning: The general long-term plan.

Whether or not to use this level of AI granularity within the system will be a design decision that is also affected by the genre. It is important to make the appropriate choices initially when creating an AI-driven game, because it is usually hard to change approaches late in the development phase.

Interaction AI

This is the most complex video game AI category. It involves everything from the actual mechanism of interaction between the player and the game to the way that the game universe feeds information back to the player.

In a simple text adventure, for example, it might be paramount for the player to communicate with NPCs via a textual interface that emulates conversations. This implies some kind of communication AI—an area understood quite well, but which is often less successfully implemented.

We might understand the challenges of conversational AI, and we definitely understand why it is so difficult to achieve, but we are unable to solve it. Were this an area of AI that was solved, the Turing Test (and the competition that goes with it) would already have been passed and the competition won. We do, however, have a collection of sophisticated models to deploy that *ought* to work reasonably well within the context of a game.

Games that use immediate-control methods, such as driving and other sports simulations, might implement intelligent control methods, such as the driving aids often found in Formula 1 simulations. These need to use AI that is similar to the opponent AI in order to help the player (especially casual or new players) cope with an extremely complex simulated environment.

Then, in games where the player controls a team leader and the interaction of other team members is handled by AI (if only temporarily), management must also be maintained in an intelligent fashion. Like in soccer simulations, this will likely be a mixture of prescribed AI and flexible management algorithms.

Of course, AI exists in everything in between. All genres will have some form of smart interaction. The previous three examples are only a small taste of the possibilities. A single game will likely employ a mixture of approaches that cater to the player's interaction with the game universe, as well as the opposition and the game universe itself. These will likely be layers of approaches—the buildup of a complex behavioral model from simple decisions based on AI routines. This sounds complex, but it is easy to understand if we consider games that include smart weapons with automatic aiming (low-level layer) or the interaction of squad members in achieving a task (medium-level layer).

This concept of layers of AI will be expanded upon as the book continues. For quick reference, it can be equated to the span of control of the system. So, low-level AI might be concerned with single items (objects or agents) within the game universe. Medium-level might deal with groups of low-level items working together.

High-level AI might deal with organizing the general strategy of the AI responsible for providing opposition to or cooperation with the player's agents. These layers are not static, however, and a game will define its own spans of control according to the general granularity of the control systems. It may be that the lowest form of control is a squad, or that, as in *Creatures*, the highest level layer is a Norn, or in-game creature.

As always, there are two sides to this—such as routines to help level the playing field for the player by counteracting the perfect-opponent syndrome, as well as obstructing (hindering) the player's success in a measured fashion. We don't want a game that is impossible to complete; but then again, we don't want a game in which simple memory is enough to take the player through to victory.

Environmental AI

Environmental AI is a tricky proposition. It doesn't deal with opponents, but with the game universe itself—for example, in the game *SimCity*. Often categorized as best used in games in which the player is, in a sense, playing against himself, environmental AI is generally employed in simulations and god games.

Environmental AI essentially handles the game universe's reaction to the player's actions. This might manifest itself through movement AI or interaction AI and might contribute to the input required by the planning AI.

With this in mind, we also need to discuss (at a high level) some basic AI theories that can be used to implement these four areas. Since there is no sense in reinventing the wheel, if the developer can find some tried-and-tested techniques, then this will save implementation time, which can be better used for designing AI into the game, rather than trying to bolt it on afterward.

Common AI Paradigms

Much of AI programming in video games is related to the ability to find a solution within a search space of possible solutions. This solution could manifest itself in a number of ways—chess moves, behavioral responses, or textual output—and employ everything from alpha-beta pruning to Hopfield nets (we'll get to these shortly).

There are also some neural network (NN) possibilities within search strategies, especially when providing learning algorithms to augment the basic behavior of the system. The search strategy employed has to be efficient within the context being used, as well as good enough to provide an answer that is acceptable to the system.

We need a strategy to identify possible solutions within a solution space and then select one that satisfies the criteria, sometimes taking into account the opponent's own moves or limitations within the game universe. This means that sometimes a straight calculation is not good enough, and we need to do a simulation to figure out the answer. Therefore, ways of doing this must be determined that are computationally inexpensive—ranging from simple statistical analyses to neural networks and fuzzy logic.

Behavioral AI need not necessarily depend on searching for solutions. It can also employ techniques to determine the general behavior from external stimuli. This can be anything from a simple finite state machine (FSM), in which the input determines an action and end state based on the current state of the FSM, to a neural network that has been trained to perform an action by using positive feedback. These techniques will become building blocks that we can use to model AI in a video game and provide a basis for adding A-Life to video games, as well.

Simple Statistical Analysis

Statistical analysis is still widely used in AI routines, especially in video game development. Analyzing statistical feedback from the game universe is a good way to determine the behavior of a component within the system.

Statistical analysis is used in conjunction with state machines and other traditional methods to empower the AI within the system. Feedback and analysis of that feedback still have a large role to play, however, and should be made the most of, in the opinion of the author.

One way to look at it is that statistical analysis takes observed probabilities, applies them to the evolving situation, and derives behavior. For example, if we determine that a given chess board state can be generated from the current state, and that it produces a winning situation more than 50% of the time, we might choose it over a state that has produced a winning situation less than 50% of the time. However, by itself, this would be unlikely to generate good chess gameplay. Instead, we need to apply some analysis to the games and decide whether 50% is an appropriate benchmark. In doing so, we might determine that a value of less than 99% is not statistically significant. In other words, just because we won half the time from a new chess position, it does not mean that we will win this time. However, if we have

won 99 times out of 100, the new chess position is a good one to take in the next move. This is one form of statistical analysis that can be used.

Typically, statistical analysis can also provide input for most other kinds of AI, such as NN, FSM, or fuzzy state machines (FuFSM), and allow us to use statistics to generate behavioral responses. This kind of decision-making via probabilistic knowledge helps to reduce complexity and increase the observed efficiency of AI algorithms.

Statistical analysis techniques also produce evolving behavior in which probable actions are based on weighting outcomes balanced against probabilities. This allows the AI engine an additional level of sophistication—and very cheaply. The analysis is fast, and the result can augment the basic decision-making process.

Finite State Machines

FSMs are the building blocks for some quite complex AI routines. They are also useful for modeling discrete behavior where the set of outcomes is known. If we know the number of states that are possible and how we can transition from one state to another based on an action and generate an action as a result, then we can model it with an FSM.

A state of an in-game object (agent, avatar, and so on) is a terminating behavior. For example, a guard might be in a state of patrolling, or a soccer player in a state of dribbling with the ball or marking an opponent. The AI system can signal that the in-game objects or agents can move between states, which they might do with or without an accompanying direct effect on the game universe.

At any given moment in time, the machine is in a possible state and can move to another state, based on some tested input. If a guard is playing cards, and he hears a noise, then we can change his card-playing state to a searching state, in which he will try to locate and neutralize a perceived threat.

Within an active system, states can be arbitrarily added, depending on the application of other AI routines; but this is less common. However, scripting interfaces to FSMs are actually quite common, as we shall see. It provides a good way to create intelligent behavior, so long as the associated actions and triggers are well defined in AI terms.

Usually, each state will also have an action associated with it, and we typically go from one state to another via a given action. These actions can be simple or more complex. A simple action in a fighting game might just be to move forward or backward. A more complex action might lead to a transition to another FSM (or to run one in parallel), such as an entire attack sequence, triggered by a single state change.

In this way, the control granularity can be changed, and the level at which actions are handled depends on the implementation. For example, we could decide that a given action sequence, like fighting, is too complex to be represented by a collection of FSMs. Therefore, a hard-coded AI routine would be provided, rather than an FSM that provides transition details from the current state to the fighting state. This moves beyond an AI routine into scripted behavioral patterns, with the AI used to choose between such scripts.

Alpha-Beta Pruning

Alpha-beta pruning is, in a sense, the combination of two game theories. It uses a *minimax* test, which attempts to establish the value of a move based on minimizing the maximum gain from that move by an opponent. Based on a plethora of possible moves at a given depth of a move tree, minimax is designed to return, for a given player, a balanced move that offers good safeguards against a player winning.

However, on its own, minimax requires analysis of the entire search tree, and therefore its use needs to be optimized in an environment when time is an issue, such as in video games. Alpha-beta pruning offers that optimization. We can prune out those trees that yield results that do not advance our search for the best move.

Commonly, alpha-beta pruning is used in strategy games to help plan a sufficiently efficient (in game terms) path through the various game states. Think of chess, for example, where it is possible for the AI to create a look-ahead of every potential board state from one position to another. This is a waste of resources; alpha-beta pruning can reduce this by discarding those that are less efficient or actively wasteful.

This could be used in almost any genre where there is the possibility that a path through the various game states can be abstracted as a series of intermediate states. Whether this has an actual application in a shooter, for example, is up to the game developer; it may not be worth the effort or may not have an application.

In a video game, let us assume that the AI is capable of analyzing a new state from an existing state and can give it a score. The score attributed to the state, from the game's point of view, is optimized toward the player's failure; the worse off the player becomes, the better it is for the game (opposition).

When using alpha-beta pruning, the trick in the minimax algorithm is that the AI also takes into account the opposition's moves (in this case the player), as well as its own. So a given move is examined to find out if it is better or worse than any previously known move. If either of these is true, we move on and discard this move. After all, if the move is really good, then the opposition will do everything to avoid it; if it is worse than the current worst move, then it is just a bad move. Anything in between is a possible move.

Then we pass the move along and swap the worst (alpha) and best (beta) scores and take a look from the point of view of the opposition, having first adjusted the alpha score to match. This is our current best move. When a point is reached in which we can go no further—that is, the resulting move ends the game—then this move becomes our choice for the next play.

In selecting it, any of the unlikeliest or worst moves have been avoided, and the search time of the game tree has been reduced. There are two caveats: First, we need to be sure that the nodes are being evaluated in the correct order; second, we need a game tree to search. As long as both of these are satisfied, an appropriately good move will be selected.

A*

The A* search algorithm is also a game tree search algorithm, but without the caveat that the entire game tree needs to be known at the time it is executed. The game tree will be built as it goes along and is designed to return a path through that tree.

Like the alpha-beta algorithm, A* will discard any path that is worse than the current path, but it does not take into account the alternative two-player nature in the same way. The most common illustration of A* is in its use as a navigation (mapping plus planning) algorithm.

Assume that we have two points on a map: A and B. We know that the best route from A to B is also the shortest one—a straight line (that is, as the bird flies). This does not take into account any information regarding terrain. In an equal-cost world, where movement from one point to another carries no particular cost or blocking artifact, the best route from point A to B is indeed a straight line.

However, our map shows us that there are some good paths, bad paths, and impossible paths. The A* algorithm is designed to return the best path, taking all of the map's possible paths into consideration. So, our A* game tree is one of possibilities, where each possibility is weighted by the approximate cost of selecting it in terms of the most efficient path and any costs implied by its selection (a hill, for example). In essence, the A* algorithm will select the most efficient path through the search space, based on an evaluation of these two variables. Of course, there are many ways to decide what constitutes the "best" choice, depending on the nature of the game.

Therefore, the chosen path is the result of the shortest path plus the anticipated cost of taking that route. When the shortest (most efficient in terms of in-game cost) is not known and set to zero, it becomes a straight analysis of cost. While this is still valid, it might not produce an appropriate result.

The standard terminology for the A* algorithm is that it is a heuristic search algorithm.

Neural Networks

"Neural network" (NN) is a buzzword in both AI and video game design. It refers to modeling in a way that mimics human brain organization, using memories and learned responses. Imagine a soup of "neurons" that are able to take inputs and post outputs to other free neurons within the system—connected by *their* inputs—allowing trainable network of neurons to be built. In a sense, the connections themselves represent the learning experience.

The network can be trained by adding neurons with specific qualities, for use either in-game or during the play session. Paths through the network can also be reinforced by training, as well as new paths formed.

In a pattern-recognition network, for example, we first train it with connections of neurons that fire according to sensory input that is represented by a shape, along with the information about what that shape is. We then give it random shapes and let the network decided whether the new shapes are the same or not. The more it

gets right, the better the network will be trained toward recognizing the target shape. The exact nature of the neural network and individual neurons within it will depend entirely on the game being designed.

This only holds for those NNs that are allowed to continually adjust themselves once the initial training period has finished. If the NN is provided statically trained (that is, its weights are set by training, or hard coded, and it is not allowed to evolve), then, of course, no matter how many of the random shapes it gets right, it will not progress further.

In my opinion, this is one of the greatest errors of NN implementations; a fully trained NN can be allowed to evolve by adding other mechanisms that alter its output. The original trained NN can be kept as a template, and subsidiary NNs can then be used to adjust, on a continual basis, the choices made by the NN so that the game can adjust to new information. After all, we humans do not stop learning just because we leave school.

One thing that is common to all implementations is that the NN needs to be initialized at the start of the play session, either in code or as a data file. This training might contain general strategies that can be modified during the play session to match the machine's success against the player's, which provides a level of adaptive experience.

In this way, once an NN has been trained in the development stages, it can be supplied as a blank network of weighted nodes that is then initialized with the results of training, which restores it to its trained state. Following this, the NN can then be allowed to adapt further, augmented with other NNs to allow an adaptive network to be simulated, or kept static, depending on the requirements of the game.

Expert System

An expert system can be described as an example of knowledge-based reasoning. It is a well-understood and fairly mature technology. Generally speaking, an expert system comes with built-in knowledge. It recommends courses of actions based on question-asking to narrow down choices, or it opens up new avenues of possible solutions.

The system can also be improved by adding to the knowledge during a play session, depending on available processor resources. In the majority of cases, however, training an expert system needs to be done manually. This can be done by using a design tool to code the knowledge base or (less commonly) by extensive play-testing to extend the system's capabilities from a basic starting point.

Unlike neural networks, expert systems are mostly simple rule-based systems. The goal is to take inputs and decide on an outcome. In the real world, this is like medically screening toward the diagnosis of a specific ailment.

An expert system is, therefore, built by hand (or procedure) in such a way that the questions are logically linked in a path through the system. An NN, as we have seen, is by contrast a black box that is allowed to learn on its own within the parameters of the game universe.

The expert system asks questions and allows the user to select from the options and then recommends one or more courses of action. This information can be fed back into the system either directly—for example, "9 out of 10 people agreed, so

choose option B"—or indirectly via another AI technique. The indirect AI might need some form of memory implemented as a neural network to reinforce the given course of action that resulted in an improved condition.

Expert system implementations can become very complex and processor intensive, so care needs to be taken. They are also generally quite inflexible rule-based systems that are not well suited to general problem-solving. Having said that, an expert system is very useful within a video game development project because it deals with data rather than behavior; therefore, it can be adapted to different data sets.

Fuzzy Logic

Another buzzword, fuzzy logic allows us to choose the inclusion of different sets of values at the same time. In other words, if our video game is creating a population of entities to form an army, we might decide that they could be aggressive, courageous, and intelligent, but at varying levels. For example, we want one entity to be somewhat aggressive, not very courageous, and quite intelligent—all at the same time—while another entity of the same type will be very aggressive and courageous but not too intelligent. We can therefore give approximate rather than precise values for each criterion. This is different from, say, Boolean logic, which is based on precise binary values—the entity in question is aggressive (or not), courageous (or not), and clever (or not).

Fuzzy logic also allows the grouping of more than one (possibly contradictory) set. In the previous case, although our entity is quite intelligent, it might make less-intelligent decisions occasionally. So a decision can be made based on several inputs, and that decision can be fuzzy in its outcome. For example, if the entity is feeling aggressive but a bit cowardly, and it is unable to make a right decision, it will attack only a little bit (low level of aggression).

This leads us to fuzzy state machines, in which we discard the crisp decision-making of a finite state machine in favor of being able to introduce a weighted probability or pure chance of taking a specific course of action over another. This is akin to mandating that if our entity is feeling aggressive and clever, then it should attack most of the time.

Hopfield Nets

A Hopfield net is a special kind of neural network in which each node attempts to attain a specific minimum value, taking into account its current value and the values of its neighbors. Each node in the network is visited at random until none of the nodes change state or an arbitrary limit has been reached.

Hopfield nets can be used for almost anything, from pattern recognition to trained behavioral responses, and are very useful in video game design because they can be easily trained; simply modify the minimum value that each node aspires to. At the same time, because they are based on a deterministic algorithm, each node's behavior is easy to model.

A slight modification to the general algorithm is known as a stochastic Hopfield network. In this variation, an entity that needs to select from various possible actions does each one in turn (simulated) and picks the best outcome or another random possibility. This randomness gives us the stochastic behavior.

Hopfield nets are useful in providing some natural logic, but evaluating one in real time can be time consuming, despite the fact that they are easily trained and easy to model. A Hopfield net also relies on a limited (or at least finite) problem space, so it is not appropriate for cases in which the player might do something unexpected.

Applying the Theories

Having looked at some examples of the theories behind video game AI, we can now turn to several genres and discuss examples in which AI has been used and, more importantly, which techniques might be employed in current projects in order to implement AI behavior. Note that much of what is presented here is conjecture—intelligent guesswork combined with common knowledge.

The more successful a technology is, the less willing a developer might be to share it with the competitive video game marketplace. However, some basic techniques and technologies that have been implemented over the years are well understood by most developers. Independent developers are also occasionally willing to share their experiences through articles, Web sites, and interviews.

In addition, through conferences on AI and books, the basic techniques are shared and honed, and in a professional capacity, developers are often more than willing to share their own personal triumphs of game AI design. For laypeople, these remain out of reach, even though there is an active exchange of ideas.

Motor-Racing AI

When creating AI for a motor-racing game, the enemies are the opposite numbers on the track, and everyone is fighting against each other, the clock, and the track conditions. Therefore, the AI will generally be geared toward planning, movement, and interaction. Of course, the player might also be able to tune his vehicle; in this case, AI is needed to provide a behavioral model that can be modified, such as using straight simulation and calculation of effect from forces, or it could be something more adaptable.

The basic steering algorithm could feasibly be based on "radar-style" feedback from the game engine, giving the AI positional information to work with. From this, the planning AI needs to decide what the best route through the track might be, although the racing line could be encoded for each track and the AI allowed to follow it. In either case (and the latter is less computationally expensive), the AI still has the task of negotiating through the course, taking into account the positions of other vehicles, and managing speed, cornering, and obstacle avoidance.

Interaction AI, such as helper routines to aid the player in steering, braking, and throttle control, is mainly reserved for Formula 1 simulations. But it might also

appear in games where the racing experience is technical and should be tailored to the evolving experience level of the player. In addition, each driver's behavior also needs to be modeled. This might include routines that allow the AI to display aggression or prudence and in-game vengeance against the player for previous actions.

Different drivers might have different traits; therefore, the AI may make different decisions, based on the situation being evaluated. This kind of behavioral modeling might need quite advanced fuzzy logic to be deployed, along with some way to remember and learn behavior, based on an observation of the player (and even other drivers).

For example, these algorithms might lead to aggression combined with nerves of steel, possibly equating to recklessness on the part of the AI-controlled driver, or it could alternatively lead to prudence. An aggressive driver might be tough to beat but prone to vengeful attacks on other drivers, which might lead to mistakes. A prudent driver, however, might be easy to beat, as long as the player holds his nerve.

The *S.C.A.R. (Squadra Corse Alfa Romeo)* racing game also deploys another trick—the ability to induce a "panic attack" if the player drives up very close to another driver (tailgates). This will lead to a "wobble," in which the driver loses some control. What is important is that any driver can induce these wobbles—be they AI or human controlled. The end result is a temporary suspension of regular AI driving skills in favor of a "highly stressed driver" behavioral model. This might not be implemented as a behavioral model at all—just exhibited as a twitch in the steering controls, but it is still an example of worthwhile AI.

In addition to the behavioral model, which may or may not be implemented, the AI also needs the position information to steer the vehicle. This includes deciding where to be in relation to other vehicles, both in general and at close quarters. For example, rubber-banding can be used to keep the car a certain distance from its neighbors, thereby reducing the threshold for more aggressive drivers.

The AI also needs to have a built-in scope for over-compensation errors and other traits that might lead to mistakes in behavior. Otherwise, the player will fall victim to the "perfect driver syndrome" and find that no matter how hard he practices, he cannot beat the opponents.

Based on these concepts, the planning and behavioral modeling AIs now have a path to follow. This is where physics comes into play—the handling model, when applied to computer-controlled vehicles, will mirror the options offered to the player; in other words, no cheating, except when allowed to help the player (such as to slow down the lead vehicles). The accurate physics model must then be coupled with the AI in order to keep the driver's car on the road while following the direction of the planning AI. This might also include errors in judgment or mistakes in the driving model.

These can be linked to the player's own behavior—in other words, worse (more mistakes) for less-experienced human players. This provides an alternative to simply slowing down the lead vehicles, which can appear artificial.

The AI model that does the driving should also be able to be manipulated with feedback from the road so that decisions can be made based on the relationships

between steering, power, speed, and road traction, as well as other factors, such as wind resistance. We might have a prescribed driving algorithm that allows the correct compensation of actions versus the actual conditions, which is then augmented by the AI to produce the behavior. This "feeling" for road conditions needs to be mapped into action and take into account that conditions might arise that change this feeling and possibly the handling model. In other words, driving conditions that affect player should also affect computer-controlled drivers. This includes vision, wet surfaces, wheel types, and obstacles in the road, as well as power-ups and weapons for futuristic racing games such as *Wipeout: Fusion*.

The processing of this information leads to certain actions being performed, which is the output from the AI routines. We could model these at the finite state level (speeding up or slowing down), but we also need fuzzy control mechanisms to allow nondiscrete decisions within gradients of possibilities.

Examples

The aforementioned *S.C.A.R.* manages to imbue drivers with different personalities, thanks to an RPG-style capability matrix that weights decision-making, probably using fuzzy logic algorithms to model driver behavior. It is quite successful, and the same weighted matrix is also applied to the player. By experimentation, it might be possible to use the matrix to augment the behavior of the control system and vehicle, again using some fuzzy logic. The end effect is that as the player progresses and improves his score (in the values) within the matrix, his experience also improves and evolves.

Each driver that is pitted against the player reacts differently when its score is compared to the player's in each category. For example, if the player notices some drivers with low aggression, he might choose an intimidation strategy to achieve a better score or driving position.

(Before the race begins, the player is given the opportunity to examine other drivers on the grid through a user interface that allows examination of them before the grid is shown and the race begins.)

The *Formula One Grand Prix 2* developers claim to have taken this one step further and have modeled their AI based on real Formula One drivers. Their data model contains information pertaining to the behavior of each driver so that the AI can make the appropriate decisions during the race.

This approach also needs feedback and memory during a play session. More aggressive drivers will tend to retaliate, which again might make use of AI techniques to modify the default behavioral model in a neural network. This could result in a particular driver AI "learning" more aggressive driving modes in certain situations, such as when encountering an entity that had cut in front of the driver at some point in the race.

In addition, there are "career" modes. Drivers with less to lose take bigger risks; drivers with bigger engines learn to accelerate harder out of turns but slow down when entering them because they know they can make up the time elsewhere.

Potentially the most difficult part of driver AI will be selecting the best path, based on the positions of other drivers, the state of car, and so on. This requires maintaining a certain speed and distance from other cars within the parameters of the environment, such as the track and road conditions.

Action Combat AI

AI in action combat games has to strike a balance between perfection and incompetence, which is extremely difficult to get right. This genre includes first-person-shooters (FPSs) such as *DOOM*, *Quake*, and *Half-Life*, as well as mech-oriented games such as *Heavy Gear*, *Tekki*, and the slightly ill-fated *Transformers: The Game*.

If the AI is overdone, then the result can lead to impossible opponents, such as the *DOOM* Nightmare Mode; if it is done badly, the result can range from the comical to the irritating. This is especially true if the AI is supposedly cooperative but ends up being more of an obstruction.

One aspect of FPS AI in particular is the ease with which the targets can be acquired and dispatched. On the one hand, the player needs to be able to locate a target and shoot it with an appropriate level of difficulty; on the other hand, the AI has to ensure that the enemy targets act in a way that puts them in a position to attack the player. More often than not, the actual implementations lead to predictable, dependable, and easy-to-confuse AIs that either fail to account for their environments and are unable to deal with new situations or follow such a strict pattern that the enemies' positions are unrealistic, but they are easy to shoot.

Two examples spring to mind—the slightly odd behavior of the pedestrians in *Grand Theft Auto* and the suicidal behavior of the traffic in *Transformers: The Game*. In both cases, the AI completely fails to model intelligent behavior, leading to comical effects with no real drawbacks, aside from slightly embarrassed developers.

As with the driver AI, information to be processed includes items such as the position of opponents in relation to the game universe and player's position, internal status (damage, weapons), and the strategic plan handed down by the planning AI.

The position of opponents can be based on a radar-style system—line of sight (visual) coupled with predictive AI—to track the player in an intelligent fashion. The AI will likely combine data on possible paths with some form of statistical measurement of progress, especially in free-roaming games or flying/shooting environments.

The behavioral AI needs to take into account the status of the actor, which leads to behavioral modeling. This can be represented as FSMs or fuzzy logic, enabling the AI to make a decision based on the temperament and status of the entity. In other words, if the entity is hurt, it might decide to hide and take no risks; but if not hurt, the entity might decide to cause as much damage to the player as possible.

Managing this incoming information and making decisions need an objective AI with responsibility for setting short-term goals with respect to the rest of known status data (Where are we going? What do we have to do?). This might be a combination of A* pathfinding with some fuzzy logic to discover the best solution, or it could just be a rule-based adaptive model.

It might be tempting to cheat the player a little as an alternative to implementing complex AI. For example, opponents should be using line of sight in order to determine whether the player is in view and where he might be going. This needs an observation and prediction AI algorithm. However it would appear easier to just retrieve location data as required from the system, because the game knows where all of the elements are. In fact, this is the same information that would be used for radar-style views (though that might not be part of the game universe or scenario). If radar is not part of the equipment, then the AI routine needs to take into account the fact that radar cannot be used, and that derived algorithms from other information (such as shots fired, or visual or other input) must be used to guess at the player's position, or it must adjust the data to negatively compensate for cheating. The design needs to support one or the other.

In short, the player and opponents should be on the same level. The AI can be tough, but it should remain fair, unless some form of unfairness is supported by the scenario. For example, the *DOOM* Nightmare Mode includes a resource boost for enemies, making it nearly impossible for all but the best players to overcome.

Within the AI, there are several basic behavioral possibilities that can be implemented at the low or high level—such as run, hide, shoot, track, or avoid. A system can be modeled in which the basic states are chosen by the AI algorithms, but each one is encoded as part of the game engine; or we can give more scope to the AI, enabling it to control movement in a scripted fashion.

The granularity of the behavioral modeling allows for implementation of the behaviors supported by script. For example, in *Quake*, the AI is reasonably fine-grained and allows access to the various entities' AI routines. This gives the player the ability to play with the AI and tweak it. This level of granularity adds an extra dimension to the game while providing a good platform for sculpting the AI behavior during development.

Feedback from the game environment will also lead to actions. Here the AI is playing a reactive role rather than a proactive one. So instead of making a decision in the next cycle based on what the AI wants to do (planning, objective, or behavioral), a reaction is modeled on something that has happened that is beyond the control of the game AI.

Observation of the game universe (or the entity's own state) might also lead to a change in the current behavior. If a state machine is being used to manage general behavior, this might be as simple as a monster running away after being mortally wounded. Of course, we might augment this by using a fuzzy state machine that allows the monster greater freedom to decide on an action, coupled with some fuzzy logic to model its actual behavioral tendencies. The state "flee" might still be the end decision; the AI routines to actually manage the task of finding somewhere to hide are encoded not as behavior, but as algorithms inside the program code.

The same can be applied to weapons management—such as select, fire, or reload—in which the AI has an opportunity to make the perfect choice each time. Again, this would be unfair unless the player had proven himself to be sufficiently skillful

enough to possibly be victorious. This behavioral change with respect to player skill is an AI technique that has been implemented with only marginal success thus far. Some games manage it better than others, as we shall see; but the overriding theory is that certain aspects of behavior can be changed, depending on the player's skill.

We might choose to alter the accuracy of a weapon in the hand of an opponent or have it choose an arbitrary weapon rather than a correctly ranged one. As in the driving AI example, this can simply be embodied in a series of mistakes, or it might be something less easily identified, depending on the design of the game.

Examples

The game *Half-Life* has monsters with many sensory inputs, such as vision, smell, sound, and so forth. These monsters can track the player intelligently based on that input. Whether this is handled as modified game-universe state data (taking position data and altering it slightly) or as modeling of senses (or a combination of these) is not known. However, it does seem to work in certain circumstances.

In addition, the game's monsters are aware of their states, and help is sought if they feel that they will not prevail in a given situation. This behavior seems to work as described by the developers.

Half-Life also uses A-Life style behavior modeling (that we will come to in Chapter 3), such as flocking, to deliver a more realistic gaming experience. The combination of finite state machines (a monster is chasing, attacking, retreating, hiding) is balanced with fuzzy logic, creating flexible decision-making and modular AI that allows that flexibility to manifest itself in a variety of ways.

Examples of scripting can be found in *Interstate '76*. AI programmer Karl Meissner e-mailed GameAI.com with a few observations on AI and scripting that offer some great tips for budding AI programmers:

> *"The script specified high level behavior of the cars such as an attack, flee or race. This allowed the designers to make missions with a lot of variety in them." [GAMEAI03]*

This is an inkling of the inner workings of the AI routines in the game engine; only the very high-level behavior is exposed to the scripting interface. The scripting was implemented via an efficient virtual machine:

> *"The overhead of the virtual machine is probably less than 1/2% of the CPU time, so it is efficient." [GAMEAI04]*

The virtual machine that is referred to here is a kind of interpreter within the game system that is supplied with scripted commands. It evaluates them and passes the result back to the game universe. It is known as a virtual machine because it is a machine within the machine, but simulated in software.

The benefit of a scripted interface to the AI is that development can be done faster and (arguably) better. It also allows level designers to have a hand in creating

the game; if it was all hard coded, then they would not have that opportunity. As Meissner notes:

> *"This meant the designers spent a lot of the development time writing scripts. But it was worth it in the end because it made each mission unique and a challenge." [GAMEAI05]*

The flip side to this is that the developers must implement proper scripting language, teach the staff how to use it, and then let them experiment. This all takes time—as does testing the implemented scripts within the confines of the gameplay and universe management.

If a coarse-grained interface has been used, then the other AI routines that have been implemented to produce the actual behaviors must also be tested. Again, this requires time—time that might be scarce on a large project. *Interstate '76* uses finite state machines that are implemented in a simple scripting language:

```
<state> <action>(<actor>)
    if (<condition>)
        goto <state>
```

In the example, we might have multiple states that are similarly coded, and all test specific conditions. The flow of control is simple:

stat➔action➔condition➔new_state

The action might be continuous, and "new_state" might be the same as "state." The internal API then exposes certain possible functions, such as locating the player, attacking, checking to see if the entity or player is dead, and so on. These low-level functions are encoded into the game itself, but they could also have been implemented or run as a script. Remember that layers of scripting reduce efficiency slightly, but there is a payoff for added flexibility—the developer might not be willing to pay the price of less performance.

For the non-programmer, the explanation is simple: The scripts need to be executed inside an interpreter, which is not as fast as straight computer code. Even when the script is reduced to very simple and easy-to-execute low-level encoding, there will still be some level of inefficiency. As Meissner explains:

> *"Originally, there was much more fine grain control over the attacks. As the deadline approached, this all got boiled down to the function, attack (me, him), and then the Sim programmer (me) did all the work. [GAMEAI06]*

So action combat games generally make good use of AI—pathfinding, behavior, adaptive behavior, reasoning, tracking, and so on—via hard-coded AI routines and extensible scripting interfaces. There is also ample scope for improving the existing implementations, with negligible performance impact, through the use of A-Life such as flocking and other emulations of natural behavior.

Fighting AI

In one sense, implementing fighting AI ought to be easy; it takes place in a small game universe with prescribed moves. It is also sometimes part of a larger game storyline in which fighting provides a means to an end, rather than the actual focus of the game. In this case, there is less scope for real AI that can adapt to the player. Still, players have come to expect some level of intelligent behavior. In this case, learning is hindered by infrequent fighting interaction with the player, so rule-based AI, with some small level of randomness in the decision-making process, will probably be the most likely implementation.

On the other hand, when the player and AI are always fighting, there are more opportunities for the system to learn from the player and adapt itself according to the player's style. This is a fairly advanced use of AI that has become a reality in some more recent games.

The upcoming *Street Fighter IV*, for example, by all accounts will make use of a very adaptable AI. Many arcade fighting games also seem to have this approach wired in—from *Virtual Fighter* to the original *Street Fighter* and *Mortal Kombat* machines. However, in these, there always seemed to be a killer move that would work every time, something that should be avoided in a game if at all possible by deploying adaptive AI.

Fighting AI is characterized by very discrete goals in the short term, coupled with reactionary mechanisms that are designed to counteract the player's own moves. The simple actions and movements are then augmented by some tactical planning and possibly a strategic element that ties each bout to the next via a storyline, but the information is retained between each individual fight.

So we would like some gut-level reactive AI, some tactical AI, and possibly some strategic AI, depending on the exact nature of the game. This could be coupled with pattern recognition, adaptive behavior, and learning systems, perhaps using simple neural networks. If we consider the fighting AI in isolation from the rest of the game, each bout can be managed as a series of FSMs at a high level to create sequences; FSMs can be combined to enable medium-level behavior that is dependent on the goal set by the managing AI. This could be as simple as just beating the player to a pulp and obtaining a knockout (or killing the player) or winning some other way. The retreat or surrender mechanism could be part of a larger game (or scenario), but what is important is that this behavior is managed from outside the fighting AI, but it is offered as a goal to the planning AI that controls computer-opponent movement.

Other strategic elements that will affect the AI are point and knockout wins in classic fighting games. A canny computer player might hold out for a points win if it feels unable to guarantee a knockout. This kind of behavior must account for the perceived status and characteristics of the player-controlled fighter. This input information can be augmented by pattern recognition that serves two purposes: learn from the player to acquire new tactics to reuse, or learn how to counteract player attacks for use in future fights. Of course, any neural network or expert system that has been used for training will need to be prepared with some basic behavioral models out of the box.

The system receives movement information that can be translates into sequences of moves (or combos), and this can be used to augment any pre-installed movement sequences. Some reaction AI models will have to be included to deal with defensive mechanisms.

So the movement information model is not just kick/punch/duck; it will also include movement to the left or right, as well as jumping, and could be used to build networks of behavior that are held together by adaptable FSMs. In this way, the behavior can be changed for a particularly gifted human opponent, while remaining beatable.

One aspect of this is that this information is available to the system before being used in the game to alter the visible game universe. In other words, the machine knows before it animates the player character what the move might be. As in other implementations, it is tempting to use that information as a shortcut, rather than create properly balanced AI. However, using this information can be difficult to avoid; our counteraction is to flaw the AI somehow, as was illustrated in the previous section. As long as we compensate for any predictive AI that was left out, it is not considered cheating. But remember: The moment that the machine uses information before it is visible to the player or before it is used to update the player character, a level of unfairness creeps in.

AI also has a role in providing feedback on the state of the player, information that is sent to the player by altering the way that the player character reacts. A punch-drunk boxer under the player's control might be managed by clever AI routines that emulate the effects of the boxer's condition, and this feedback is also sent to the data store of the opponent, who may or may not act on this new knowledge.

So the system also needs information on the avatar (player), which gives rise to the possibility for some interesting player dynamics. If behavior is modeled accurately, the player might be able to lull an opponent into a false sense of security before landing a knockout blow, for example.

Some fighting games already implement similar mechanisms, allowing the player to play badly right up until the end and then unleash an attack with a knockout blow. This is an unintended consequence of adaptive behavior that exposes a flaw in this approach over strict rule-based AI.

Examples

There are fighting AI engines implemented in almost every game that contains some version of "player versus computer." Swordplay abounds in games such as *Prince of Persia*, and there many fight-oriented marshal arts games, from *Street Fighter II* to the *Virtua Fighter* franchise. However, few have attempted to implement AI that is capable of adapting to the player while maintaining a unique playing style. They all allow for difficulty ramping, where the computer challenger improves as the player gains confidence and skill.

More often that not, these take the form of simple speed and damage-resistance boosts, rather than true behavioral changes. Also, the entity might gain the ability to

use different attacks over time, which is halfway between AI and adaptive systems, and a workaround.

There is some evidence that *Virtua Fighter 2* tried to offer adaptive play and unique character styles (beyond special moves) but seems not to have delivered it properly. At least, reviewers were unimpressed. *Fighting Wu-Shu* also claimed to be a fighting game that delivered this kind of AI behavior. It is an approach that is clearly better than cheating. Characters move incredibly fast, intercept player moves, and dodge out of the way, providing more interesting gameplay than when a static rule-book is employed. Players like to be able to read an opponent, though, so perhaps the true solution lies somewhere in between.

This would entail some form of FSM to drive the AI process, such as a semi-static rule book of combo moves that reflects the challenger's style, coupled with some fuzzy logic to dictate the circumstances under which these moves are hooked together. In addition to adaptability based on the entity's state, some learning algorithms can be placed on top to enable the entity to copy the player's moves or combos that resulted in a hit or win.

Due to the fast-moving nature of these games, there is no time to perform in-depth searches of the problem space, so split-second reaction AI has to be favored over pondering, strategic AI routines. There seems to be room for better AI in this area, however, judging by reviews of fighting AI in the marketplace.

Puzzle AI

Puzzle games are traditionally an area that computers do very well at; they are usually based on logic and lateral thinking, which can be easily programmed. In addition, we can often calculate a large problem-search space, using an A*, alpha-beta pruning, or minimax routine to enable the computer to select a possible best move from a selection of lesser alternatives. In addition, new games and mini-games are being added to the existing glut of board and card games already on the market. All of them use some kind of AI—from strict chess-playing algorithms to the looser behavioral models used in some poker games.

In video games, puzzles sometimes form the foundations of larger games or just serve as games within games, where the puzzle AI is probably less important. Nonetheless, it is a useful and interesting gaming AI field if only because it illustrates some of the classic AI routines very well.

In fact, many puzzles have been "solved" by AI, such as chess, *Connect 4*, go-muku, and possibly tic-tac-toe. All of these games are based upon strict rules that make them easy to predict and evaluate. This is helped by the restricted environment and restricted move search space. We know that the possible range of moves is finite, even if it is a huge set.

These games been around a long time and are well understood, which means that they are open to being implemented as algorithms that can compete with humans. If a computer can repeatedly beat a human under competition conditions, then we can say that AI has solved that game. These are quite sweeping statements, and some of

the AI is very complex and outside the scope of this discussion. What we can discuss are the definitions of action and output as they relate to video game creation.

The puzzle game exists within a relatively easily defined information space, in which both sides' movements are restricted by the game rules. This is a predictable, limited problem domain—as opposed to a game like *Grand Theft Auto*, where the player is free to go and do wherever he wants; actions cannot be planned.

A restricted game universe, such as a puzzle game, on the other hand, delivers information that provides the basis for various actions (moves) that are arrived at after careful analysis of all the possibilities. It is normal for the puzzle-playing system to simulate the game universe (board and/or pieces) in future positions, based on possible/probable moves by either side.

The eventual action is borne out of this analysis and should be the result of careful evaluation of each possibility. In our discussion of alpha-beta pruning, we saw that the machine will attempt to find a move that satisfies conditions relating to an evolution of each player's best and worse outcomes. This form of look-ahead takes each problem domain representation one step forward at a time and then picks one possibility as the move to make. Chess is a complex example, but a game such as *Connect 4* might prove easier to examine.

In *Connect 4*, a player has the choice of eight places to put his piece, so a look-ahead of one level reveals only eight possibilities. Each one of those then has eight possibilities for the opposing player (if we are using the alpha-beta method). So a look-ahead for two levels has 8×8 possible decisions: 64 in total. In this way, the player's moves can be modeled along with game's response, and an appropriate move can be chosen from all the possibilities, using a variety of different algorithms, including A*, alpha-beta pruning, and Hopfield nets. For simple search spaces, A* planning might work quite effectively, as long as we can estimate the efficiency of the ideal solution algorithm at each stage—another example of finding the least costly or most beneficial solution when using a problem-space algorithm.

The look-ahead is very effective in terms of prediction, but it also involves some quite extensive number crunching. Anything that will reduce the number of comparisons we have to make will help; the game designer will have some important decisions to make when developing the game information and search space representation. For example, specific opening books can be used to augment the initial sequences because these have predictable or known response patterns. When designing a new game, situations can also be invented in which the player or computer will be restricted to certain moves, thereby removing the need to analyze all possibilities.

Once the set of possible moves has been pared down, some kind of move analysis must be performed—a weighted comparison—in order to ensure that the best move has been chosen. Any game that involves a puzzle can use weighted comparisons of solutions in this way to determine the best response to a given move.

In non-puzzle games, we can also use this technique, as we will see in our strategy AI discussion. In a sense, some of these same theories and practices occur in non-puzzle games, and many of the technologies can be reused in new environments.

However, humans are still better at some things, due to our superior spatial awareness and environmental perception. We excel over computers in some things, and vice versa—for example, humans can visualize a problem solution better, but a computer is better at performing in-depth analyses of all the possibilities.

Using A* for a problem-space deep search takes a long time. Even the most optimized alpha-beta pruning methods are not appropriate for a look-ahead that is all encompassing. The famous chess program *Deep Blue*, which has successfully beaten Gary Kasparov, uses a 12-level look-ahead—over 3 billion possible moves.

Adventure and Exploration AI

In a sense, adventure and exploration AI problems are similar to those in puzzles. While the actual game is not really puzzle oriented, the same AI techniques must be deployed in order to be able to counteract gifted or persistent players.

AI is used in many RPGs and other game genres, such as MMORPGs (Massively Multiplayer Online Role-Playing Games), to provide behavioral modeling and forms of goal-setting (scenario), communication, and interaction AI. Nonplayer characters often inhabit the game space along with players, and they need to be controlled in an autonomous fashion by the computer.

The AI itself may consist of several parts, depending on the autonomy allowed to machine-controlled NPCs within the game universe. The AI must be able to plan and map, as well as interact and sometimes do battle with the player or other in-game entities.

Planning and Mapping AI

Part of the planning AI will use goals set by the system at a high level to set the scene for a given chain of events. Some of these plans might be prescribed, and some might be developed as part of the AI process in order to challenge the player.

The planning AI behavior reflects progress against plan; in other words, it needs to be adaptive in the sense that new plans can be created to manage unforeseen circumstances. If the system is managed in terms of point-by-point movement, this is relatively easy to achieve, but most games also require the AI to manage spatial awareness and inventory at the same time. Spatial awareness is often embodied in mapping algorithms. The principal idea is to implement an algorithm that is capable of knowing where the NPC has been and maintain an accurate map of this information. This information can then be used to chase or avoid the player, depending on the behavioral model being deployed by the entity's AI at a given moment in time. This behavioral AI will likely be managed by a separate action or combat AI, as previously described, with the exploring or adventuring AI providing high-level goals.

This mapping information can be discovered and retained or, more likely, passed on to the NPC by the game as a shortcut—an alternative to using memory-intensive AI routines. In addition, the various tasks or goals that the machine tries to complete are maintained as part of that process. These might range from simple "block the player" tactics to more substantial strategies, such as in *Half-Life*.

The AI itself might need to strike a balance between planning and mapping in order to maintain efficiency. In other words, a very low-resolution map might be used, freeing resources to enable better behavior and intermediate planning. On the other hand, mapping might be more important, so we could rely more on the development of instinct-style planning, but with a perfect map.

Once again, information provided to the AI takes the form of position, state, and environmental information. At a high level, the game universe sends information, and the actor receives that information.

Behavioral AI

The behavioral AI model (passive versus active) will dictate whether the management AI is playing a role to keep the actor informed of the game universe, or whether the actor is allowed to roam freely. There is an implication here that concerns processing overhead.

If the actor is allowed to roam freely, then the controlling machine needs to model its behavior much of the time. If it is the game universe that prompts the behavior, there is much less scope for the NPC to find itself in a position not prescribed by the game engine, and therefore it can be activated based on player action.

Coupled with this is the status of the player *vis-à-vis* the various NPCs in the system. Their behavior could also change, depending on the player's status; clearly, a passive role is going to be easier to modify than an active one where advanced AI techniques (involving weighted matrices) will be needed. The NPC behavioral models will fall into either cooperation or obstruction patterns, following lines similar to combat action AI.

Among other data available to the AI is the position information of the player, as well as other visible in-game artifacts, such as other NPCs. On the one hand, we could model them all as individuals, but with similar behavioral patterns that follow a rule-based system—such as in *DOOM*, where beings rarely cooperate against the player—or we could allow them to communicate and cooperate. This could be engineered in a coincidental, swarm-based fashion, such as in *Robotron*, where beings seem to cooperate. This is the easiest algorithm, but it is not really an example of AI, as such; rather, it lies somewhere between instinct-style homing and point-and-shoot AI.

For example, in real cooperative AI, actors have to worry about not killing each other, something that neither *DOOM* nor *Robotron* takes into account. On the other hand, more advanced games like *Half-Life* do implement this kind of AI.

The pathfinding AI is yet another example of search space analysis that uses algorithms to determine the correct choices. In this case, pathfinding often uses an A* algorithm to find the shortest path with a high score in terms of movement cost. Using this algorithm, it is also reasonably easy to work planning data into the evaluation process. The result is that the path that is chosen is the best possible (lowest movement cost), assuming that all else is equal, such that a path that follows a plan is chosen over one that does not.

Some of these assumptions will change with respect to the anticipated cost of encountering the player or with the various behavioral matrices of the entity.

Some fuzzy logic will likely need to be employed to determine whether the entity will choose a path that will bring it into contact with the player, assuming that the combat and action AI contain sufficient intelligence to be able to predict where the player might be.

Finally, networks of FSMs can also be used to implement behaviors such as "patrolling-style" movement. These could be coupled with basic learning algorithms to allow reuse of the behavior in different situations. For example, a wall-hugging FSM could be reused to develop a path in memory that could then be employed by the AI to determine, with an A* implementation, possible variations with respect to the changing game universe.

Having evaluated all the information, there are plenty of possible actions that can be taken, depending upon several factors. For example, the freedom to move around and collect items or weapons will have an effect on the range of actions permitted.

The designers must decide exactly what NPCs are allowed to do and what they cannot do, because these actions will become part of the underlying AI implementation. There are some shortcuts that can be taken to allow a high level of interaction, such as smart objects with triggers that promote behavioral responses.

Environmental AI

Environmental AI is similar to an environmental trigger—like the smart tiles encountered in the first section of this chapter, which are designed to induce a reaction in the AI. Using these kinds of techniques, we can allow the player to put things into the environment and get responses from NPCs.

Some of these responses might be simple, prescribed actions or reactions, such as in *Ravenskull*, for example, which uses environmental tiles to guide the growth of fungi, as did *Repton*. While these are not examples of advanced AI, they are techniques that can be deployed effectively.

These days, we can also have more clever environmental triggers that guide the AI processes, rather than just providing an automatic response. The kind of AI that we are striving for is comparable to basic instinct, versus more powerful problem-solving or knowledge-based systems.

Most movement algorithms are not based on look-aheads. In other words, we don't let the NPCs cheat; they need to move just like the player. However, some analysis of the best possible outcome in a given situation might still be applied in order to mimic lifelike behavior.

Since the AI has to cope at several levels, it might end up being encoded as a simple FSM that governs which AI routing will be used in a given situation. This is an example of using an FSM for behavioral patterns, as opposed to just using it to model static behaviors.

Interaction AI

Finally, we have actions that relate to the interaction between players and NPCs within the confines of the game design. This could include the exchange of items

or conversations on a point-and-click level (or something more advanced, as in *Ultima Online*). In-game conversation needs its own brand of AI based on language processing, which will be covered in Chapter 3, "Uses for Artificial Life in Video Games." At a more basic level, the AI might include triggers that elicit various responses under the guise of true communication. This level of interaction can use any of the techniques discussed so far, from smart objects to FSMs and neural networks, depending on the exact nature of the underlying game.

If we want to strictly control the outcome of the interaction AI based on the input data, we can categorize that data into one of three groups:

- Hinder the player,
- Mislead the player, or
- Help the player.

Most interactions serve to further one of these objectives, and there are examples of each in most games. For example, hindering the player can take the form of simply attaching some environmental modification, such as a door closing, which will prevent the player from progressing further.

Misleading the player can take several forms, but could also be some kind of modification to the environment, such as planting evidence that will persuade the player to take an incorrect path. Direct interaction with the player could lead to textual lies or half-truths designed to prevent the player from solving a puzzle—or some form of Gollum-induced delusion that will prevent the player from correctly evaluating the sequence of events.

Finally, help can come from several corners—from helpful NPCs, such as the doctor in *Half-Life*, or NPCs that follow their own scripts, such as the police in *Getaway: Black Monday*. The difference in *Getaway* is that the police are fairly predictable, while any helpers designed to interact and be directed will likely need some kind of personality of their own in order to be of any real use. This also tends to make them difficult to debug and test, and they are also potentially prone to unstable behavior. In *Half-Life*, the helpers seem to exhibit a good balance between some form of self-determination while being scripted and fairly interactive.

Examples

Great examples of scripted NPCs can be found in *Baldur's Gate*, a hit with RPG fans worldwide. The scripting, which can also be changed by the player, offers new features and challenges to players.

The scripting itself seems to be rule based, with top-down processing that employs weighting to control the probability of some actions taking place. Again, as in other scripted AI engines, the actual AI remains hidden, with scripts reduced to FSMs.

The A-Life used in *Baldur's Gate* is incredibly sensitive to the player's own actions, extending to revenge wrought upon a player that robs an NPC; the other NPCs will subsequently ignore the player for a time. Criminal behavior is also punished, to the extent that guards will eventually hunt the player down. The guards are, by the way, invincible.

Simple examples of this can also be seen in *Elite*, a space trading and fighting game by David Braben and Ian Bell. In this game, the player's actions can trigger the release of the police craft, who then hunt the player down, as in *Baldur's Gate*. This might seem like an example of knee-jerk AI, but it does add to the experience of the player and creates an additional layer of immersion, however simple the mechanism.

According to GameAI.com, scripts can perform basic tasks, such as selecting weapons and choosing to fight or flee. The actual attacking AI is still controlled centrally, but presumably there are interface commands that allow the NPCs to move upon demand as well as find out information about their environment.

As an alternative, *Battlecruiser: 3000 AD* developer Derek Smart revealed on GameAI.com how neural networks are used in the game to control all NPCs. This is something of an achievement, and his comments, lifted from GameAI.com, are quite telling:

> *"The BC3K Artificial Intelligence & Logistics, AILOG, engine, uses a neural net for very basic goal oriented decision making and route finding when navigating the vast expanse of the game's galaxy." [GAMEAI07]*

This is a classic example of a planning and mapping AIs in action. In the referenced game universe, some of the navigation can be quite complex, as it is unrestricted space travel. Apparently, this navigation also employs a "supervised learning algorithm" that, according to Smart, "employs fuzzy logic … where a neural net would not suffice" to try and learn its way through the game universe.

Along the way, Smart notes that "Some training … is basically done at a low level," and the entire movement management also includes "threat identification and decision making" to help it navigate against the player's moves. Since much of the control is farmed out to neural network-managed NPCs, some training is also provided. These were supplied through "pre-trained scripts and algorithms programmed to 'fire' under the right conditions," which is an example of a starting-point neural network that is designed to be augmented with experience.

One final comment: Smart states that he "simply took what I had and … made it work for my game," which is great advice. The game development cycle is complex enough without trying to re-invent existing paradigms or invent new ones.

History is full of examples (many classics documented, often with e-mail correspondence between the maintainer and the game's authors, on http://www.gameai.com/games.html) in which games abandoned new or unique AI at the last minute because of complexities in the development process. The end game is usually poorer for the omission of good-enough AI, especially in places that called for a more innovative solution. The designers would have been better off using good-enough AI and adding a few twists to that, rather than trying something completely new.

Strategy AI

Strategy AI exists in a distinct genre between puzzles and simulations. It has much in common with games like chess, is often based on simulated realities, and lacks

the first-person perspective or action element that would place it in either the adventure or combat genre.

Strategy AI will contain all kinds of different AI algorithms as it tries to outwit the human player on several levels. Unlike strict puzzle or board games, the rules and environment might be open to interpretation and modification during a play session. Therefore, an AI is required that can cope with planning, mapping, and strategy within a changing playing environment.

Subsequently, there is plenty of scope for feedback and rule-based AI, as well as search-space algorithms such as alpha-beta pruning and A* pathfinding. The individual units used as "pieces" will be given some degree of freedom, and their AIs also need to be implemented. This is rather like giving a chosen chess piece the freedom to decide where to move.

The planning unit will use rules and feedback from the game universe to decide which unit (or piece) should move and what its goal should be. Therefore, all the elements of the planning AI are also necessary.

Information and Game Rules

In a sense, strategy AI pulls together all the other aspects of video game AI, from balanced combat to vehicular and unit movement, as well as autonomous and scripted behavior. The game rules and status of the player provide the decision-making process with the necessary framework and will dictate the kind of AI algorithms used.

This requires that the designers allow for the maintenance of the machine's status and game universe (including other NPCs) with respect to the player. These analyses are all part of the AI processing. This helps when it comes to planning the next moves with respect to what the player might do, either based on statistical analysis or using a neural network to provide first learning and then anticipation. When the game is deployed, we can modify the system's behavior to react to strategies that the player has devised.

These observations are made easier by the presence of game rules that restrict possible options, but larger, more complex and populous game universes become more difficult. This means that strategy AI, more than in most other genres, has a lot of information that needs to be processed in order to carry out game actions.

The possible range of actions depends on the game universe and the rules that govern it; they can be local, global, or environmental. For example, a local action can affect a small area, a global action will have wider-reaching consequences, while an environmental action will change the way the game is played at a fundamental level.

Actions often just amount to hurdles for the player, rather than direct actions against him (like shooting them), which are more common in other genres. Nonetheless, the actions can still be modeled by FSMs or other kinds of action networks in which a state change and action can be used to generate other changes/actions.

Generally speaking, every action's goal will be to prevent the player from achieving something—another application for alpha-beta pruning and related techniques. Games such as *DefCon*, where strategy is the key element, take place at a slow pace,

and each move made by the computer is preventative, rather than a direct attack.

Board games such as Othello and Reversi are other examples where there is no direct attack, just some vague sense of "prevention," as opposed to chess, where elimination of the player's pieces is high on the agenda and is key to winning the game.

Even Scrabble has some strategy mixed in; playing the right tiles with the right points at the right time, on the right squares, goes beyond simple puzzle-solving. It needs a more heuristic approach.

Examples

Age of Empires developer Dave Pottinger (Ensemble Studios) describes his approach toward augmented AI with learning that can carry over play information from session to session. Even though the final outcome in *Age of Empires* did not quite live up to his expectations, the initial intention is embodied in a statement posted on GameAI.com:

> *"This has helped the quality of the AI out a lot. Well enough, in fact, that we'll be able to ship the game with an AI that doesn't cheat." [GAMEAI08]*

When the game is played for first time, the sides are even, and we need some way to progress with the player. This gives the machine some ability to counteract the player, rather than just follow the rules that dictate its initial behavior. Learning is the key to achieving this noble goal.

As Pottinger notes, "If the AI does something markedly different … then you get a more enjoyable experience." This is one of our key goals in implementing game AI: Entertainment or added value is as important in some games as increased difficulty.

So one challenge is to create AI that does not simply repeat the same actions when combating the player's strategy, and another is to ensure that the player cannot find a strategy to beat every possible machine strategy. This is a tricky, almost an impossible proposition, because the player is the ultimate opponent. Pottinger goes on to say:

> *"I've yet to see an AI that can't be beat by some strategy that the developer either didn't foresee or didn't have the time to code against."*

Given that we cannot foresee everything, and we do not have time to code for all possibilities, we need to implement some kind of learning and analysis into the strategic AI. Neural networks coupled with alpha-beta pruning would probably be able to handle this reasonably efficiently.

However, in the end, *Age of Empires* failed to ship with built-in learning and apparently employs an expert system combined with finite state machines to create the behavior. This has brought the game some criticism, but on the whole it works fairly well.

We can also see evidence of multilayered AI in strategic AI implementations in games such as *Close Combat*. Developer Bryan Stout describes some of the strategic AI's workings on GameAI.com:

> *"The SAI is comprised of three main systems: the location selector, the path planner, and the target selector." [GAMEAI09]*

Close Combat relies on player input and uses a system based on hierarchy, with the player giving the high-level orders and the strategic AI determining the medium-level goals, including target selection, and then carrying them out. The same AI is then deployed from the other side, but with a computer AI giving the high-level orders. Gary Riley, another *Close Combat* developer, picks up the thread:

> *"When dealing with enemy teams, the location selector hypothesizes the location of enemy units rather than cheating by looking at the real positions of teams that it should not be able to see." [GAMEAI10]*

Here again are the negative connotations of cheating. This time it is to the benefit of the player, and we can only assume that the enemy teams together under a guided AI system where high-level goals are generated, whereas the human player is not modeled along the same lines. Either way, there is a clear possibility that cheating might be an option, but the developer chooses fairness and simulation over a shortcut.

Judging by the look and feel of the playing methods in *Advance Wars*, a similar method is used for employing tiles, pathfinding, goal selection, and target acquisition and destruction. It is also likely that actions and movements are based on weighted values of enemy units, as well as probable movement of other units (the player's) around the "board." Riley explains how this was implemented:

> *"For example, the simulator determines line of sight to a team by tracing from individual soldier to individual soldier, but the high-level AI has to have some type of abstraction which divides the map into locations and provides information about whether a team can fire from one location at another. If the abstraction didn't work well, you'd either have teams moving to locations from which they couldn't attack the enemy or moving out of location from which they could." [GAMEAI11]*

Clearly the information representation is a major issue, as was previously noted. Choosing an appropriate abstraction will also have an impact on the solutions selected for determining how a unit should move. Riley continues:

> *"The solution we ended up with was to iterate over all locations on the map deploying teams into the divided map locations and then have the simulator determine whether a line of sight existed (which took a considerable amount of time)." [GAMEAI12]*

This is a fairly time-consuming solution to an interesting and vital problem in strategy AI, and there may have been better solutions available, but it is likely that the pressures of meeting deadlines meant that the final decision was rendered less than efficient out of necessity. Alternatives might have been Hopfield nets or alpha-beta pruning using line of sight to register locations appropriately.

Finally, another example of layered strategic AI is evident in *SWAT 2*, published by Sierra FX. Christine Cicchi (Sierra) offers some excellent insights on the GameAI.com pages:

> *"What makes SWAT 2 different from most other simulation games is that challenging, realistic gameplay requires a high degree of coordination among the non-player characters. A SWAT element must move together and act as a team, timing their actions with those of the other elements. Likewise, the terrorists have scenario-dependent goals and actions to pursue without the confines of a linear script." [GAMEAI13]*

These scenarios are likely hard coded and allowed to develop within a broad scope as the game progresses. Underneath this layer, there is the tactical AI:

> *"Tactical AI gives units their individual personality and intelligence. It tells them how, when, and whether to perform advanced behaviors like finding cover, taking hostages, or running amok." [GAMEAI14]*

This is embodied in low, medium, and high-level behaviors that can range from simple movement to shooting and multistage movement, right up to combination-based behaviors such as "advance under cover." In a given situation, each unit is then managed by AI routines that account for internal settings in tandem with a stochastic algorithm:

> *"The unit's response depends upon its personality, but with a slight random element to bolster replayability and realism." [GAMEAI15]*

Fuzzy logic is used to handle the four main characteristics—aggression, courage, intelligence, and cooperation—and produces an action based on following a simple set of hierarchical rules: "If courage is low, run away. And it provides a great way to add realistic AI for very little overhead, as long as the options are available. We just need to choose between them.

Fuzzy logic comes into play when an actor has a personality that tends toward a certain temperament but still has elements of others. Subsequently, even a courageous unit might conceivably run away (all else being equal) or be pressured by circumstances into running because its courage is overridden by other factors.

This approach is geared toward achieving the right behavioral balance in an otherwise strict rule-based AI system. The end result deploys fuzzy logic and fuzzy state machines to make sure that the rigidity of the system has a counterpoint that makes it more playable and ensures that the player cannot repeat the same behavior and win each time.

Simulation AI

Finally, and almost as an extension to strategic AI, we have simulation AI. This is generally linked to environmental simulation, although there may be some entities that are managed by the AI system that have an indirect effect on the success of the player as well as the general game universe. Examples include *SimCity*, *Theme Hospital*, and *Railroad Tycoon*, as well as the seminal *Black&White*, which is a god game with some real AI and A-Life implementations. (We'll look into these in Chapter 3.) The key here is that the player is, in a sense, just playing against himself and the environment.

There is no real opposition beyond the game rules; they are there for the player to master, but generally the player's downfall will be greed or mismanagement of the game universe. The conjunction of rules is all interconnected, and so this mismanagement or greed can cause the environment to rebel and have a direct effect on the player.

Conversely, part of the attraction is that the player *can* have an effect on the environment and observe that effect as dictated by the game rules. Sometimes things go the player's way—mostly they do not, and the play session becomes a series of little crises to be resolved.

Information and Game Rules

As in our other examples, the game experience is basically just the control of variables with little direct action. Strategic AI offers some basic, direct action in games that involve combat; but in simulations, this is generally not the case. So it becomes more important than ever to get the data representation and monitoring right.

The actions in the system are therefore the manipulation of events within the game universe. In *SimCity*, we can point out areas to build things, like roads, houses, and so on. Sometimes these things appear, and other times they are dependent on the system's game rules. For example, unless there is a growth in population (event), no houses will be built in the residential zones. Without the houses and citizens, there is no need for commercial zones for them to shop in. Furthermore, without industrial activity, there will be no jobs for the citizens, and therefore they will leave.

The system's input variables provide an alternative to the direct action favored in other AI models. This also means that players are quick to recognize and predict certain behavior, which is part of the problem with most simulation AIs; but arguably, predictability also forms part of the appeal for players.

Problems and Challenges

In order for the game to be successful, we need to find ways to make it fun, while also allowing the AI to work on a strict rule-based platform. Without the rules, we cannot model behavior; but at the same time, this might restrict the game universe too much.

The key is to avoid micromanagement but still allow flexibility. This allows the AI some scope to make decisions on behalf of the player, which is key to the simulation side of the AI. It can be performed by real simulation or just statistical analysis and calculations.

These issues are more about actual gameplay than they are AI, except that the gameplay in a simulation is, in a sense, only really about the AI. Games such as *Creatures*, *Beasts*, and *Pets* also fall into this category, but they are more A-Life than AI and will be discussed in future chapters.

SUMMARY

As we have seen, prescribed AI abounds in today's video games. When we say "prescribed AI," we mean that the behavior can tend toward the predictable over time. It can be overused in gaming systems that are based on fairly inflexible rules and that predict a certain behavior that the developer has decided is challenging enough.

Prescribed AI can be augmented to give the appearance of adaptable behavior and sometimes even manifests true adaptable behavior, but often this is an illusion created by clever algorithms. The intelligence in these cases is only apparent at first. In time it becomes clear that 90% of AIs do not learn from the player's actions, and that a single, reasonably successfully strategy can be found that can win for the player every time.

In many cases this is not a bad thing; it is just that the AI used today can be improved without sacrificing quality elsewhere, while also offering more value to the player. Of all the examples we have seen in this chapter, the most successful ones have deployed advanced techniques, but without undue impact on the rest of the system. So the question is: Is bad AI better than no AI at all?

Judging by reviews, observations, and experience, it is fair to say that AI could be done better in the majority of cases, even with a few tweaks. That is not to say that the AI is necessarily bad—just that it could be so much better. Of course, there is also a minority of users that just get it plain wrong; the AI has been misunderstood, to the detriment of the game itself.

It is necessary to have a certain level of AI, but not always "intelligent" AI *per se*; we can often get away with the appearance of intelligence by using a combination of different algorithms, as we have seen in the previous examples.

In the end, we must decide what it is we are trying to achieve. Should the game be harder, easier, or just more natural? Or should it be different every time we play? Is the AI something to be turned on and off as a kind of difficulty ramping mechanism, or is it fundamental to the game itself?

A game without AI, whose behavior is based on strict pattern-following and rules—like the ghosts in *Pac-Man* or the aliens in *Galaga*—is still fun and challenging, after all. However, they can lack the longevity that games with high replay value through applied AI can offer.

SimCity, on the other hand, with its detailed AI-centric approach, is an example of a sim that will have lasting value and extreme longevity of game and interest.

So games with AI are not necessarily harder to play, but they can feel more natural, making them more immersive. The natural next step from realism is to augmented realism, where we can make the game more accessible to the player with additional AI for things like driving support or, for example, automated soccer player AI.

Balancing the AI

The balance between tough adaptability and predictable AI can be a hard one to strike. On the one hand, scripted behavior is often not quite good enough to challenge the player, and repeatable AI routines always tend to perform identically. This can make the game slightly boring, especially if the behavior remains constant throughout. However, adaptive AI, combined with augmented environmental senses, can be impossible to beat. The perfect opponent is as frustrating as a poor playing experience (unless the player specifically chooses to play that way).

Perhaps balancing AI is a question of allowing it to be turned on and off in the same way we can turn other in-game features on and off, such as steering and aiming help. This would at least enable the flexibility for the player to choose the experience that he would like. It could be combined with extensible AI scripting to allow player to dumb down or smarten up in-game NPCs, as desired. If we had a scripting model in which the game could choose smarter or dumber versions of itself dynamically, we would be able to adapt to the skill level of the player in a fashion that would provide a balanced experience, without the intervention of the player.

AI and A-Life in Video Games

The next step is to combine these techniques with the notion that A-Life allows a flexible logic that mimics patterns found in nature. This can be used to provide a playing experience that is often touted, but that rarely manifests itself.

In a sense, it is the combination of "modeled instinct" and reapplication of knowledge that takes the best of AI theory and augments it with some new ideas. It is important to understand that the foundation of A-Life is the AI; A-Life is just a way to express the AI in a more natural embodiment—and in a way that we can achieve much more with very little increase in overhead.

For example, A-Life makes mistakes naturally, while AI has to be programmed (scripted) to make mistakes. We saw how sometimes the result of an AI algorithm must be augmented in order to address the "perfect opponent" syndrome. But if we use A-Life techniques, this becomes part of the system modeling that creates the in-game behavior.

A parallel can be found in the creation of electronic music using MIDI or even good samples. Instruments that have a natural resonance, such as guitars and drums, sound unnatural when produced electronically. This is usually because each time the sample is played, it produces exactly the same waveform. Real-life instruments have variances in their sounds due to the immediate environmental conditions and due to imperfections in the metals/wood that the instrument is constructed of—imperfections that give each guitar/drum a special resonance of its own. No matter how badly it is played, a band playing a rock track will sound like a band playing a rock track, but it will still sound better than its electronic equivalent unless the technician has taken time to tweak the mix to make it sound more natural.

Some AI in games is bad because a perfect response is produced each time. Conversely, when that response has been badly implemented, we get the imperfect response every time; and it is all the more noticeable because it is persistently repeated.

We cannot just dumb it down because that will lead to mistakes. On the other hand, if we leave it intact, it will be either too tough or too patterned. Some real (or observed) life should be mixed in to break up the repetition and also keep the AI routines from making too many silly decisions.

In the end, the rules of the game must still be respected, so we need to make sure that AI is used to enforce them, with A-Life used to generate behavioral patterns within those rules. This is why we need both AI and A-Life and why AI is an important building block in the process.

As we shall see in the next chapter, A-Life provides a naturalness that can make it applicable in many different areas—not just the game itself, but in development, testing, multiplayer design, and continual evolution through mimicry. A-Life is both a tool for ensuring that the system is well developed, implemented, and tested, as well as a paradigm for creating in-game behavior. It is up to the developer to decide exactly how much A-Life he wants to use or how little he has resources for.

References

[EDGE01] Review of *Transformers: The Game* in *EDGE* magazine, September 2007, No. 179, p. 90. [GAMEAI01/02] http://www.gameai.com/games.html *Close Combat 2* section, quoted e-mail from AI developer John Anderson

[GAMEAI03/04/05/06] http://www.gameai.com/games.html *Interstate 76* section, quoted e-mail from AI programmer Karl Meissner of Activision

[GAMEAI07] http://www.gameai.com/games.html *Battlecruiser : 3000AD* section, quoted e-mail from developer Derek Smart

[GAMEAI08] http://www.gameai.com/games.html *Age of Empires I/II* section, quoted e-mail from Dave Pottinger of Ensemble Studios

[GAMEAI09/10/11/12] http://www.gameai.com/games.html *Close Combat* section, quoted e-mail (reprinted on GameAI.com with permission from Atomic Games) from developer Gary Riley

[GAMEAI13/14/15] http://www.gameai.com/games.html *S.W.A.T. 2* section, quoted e-mail from Christine Cicchi of Sierra FX

This page intentionally left blank

CHAPTER

3

USES FOR ARTIFICIAL LIFE IN VIDEO GAMES

In This Chapter

- Modeling Natural Behavior
- AI Techniques in A-Life
- Using A-Life in Video Game Development
- A-Life in the Design Phase
- A-Life in the Development Phase
- A-Life in Video Game Testing
- Post-Development A-Life

This chapter will outline ways in which a game can be enhanced by the addition of artificial life techniques that cannot be easily or efficiently produced by the AI-style implementations covered in Chapter 2. This is not to say that we will discard AI in favor of A-Life; instead, the AI routines will be extended to produce A-Life behavior.

In essence, this chapter sets the stage for the rest of the book, as we tackle the main areas in which A-Life and AI techniques will be applied to make video games more enjoyable and simulations more realistic. You should now be able to spot A-Life's advantages over basic AI within game design/development, as well as understand the importance of these techniques.

As in Chapter 2, those games that have used A-Life, sometimes with varying degrees of success, will also be illustrated. Suffice to say that A-Life has not been fully realized in video games, but the future is both interesting and challenging. Processor power and resources are increasing, and players are not only more demanding, but also more sophisticated.

Uses for A-Life as other links of the tool chain are also discussed, such as the ability to include A-Life techniques in the design, development, and testing phases of video game development. These areas are often neglected in an analysis of AI and A-Life in video games, but they can prove very fruitful—even more efficient than using human resources.

Finally, the post-development phase is discussed. There is particular emphasis here on the ways that A-Life can provide longevity in a game's shelf life due to the unpredictability of A-Life algorithms, as well as the opportunity for players to extend the game themselves (mods) and share those extensions with other players.

We should not forget that one of the barriers to using A-Life in game development is the aspect of quality assurance and the inherent unpredictability that A-Life can bring to a game. While it is true that unpredictability can enhance the AI behind a game, it can also be its downfall. Not being able to predict the behavior of a system causes concern for QA departments, as they cannot be sure that the game will not, when it is deployed, do something unexpected in the field.

Of course, we deal with this. Equally obviously, we try to make sure that the reader understands the steps that must be taken to limit the excesses of a system that is governed by unpredictability. It will be the strength of these limiting processes that keep a constant watch over the game universe that are responsible for making sure that the worst-case scenarios are not realized.

However—and this is an important point—it is undesirable to manage the system to the point that the unpredictability and naturalness of the A-Life is controlled out. While it is a good thing to recognize that there are QA issues to answer, we cannot let them dissolve the gameplay to the extent that the A-Life might not have been there at all.

Upcoming title *Grand Theft Auto IV*, previewed in *EDGE* magazine in April 2008, is a good example of some natural, managed unpredictability. As it stands, there are occasions where downloadable content can lead to random side-adventures that do not disturb the fabric of the story arc woven into the fabric of the game.

The emphasis is that these things happen in real life; consequently, there is a place for them to happen in video games.

In video games, A-Life uses AI to imitate lifelike behavior; it is the cooperation of individual elements that allows behavior to evolve in a realistic fashion. These elements rely on an understanding of AI.

Bear in mind that small AI/A-Life rules, when combined, produce big results. The union of AI and A-Life is almost always *more* than the sum of its parts.

In addition, genetic A-Life techniques can be used to combine behaviors and parameters, "breeding" better results. These include all manners of techniques that will be covered later in this book; for now, we just need to understand that these tools exist and that we can use them.

Before we look at how all of this can be achieved from a theoretical and applied standpoint, some minor misconceptions should be cleared up—principally, the notion that natural effects equate to unpredictable results; in other words, if something is unpredictable, then it appears natural.

However, most natural behaviors are based on a combination of applied rules at varying levels. For example, to some people a mountain range might look fairly random, but a geologist will tell you that the range is the result of various, predictable processes.

Simply modifying AI with some pseudo-random number-generating routines, while effective under limited circumstances, will not produce A-Life on its own. We still need to add artificial genetics and other techniques to create natural effects in video games, although these will no doubt include some randomness to mimic chance in real life.

The underlying process, however, is created by a continual application of interlocking rules that each contribute to the overall effect. These rules should be aimed at producing natural behavior that is sometimes predictable (birds do not often fly into walls …) and sometimes not (… but they do fly into walls sometimes).

Modeling Natural Behavior

So we want to use AI in video games, and we expect it to generate behavior that is both natural and predictable enough to be reliable; but one issue is that it can turn out to be merely predictable, or even unstable. This is a tricky balancing act, as was noted in Chapter 2.

Partly, this is due to the implementation of AI as a reasoning system. It basically processes information and makes a decision. Since the rules are often inflexible, the behavior will also be inflexible—hence, the player might recognize techniques that are guaranteed to beat the computer every time.

Some of this can be passed off as the player learning the opposition's weak points, which also happens in the real world. But if this happens too often, then the result is less than satisfactory.

Some stochastic techniques can be added to introduce variations (see the "Hopfield Nets" section in Chapter 2), which will instill a variable level of chance into the

decision-making process. For simple systems where there is a limited number of variables to manipulate, this will work.

This approach is, however, just a workaround and is not a real solution to the problem. Granted, it is computationally cheap, but in very complex systems, the effect may well prove to be negligible. Sometimes the sheer number of variables will produce behavior that is different due to varying circumstances.

For example, in complex simulation and strategy games, the weight of the decision-making process will often be greater than any false randomness aimed at manipulating choices made by the computer. This is usually because the events are always the same; they follow a path through the game, and the same behavior is likely to emerge each time. Put another way, when playing a game that has a goal that is always the same, the player is likely to try and solve it in the same way—or he finds a way that works.

This makes it all too easy for a player to find a "winning" technique and reuse it in different situations, always getting the same result. Introducing randomness into this kind of system does not make sense. It will frustrate the player and ultimately make the game less fun and therefore less successful.

As was mentioned, part of the issue is that players naturally adapt to different behavior patterns, so even if we change the behavior slightly, it will not be long before the player works out a new strategy—which he will use repeatedly—that is, until the machine discovers what is going on.

There are two options here: change the behavior stochastically or use another form of AI. We can counteract this with learning algorithms so that the machine adjusts itself intelligently to the player's new strategy.

We saw some of these techniques in the previous chapter, but they need to be combined with alternative algorithms to increase the naturalness of the end result. Rather than just introducing randomness, we want to instill flexibility.

Adding the ability to flex the rules that generate the behavior permits "better" AI that borders on A-Life. We can also change the rules dynamically, rather than just change the outcome, which will create richer behavior. For example, if we have an action game in which an FSM generates the behavior of an enemy unit, the options chosen or the rules used to apply the FSM can be adapted.

Adaptive behavior borders on A-Life, but there are additional techniques that can be used to bolster this simple approach. The goal is a richer experience for the player, and with little or no computational overhead beyond what is necessary for the game AI. Otherwise, we are taking processing resources from other aspects of the game—a game that might be a very large and complex system.

Traditionally, AI is computationally expensive, especially in problem domain searches that seek to find a best (most successful) fit or the most possible solution from a known set. For example, the A* and alpha-beta pruning with *minimax* algorithms work with search trees of the whole or partial problem domain.

We can reduce the impact by sorting the search space or ignoring those items known to be irrelevant, but these approaches are still expensive. Existing games use other tactical AI techniques that are equally expensive (such as testing all possible

unit moves) and are sometimes dispensed with in favor of strict rule-based but predictable move-generation algorithms.

In addition, scripted models, FSMs, and other interpreted instruction-based approaches are also computationally expensive. They have to check for execution problems to avoid getting caught in behavioral loops and must validate the decisions so that the result is still in line with the game rules.

If not done well, this can lead to poor implementation. There are many examples, from *Grand Theft Auto* to *The Getaway* and *Half-Life*, and even more recent titles such as *Transformers: The Game*, where possible shortcuts in the AI leads to behavior that is less than lifelike and frustrating for the player.

Part of the issue is that AI algorithms tend to be iterative or recursive, both of which use resources heavily. Even an efficient algorithm is still very resource intensive, given the nature of the problem domain. We are simulating behavior from both sides—the computer and anticipated player behavior.

In certain AI algorithms, flexible rules will provide a way to save some overhead, especially in result checking. If we know that a specific rule generates behavior that is correct within certain bounds, changing it so that a new result is still within those bounds requires no extra testing.

With complex gaming environments that have many FSMs in a given network, there is a point at which it becomes impossible to manage. The designers may try to cater to eventualities and create rules that are incredibly complex. Eventually this produces a system that is hard to manage, control, and test and might also be less efficient than a system with fewer or flexible FSMs.

All of this is geared toward several key A-Life principles—producing better initial behavior, adapting considered behavior to fit new situations, and providing a variety of behaviors that make a more natural model of the perceived system. In video game design and development, at the core of this is adaptive and emergent behavior.

Adaptability

Adaptability is the ability of the system to change its behavior with respect to the player's behavior, with the goal of creating a better player experience within the game universe. The adaptability paradigm needs to take into account the relative success or failure of all entities within the game universe to deal with changes brought about as part of the playing experience. We have to consider the machine-controlled entities as well as the player-controlled entities in order to obtain the whole picture and create better gameplay.

Emergent behavior and adaptive techniques can be used to make a poor system better or a good system worse, dynamically. By "good" or "poor" system, we mean the level of intelligence that the system should display, rather than the success of the system itself.

In this case, the intelligence that we want to adapt is the measurement of the relative success of a static algorithm. We might want to make an otherwise near-perfect algorithm, exhibiting behavioral capability (skill) that is equal to or superior to the player's own, behave in a less skillful or intelligent manner.

This is rather like offering the player the choice of which level to play at, but without presenting it artificially via a menu. The system will *detect* player capability and adapt itself accordingly.

Taken one step further, it provides a good way to counteract the player's killer strategies. But it also opens a door that has a potential downside: The player might be able to manipulate the technology and encourage artificial downgrading of the intelligence in order to make winning easier.

For this reason, care needs to be taken when instilling adaptability; remember, a fair level of play should neither overwhelm the player nor underperform. In the past, simple, easily manipulated mechanisms have been used, especially in racing games, where the vehicles are artificially slowed down to cater to below-par players. But this can lead to easily completed games, rather than novice-friendly playing systems.

The adaptability of the system is linked to an abstract notion of fun—the player must enjoy the game, and we should only add A-Life and AI to the system that is in line with that notion. For some players, that fun equates to a higher level of challenge than for others. Some would rather complete the game easily, yet others relish the challenge, and that balance is as much a part of the game design as addition of AI and A-Life.

Since rule-based AI is not good at adapting, we need to augment it with technology that extends its capability to adapt in many different ways. The innate inflexibility of rules can mean that AI will have a hard time adapting to playing *styles*, whereas it might be easily toned down to compensate for playing *skill*.

Most AI systems are rule based—including FSMs, expert systems, and even fuzzy logic—and those rules are often implemented at the design level. However, neural networks and other trained systems exhibit fairly flexible behavior and can be augmented with unsupervised learning algorithms to prevent rogue behavior.

An expert system, for example, can be made more adaptive, but it needs to encompass learning and fuzzy logic to ensure that it develops along with the player. Rule interpretation, rule set triggers, and outcome analyses can all be manipulated in response to the success of the system or the player.

True adaptability has been tried with some success, though some games that have set out to use it have failed to get past the design stage. This is often because the development schedule tends to narrow at the end of a project, which is when most of the AI is traditionally implemented.

So why isn't AI delegated to an earlier spot in the production cycle? Partly, this is due to the belief that AI needs a playground in which to be developed. That is, unless the game mechanics and universe exist, the AI cannot be tested, because it has nothing to act on.

A-Life challenges this because, in the same way that AI is top-down, A-Life is bottom-up. We can build A-Life into a game from the start because it is part of the fabric of the universe and not just some mechanism tacked on at the end to improve an otherwise bland playing experience. This is possible due to A-Life's unique use of techniques—such as genetic algorithms to help adaptability evolve over time without excessive processor overhead.

Adaptability with Genetic Algorithms and Genetic Programming

A genetic algorithm (GA) uses a representation of the solution (as parameterized data) and applies it to a given algorithm (behavior). This provides a flexible rule-based approach that can naturally propagate the best application of that rule. We do not need to search the problem space to find the best fit, nor do we need to prescribe the solution; we just need to provide a set of possibilities and then let the GA select an appropriate set of values. Due to the way in which GA is applied, a natural randomness can also be introduced to help this along.

By a similar token, genetic programming (GP) can actually alter the method by which the steps of a program or script arrive at a solution and apply its selection to generate appropriate behavior. Again, we only need to supply some examples of possible behavior—a collection of alternatives—and let the best, worst, or most interesting behavior emerge by itself.

Make no mistake, however. Identifying the solution (for GA) and the parameters that feed into it or identifying what steps should be involved in the solution (for GP) is an important part of the paradigm. If any developer seeks to use the two together in the way that is described here, he must do it within strict guidelines, because the introduction of a small, seemingly insignificant deviation can have disastrous effects on the "fitness for use" of the resulting evolved algorithm.

We also have the leeway to apply standard copycat routines that mirror natural behavior. If we can recognize a pattern, create a neural network that is self-training, and then extract that pattern, then we can reproduce it.

In other words, suppose that a player has created a combination of fighting moves that seems to work every time. A virtual DNA string can be created and passed through a movement-generation process to mimic that behavior against the player. It can also be used as a starting point for GA or GP techniques.

So whenever genetic algorithms are used, we build in the ability to adapt over the long or short term, such as within a play session or over a longer period of ownership. For example, games like *SimCity* and *Black&White* are designed to continue over months or years of evolutionary behavior.

Adaptability with Traditional AI Models

In terms of traditional AI models, we can either implement changing scripts or changing parameters with the same script. This represents the key difference between GA and GP techniques.

Genetic algorithms tend to be more flexible, but they can also be slightly dangerous if not managed properly. Occasionally, rogue behavior can creep in, which needs to be counteracted—hence the use of supporting AI algorithms. The trick is to know how this rogue behavior can be spotted and intercepted before it detracts from the gaming experience.

Learning over time becomes easier when the algorithms can adapt their behaviors, assuming assuming that learning is only a version-advanced adaptability. In essence, a combination of neural networks and genetic algorithms can be applied to help a system learn.

For example, starting with a large population of possible solutions, we try each solution in turn and eventually train the neural network to select the correct one. This is basic pattern-matching, which can also be implemented by using genetic algorithms.

To do so, some way of measuring the outcome is needed, and then a new population is derived from the most successful solutions in the population. A neural network, however, might prove to be more adaptable and useful. A single DNA strand might not be able to cope with as many situations as a neural network; many strands would be needed.

The new population that results from genetic combination will include solutions that are better and worse—all adapted from the parent solutions. So if we select different ones for different behaviors, we achieve essentially the same behavior as can be produced with a neural network.

In both cases, our goal is a solution space that will quickly optimize toward the best approaches for different situations. The population is, in a sense, self adapting and it learns.

Selection and evaluation, as well as some application of the result of the genetic algorithms, will still use traditional AI. The AI routines need to marshal the effects of A-Life in the same way that a scripted AI interface was used in Chapter 2, while also encoding the actual implementation of actions in the core software. It is also possible to keep the algorithms the same and just vary the input, output, or weight data.

In the previous genetic algorithm example, the solution (steps required to achieve the end result) was varied, but it can also be approached from a slightly different angle; we can vary the parameters instead. In this way, we can keep the same algorithm and just mitigate the behavior by using the same analysis, evaluation, and application.

For fuzzy state machines, this could be as simple as varying the threshold values that generate or manage the behavior. Each variation would produce customized templated behavior—perhaps more or less effective than the desired result.

In this way, the behavior of the FuFSM network can be varied quickly and easily, but all the behaviors, individually, are still similar. They share the particular problem-solving algorithm, but its details are applied in a variety of ways and/or to various extents.

There are two advantages in this for video game applications. First, they are easier to replicate, since they are reduced to a collection of parameters. They do not contain logic, so broken scripts or algorithms during breeding, mutation, or direct replication is less likely. Therefore, the implementation is more efficient and requires less testing.

Adaptability and Quality Assurance

In theory, at least, if the extremes of the parameters that are being used to describe the solution space produce results that fall within extremes of acceptable behavior, then the run-time variation of those parameters will be safe. Testing is still needed to ensure that the emergent results of the system still fall within bounds of acceptability, but the lack of logic makes the solution more robust.

This leads to the second advantage: There are added safeguards against "odd" behavior when the algorithms are kept more or less equivalent in the results that they are expected to produce. Here's one example of where the results of two superficially equivalent algorithms vary wildly:

2 * x 2, 4, 6, 8, 10 … for values of x = 1, 2, 3, 4, 5 …
2 ^ x 2, 4, 8, 16, 32 … for values of x = 1, 2, 3, 4, 5 …

Although only one symbol is different (* versus ^), the results are not equivalent, so we cannot say that the variation in the behavior is similar. On the other hand, if we hold the symbol static, representing the algorithm, but change the first parameter (2), then the results might be more acceptably equivalent.

The trick is in making sure that algorithms are used where two or more controlled variations produce results that are acceptably equivalent within the confines of the game universe. Each variation in the behavior will be similar, but different; imagine, for example, using genetic algorithms to create a fighting army animation.

If the genetic programming approach is used, then we can allow variations on the same theme; but if we use the genetic algorithm approach, there is a risk that the behavior of an individual entity will become too different. Unless specific safeguards are built in, the result could be disastrous, such as entities attacking their own side.

This also demonstrates how A-Life can comprise a good mixture of new and old techniques. New techniques will be more or less original to the game, and when mixed with reasonably well-understood paradigms for manipulating them, an unprecedented level of feedback for learning and adaptability is instilled into the system.

Notes on Performance

This is all very computationally expensive, which is why rule-based AI is still the most common, with some stochastic augmentation introduced to try to counteract predictable behavior.

Genetic programming is slightly cheaper because the core algorithm remains the same. Only the parameters (data) are varied to get the desired result. However, depending on the problem space, it might take more iterations to achieve this end.

Besides the algorithms themselves, there are some other areas that will affect the performance impact of an adaptive system. The size of the solution space will directly affect the time taken to find a workable possible solution. The more cases that the system has to analyze in finding one to use, the longer it will take. This is in common with virtually all the AI and A-Life techniques we discuss in this book.

The checks that are needed to locate and eradicate rogue behavior will also impact the performance. The more checks that are deemed necessary, the less processor time will be able to be dedicated to the adaptable behavior algorithms themselves. This is linked to the flexibility, of course.

The more flexible a system is, the more safeguards that may be required to limit any rogue behavior. In some cases, a flexible system might be reasonably

self-managing, and if such a system can be used, so much the better. In the end, however, the performance will be a crucial deciding factor in whether to limit the A-Life implementation or realize its full benefits.

AI Techniques in A-Life

As was mentioned, AI is used in A-Life to provide the actual behavior and to mitigate the effects of the A-Life paradigms. For example, if we have a script that allows a set of five commands, genetic algorithms can be used to combine these five commands, which are then executed by the AI routines.

Rogue behavior is then checked for by a specific set of routines designed to spot and deal with them, and the result is fed back into the system. This is classic AI-style feedback (based on work done in observing learning and behavioral modification in animals).

So this is the way we will be applying A-Life. There are essentially two ways to use it:

- Window dressing: which has no functional purpose but looks pretty, or
- Simulation: which has an impact on the flow of events in the game universe.

Both add value to the game, and the player's experience is enhanced. When used as window-dressing, there are very few consequences (like the flocks of birds in *World Rally Championship II*), but for simulations, there could be consequences for the player; the game will be tougher or easier as a result.

Most A-Life starts as a simulation because resources are limited, and there are other ways to provide immersion without the need to create in-game life. As processors become more powerful, and as more subsystems (graphics chips) share the load, window dressing will become more common, increasing the playing experience.

Take, for example, the dynamic modeling of water. Programmers have only recently begun to properly simulate water, because previously it was out of reach both in rendering terms and in terms of the processing power needed. But times (and processing power) have changed; what was once merely impressive is now correctly simulated water. A-Life also fits into this category in special cases, to an extent, and serves for simulations, rather than just visual or immersive effects.

Both kinds of A-Life are aimed at creating results that are more than the sum of their parts. When taken individually, a lot of A-Life algorithms and behavioral models are not very impressive, but the weight of the entire population and all the parts working together create something special.

A-Life uses all kinds of AI techniques as building blocks—such as neural nets, finite state machines, fuzzy logic, and daemon organization—to create the special effects that we call "artificial life." Daemon organization, in this case, means allocating specific algorithms to specific entities (daemons), which manage their interaction with each other and the game universe while they carry out other tasks that are predetermined. So A-Life can be seen as a way to organize and connect AI algorithms to provide emergent behavior in video game technology (among other things).

Consider this good example: In certain movement algorithms, it is sometimes hard or expensive to evaluate all possible combinations. We can use straight AI to try and do so, or we can apply A-Life principles to model several solutions and recombine them to find a good alternative. The result might not be the best or most efficient solution, but it will be a step in the right direction. The failures can be removed from the population in favor of solutions that progress our cause.

In certain movement algorithms, such as a flock of birds, the behavior can be modeled individually, each one programmed to follow a certain path from take-off to landing. When the models work together, the effect is a flock of birds, but this is an inflexible approach. There is also a limit to the number of birds that can be encoded, and we cannot add to or remove from the set.

Suppose the flock gathers under different circumstances. What happens if they encounter an obstacle? If we have not specifically programmed each individual bird to deal with these new conditions, we will have to create new paths for the flock.

However, using A-Life techniques, birds that all have the same basic behavior (clones) can be made to act like finite or fuzzy state machines—and very cheaply. The result is a true flock. It moves as one unit wherever necessary, not because the "flock" has been scripted to behave in this manner, but because each bird is following similar behavior.

A bonus is that the flock can adapt to new obstacles without being told. They will naturally flow around obstacles as the birds follow their behavioral patterns. The key is in how they are all similarly modeled to account for their surroundings.

Behavioral modeling aspect is important because it allows us to connect pieces of AI and rule-based systems together in a way that is a good, natural solution to a given problem. Without this, the end result is often inadequate. Useful shortcuts can also be taken by observing natural phenomena and creating self-organizing systems from the bottom up. Thus, that same phenomena will be displayed, and we are saved having to work out the behavioral pattern for each individual.

In flocking algorithms, each individual bird in the system is modeled in relation to its peers. Each bird knows how far it should stay from its neighbor, how it should react to obstacles, whether there is a natural leader (and whether that changes), and so on. When the population is processed, one algorithm (or state machine) and a collection of data describe each object within the set. We can either store a collection of parameters that change with time or create a collection of objects. The result will be the same either way.

What is important is that behavioral aspects, such as avoidance, will propagate through the system in real time. The first bird in our flock will avoid the obstacle, and birds in its vicinity that encounter the obstacle will follow either the first bird or another bird or just avoid the obstacle as best they can. The best decision will change, depending on what the highest priority input is. Closeness of the obstacle, for example, might override the propensity to follow the leader if doing so would bring it closer to the obstacle.

This sort of behavior is witnessed in real life; "thinking" animals and AI and A-Life paradigms are similarly inspired. Humans, for example, were the inspiration for neural networks, which are modeled according to the way that our brains work.

A-Life for Simulation

Most of what we've covered so far relates to using A-Life as a kind of advanced animation technique. We have not placed any real effects on the player or game universe. In fact, the flocking routine was developed by Craig Reynolds, an animator by trade, and is used frequently in animated, computer-generated films.

As some background, Craig Reynolds has worked in a variety of companies, deploying A-Life in different games. These include 3D animation scripting systems for Symbolics, Inc., where he also researched the bird-flocking algorithms and the behavioral models used in *John Madden Football*, while at Electronic Arts. Most recently, he has been employed at Sony Computer Entertainment America, involved with autonomous character technology for PlayStation 3.

Currently, A-Life is used for simulating behavior that has an effect on either the game universe or the player's status, directly. The essentials of these techniques, governing AI with A-Life, were covered at the beginning of this chapter. However, if we consider, for example, the simulated people and traffic flows in a game like *SimCity*, we become aware that there is more to A-Life than mere window dressing; it can be used for much more substantial input. The Sims in *SimCity 4* have "intelligent" opinions that are affected by the game, which is part of the entertainment. They are affected in a lifelike way by everything they see or experience within their little part of the game universe. They can also affect each other, wherein their states become contagious.

Each Sim can have an effect on the environment, which feeds back to other Sims. If one member of the population exhibits antisocial behavior, such as littering, then this will have two effects. The first will be that the others observe the behavior, and it will affect them directly, and the other effect is that the environment will change.

The "butterfly effect" states that even the subtlest of variations will have an effect on the stability of the (game) universe. The littering Sim might cause a chain of events that trickles down through the game universe, like ripples in a pond, causing effects in different areas of the system. The compounded variations become amplified as they interact with other inputs. The littering may just be symptomatic of a greater malaise, such as the influx of graffiti, which leads to the eventual degradation of the entire neighborhood. This is conjecture, because in *SimCity 4*, the only outward indication of the degradation of a neighborhood is the graffiti, which might be a symptom or a cause. Just from playing the game, it isn't possible to tell.

The player, in the case of *SimCity*, bears the ultimate responsibility for bringing the neighborhood back on track. This is also part of the attraction of *The Sims*, in which these compound effects are much more marked. So the stability of the game universe is affected by these techniques, which can manifest themselves in a number of ways:

- Flip-flop: Models alternate between two cyclical states.
- Stabilization: The elements settle down.
- Predictable behavior: Elements exhibit a non-flip-flop change between states.

Any of these effects could be good or bad. The player might be required to cause certain states before being able to progress in the game. Or the system might need to spot one of these states and then combat it by introducing an unstable element, such as mutation, into the system. This can be done by using pattern-matching and neural network-style AI algorithms to isolate stability and then compensate for it. As an alternative, the mutations can be introduced at random to ensure that stability never lasts very long.

This all deals with visible simulations that are part of the actual game universe. Simulated A-Life populations that are never seen can also be implemented within the game—populations that are designed to exist only in a simulated version of the game universe: a simulation within a simulation.

For example, rather than evaluating a series of solutions by taking each one to a conclusion and then comparing the outcomes (as in alpha-beta pruning and minimax algorithms), we can treat the moves as genetic strings and try to breed a good solution internally. Therefore, some or even all the best solutions might not be covered, but at least solutions that have a chance of being useful after several generations would be created.

This *might* be more processor efficient in certain circumstances, as well as produce more interesting gameplay. Since coverage is not complete (or even close), unless we allow a large number of iterations (generations), this kind of approach is not useful in games with a discrete solution space that is easily modeled. This approach might not be ideal for chess, say, but it could be quite good for strategic or simulated AI-based games such as *Civilization*.

Using A-Life in Video Game Development

Now that we've described the forms of AI that are embodied in A-Life, it is time to look at some of the places where A-Life can be used in video game development. Our interest is not only in A-Life's use from the game player's perspective, but also A-Life as a technology that enables video game design, development, and testing.

A-Life should be in the game development cycle from the bottom up; the decision to include it in this way should be assumed as part of the project creation. In other words, it is better to embrace A-Life from the onset in every nuance of the game development process than it is to have one developer adding A-Life on at the end. There is then an intention to use A-Life technology in the game and development process. Despite this, A-Life will probably still end up as a low-priority concern in a system that includes game mechanics, artwork, sound, graphics processing, and universe management, but the effects of A-Life will be visible throughout the finished product.

Readers will be aware that they need to have some grasp of the *possibilities* of A-Life before they start deciding what the effect of A-Life on the game should be. This means that they need to become conversant with the possibilities of A-Life (or have someone who is an expert in A-Life on the team) so that the approaches can be selected and the intention instilled in the project plan.

In part, the purpose of this book is to show where AI and A-Life are currently underused. Our goal is to give some of those areas that are lagging behind a boost and prevent AI from becoming more than an afterthought that is simply bolted on at the end of the cycle.

The reality is that all areas of game development can use A-Life, and depending on the game, its use should be exploited in each discrete area. Some games will not lend themselves to more advanced A-Life, but most will benefit from the technology in some surprising areas.

Typically, we think of the game itself using A-Life simulations only during play sessions, such as for controlling enemies, planning and strategy, and running the simulated intelligence and behavioral parts of the system. But we can also deploy A-Life techniques in most other areas of the development cycle, from design to testing, with the key advantage that created routines can be reused to make the game better. Another advantage in having A-Life as an early part of the game design is that it gains acceptance from the start, which lessens the possibility that it will be discarded at some point due to external influences.

A-Life can also be used for prototyping behavior in the design process to prove the game mechanics within a limited implementation. For example, prototyping might require incomplete game mechanics, where broad concepts exist, but no interface. At this stage, as long as the rough game rules are in place, we can begin to experiment with A-Life to find the best way to position the mechanics, as well as help form the standard intelligence that will be shipped with the game. This can then carry on into development and help with scripting and so forth.

Similarly, A-Life manifests itself in testing and simulating interactions with an eye toward finding possible bottlenecks and unforeseen behavioral and player actions within the game universe. In a sense, we begin to simulate by using A-Life as a game within the game. Of course, the nature of the game and its genre will dictate whether these tactics lend themselves to using A-Life, just like certain AI algorithms are not used in some games because they would not advance the project.

At the other end of the spectrum, there are games based around the simulation of life within the computer, such as *Black&White* and *Creatures*. For these games, A-Life is an integral part. They make deeper use of the technology and have more processing resources devoted to it.

Within a game, A-Life is also built upward from small units in order to allow behavior to evolve and emerge. The player then interacts with these units as a whole (being, entity, or creature), which creates its own life from the building blocks.

The A-Life system is usually primed as a result of analysis and design. In *Creatures*, this takes the form of some basic chemistry principles, but the interaction between the player and his "pet" creates the game's appeal and is the basis for the game itself.

The same ability of A-Life to combine routines enables designers to drop new objects or entities into the game, which previously were not envisaged. This can happen at any stage of development, or even in post-development, such as objects supplied to players of *The Sims*. This mechanism can then be used to change the game's underlying A-Life. For example, remember our "smart terrain" AI technique,

and objects that announce their effects on the universe and entities within it. To do this, we need an information domain that is well defined, although we can also plan for the additional of new items within that domain. In other words, if we want to add an emotion, like pain, to a system in which pain has not yet been defined, then the information domain must be made extendable.

Augmenting the information domain in this way means that the building blocks must be defined; otherwise, we can only add on by changing the underlying program. Therefore, the granularity of behavior is dictated by the granularity of the information domain; fewer categories means less control over behavior.

This is another reason why A-Life and AI should be included from the beginning of the project in order to realize their full benefits. With this in mind, let us first look at how A-Life is deployed in the design phase.

A-Life in the Design Phase

The advantage of A-Life over traditional AI is that we can use it before the game universe actually exists, because it will be part of that game universe. This is due to the fact that we build A-Life from the ground up, just like the game universe is being built from the ground up. AI, on the other hand, is often used to manipulate existing behaviors from the top down, so it is harder to use in the design phase (but this is still possible, as long as the approach is correct).

To have A-Life help us design the game universe, we need to establish simulations of rule-based worlds that are populated by pseudo-players, and we watch the outcomes of those interactions. At this point, there is no interface—no playable game, as such—just the rules that are designed to manage the mechanics of the game universe. In an FPS, for example, this might be as simple as interaction rules based on proximity triggers.

The environment is modeled by creating a very simple representation, within which it is possible to observe the evolution of different strategies for, say, wall following or grouping. These strategies are designed to be lifted and used as actual AI rules. Other genres based on rule interactions, such as simulations, can be even simpler; we just want to find a well-balanced combination of game rules in order to prove the concept.

This enables us to test approaches before any serious coding has been done. Of course, the germ of the idea is still needed, as well as a good plan for how it will be implemented in terms of the problem domain and the information within it. But this will need to be done regardless of whether AI or A-Life is to be used.

Any tools that are built to model A-Life can also be reused during game development itself, and even as part of the underlying game engine. In a sense, from an A-Life point of view, the design phase becomes part of the development process. The basic principle is that the behavior should be implemented first because it is the hardest part to get right. If we can mold the correct behavior of independent components, then chances are that these components (and their associated behaviors) can be combined and engineered to form the complete effect.

This is where prototyping comes into its own: being able to develop a routine in a restricted fashion and then test its validity before it is set in stone. Already a valid software engineering technique, A-Life simply adds a new dimension.

The ability to prototype designs will help in developing the right balance. This can be done without A-Life techniques, but it might not lead to the same results. A-Life, via recombination and large test populations, helps us to maximize coverage with minimum effort.

A-Life in the Development Phase

Once the game rules have been designed, along with some basic behavioral building blocks, we can begin to apply A-Life again—this time in the development phase. During the development of the game, the individual behaviors of the entities will be created.

All kinds of other developmental efforts will take place, as well. But from the game universe, management, and entity behavior points of view, A-Life can help us create viable coded examples that would otherwise have to be built manually.

As an alternative to developing these behaviors in the source code, A-Life theory can be used to create a rich scripting language. If we plan to leverage A-Life when creating behavior, then our entity-scripting language will need to be implemented to support this.

If this sounds like a waste of development effort, bear in mind that the A-Life end result might be reused in the engine. Even if the script A-Life must be interpreted and compiled into the final product's source code for performance reasons, it is still a worthwhile additional development task.

Much of the behavior of any video game is open to being scripted. The only real question is, at what level do we choose to do so? This is the granularity issue discussed in Chapter 2.

The scripting language itself will probably be based on AI techniques such as FSMs, which can be connected together to create more complex behavioral networks. Assume for the time being that only simple, single-level FSMs will be used. A single FSM can be turned into a set of adaptive scripts that emulate lifelike behavior, assuming that we have a rich enough language. We might, for example, choose to implement fuzzy FSMs that can selectively move from one state to another and trigger certain responses in the system.

We can also use GA and GP to create emergent behaviors that, if appropriate, can be reused in the game. In other words, genetic algorithms can be implemented above the scripting language to alter either the sequence or the extent of the events, and A-Life can be used to recombine the result.

A-Life can also help balance the game by building a population of scripts that will interact with the game universe on a number of different levels. Each one can be analyzed, along with the eventual result, and the rules tweaked until the development

team is happy that the correct sandbox for the game has been implemented. The result can be turned over to level designers and artists, who will turn this framework into an actual video game.

Scripting Language

The purpose of the scripting language in AI and A-Life is to provide building blocks for implemented behavioral routines. In the same way that the programming language is used to implement the game mechanics, the scripting language gives instructions for the implementation of behaviors, at different levels of abstraction. The individual instructions can be combined with programming language–style decision and repetition blocks to produce a sophisticated system for representing behavioral models.

AI implemented as sequences of finite state machines can be used to construct more detailed behavior, as was previously mentioned. Each script serves as a building block to instruct the game on how the entity is to behave. This includes flow-control instructions, as well as action, decision, sensing, and information functions that allow the FSM to interface with the game universe. Each script has to be able to determine a specific condition, test it, perform an action, and then move to a new state, usually in a top-down fashion.

Even where daemon techniques are used, where work is farmed out to individual entities in a network, there will still be some need for FSMs and language that supports decision-making, knowledge retention, and access to that knowledge. A network of cooperating daemons will process scripts independently before passing the result back to the controlling process.

Chapter 2 provided examples of behavioral scripting as used in several games, including *Half-Life* and *Baldur's Gate*. These all comprise flow control, access to game universe state information, and decision-making processes. One facet that they share is that they are simple to read and understand. In fact, the simpler they are, the better—both for A-Life and real humans—because this will make the scripts more efficient to use and execute.

As a simple example, consider building A-Life from small building blocks, such as a decision-making process that tests for a number of small conditions and performs actions accordingly. These will be implemented as FSMs in the scripting language, with each one responsible for some facet of a single entity's behavior. If all of these very simple constructs (perhaps a large number of them) are based on the same template, more variety can be introduced into the system as a whole. Smaller constructs gives us a finer degree of control. If each one performs just one simple action before moving to a new state, then we need to combine many constructs to get the behavior needed, but many entities can be produced with a wide variety of behaviors.

As soon as the constructs themselves become more complex—that is, written in a "proper" programming language—they become more difficult to implement, understand, debug, and use to create A-Life forms. The greater degree of control and flexibility allowed by a scripting language that begins to resemble a programming language, the more care that must be taken in its use.

In essence, a more powerful scripting language artificially restricts its usefulness in the design of A-Life by non-programmers. The balance to be struck is the simplicity (or elegance) of a scripting environment—language plus behavioral commands—against the flexibility to allow sophisticated models to be created.

In the design phase, it may well be that non-programmers (capable of scripting, level design, and so on) wish to instill certain behavioral models, but the QA angle becomes so important that it becomes as difficult as managing the programming part of the project. Indeed, because every script has to be carried out (interpreted) by a piece of underlying computer code, the more complex the language, the harder it becomes to implement.

So keeping it as simple as possible will make sure that it becomes more usable, and hence useful. At the same time, the easier something is to implement, use, and propagate, the more chances there are that it will be adopted.

Modifying Scripts Using A-Life

So we can use A-Life techniques to modify scripts that form the building blocks for behavior. In particular, we are talking about the application of genetic algorithms to create an adaptive system by which we create a pool of available entities for use in the system, some of which will be kept for the packaged result.

Further application of A-Life techniques can take this one step further and provide adaptive behavior within the game itself. For example, neural network-style AI can be used to govern the behavior and guard against possible deviations from the desired limits. Genetic algorithms can then be applied to derive more complex behavior, with newly scripted FSMs added either during the development phase or during the play session. For fewer deviations and a slightly less adaptive system, genetic programming with fuzzy FSMs can be used to keep the basic patterns but manipulate the chances for deviant behavior.

This is then taken a step further to improve the "gene pool" by recombining elements to create more and more effective examples. Traditionally, this might only be done for enemies that are physically close to each other ("mating"), but we can also cheat and use genetic material that is neither physically nor behaviorally close in order to create more variety.

Natural selection means that if an enemy is destroyed as a result of something that the player does, then that behavior is labeled as being an unfavorable plan of action, and it will not likely be recombined with other behavioral units. If it is not destroyed, then the material becomes available for recombination and can be handed to new instances of enemies (or even applied to existing ones) so that the game universe (or at least the enemy population) as a whole is "learning" from the player. This behavior modification is rather like the real-life equivalent of Darwinian selection.

There is a danger that, by training the enemies in this way, we will violate the underlying principle that is to let the player have fun. If the evolution is *only* based on player victory, then we will produce ever-harder AI. The intelligence will not rise, but the difficulty may become artificially high.

To combat this, we could use behavioral deviation levels to reward the player by more interesting enemy behavior. At the first level, enemies might just attack using one or two moves. At level two, they might add movement forward and backward, or blocking, and so on. This is a matter of game design and shows how tightly design, mechanics, and A-Life have to be integrated to make the game fun.

All of this can be accomplished by building in rules and by using a scripting language that is self-modifying (or at least can evolve). These techniques can be applied not only in the play session, but at development time, too.

For example, crowd emulations or army simulations can make good use of these A-Life paradigms. The sum of the parts is governed by a scripting interface that provides the general behavioral framework; each one is then modeled individually. Then, if two armies are pitted against each other, we might be able to breed a super-army by combining the behaviors of individuals. We might also learn why their tactics work.

These tactics can then be used to create default units by taking the most successful and creating templates for use in the game. In this way, A-Life can be manipulated to provide a way of simulating and breeding AI routines that are finely honed. Here again we have the choice to use the template to create new behaviors or use genetic programming to keep identical templates and vary the parameters/data that they process to provide variations.

One feature of modern video game development is that the more complex games become, the more variables there are to govern, and the less restrictive the universe is. This makes it hard to create behavioral models manually. Bigger games that are set in more complex universes (which have been made possible by recent leaps in technology) mean more information to process. Whereas humans can cope with fairly large amounts of data, "bad" AI is still a possibility, and/or practical algorithms become harder to derive. *Command & Conquer*, for example, has always been criticized for its pathfinding, as well as its tactical engine that doggedly follows goals despite obvious failures in the approaches taken.

So A-Life can be used in situations when human developers fail to see the woods for the trees. They are so close to the systems they are developing that sometimes it becomes hard to create behavior that takes into account real players' actions; it might not occur to them that a player will act in a certain way.

There are a few caveats, though. First, instilling A-Life needs to be done in *small* increments, or some of the benefits will be lost. Rich behaviors stem from many small effects that work side by side—not a few complex constructs with rule-based, inflexible reasoning systems at the core. Also, the prototype sandbox can be expensive in terms of development time. We need to make sure that we have an environment that is as close to the final product as possible, and some guiding AI must exist to make sure that the bounds are not overstepped. If both of these issues are taken into account, however, the results should be both efficient and effective, and the savings in development time could be extremely beneficial.

Building Scripts Using A-Life Techniques

In the previous examples, we looked at using A-Life techniques to create templates or to modify existing behaviors. However, we can also use A-Life to create scripts from scratch. Manual creation can be laborious, time consuming, and error prone, and breeding solutions completely from scratch is not usually viable. It is better to start with templates (skeletons) and concentrate on creating focused "mini scripts" to solve specific problems. A single entity can have several behavioral traits that can be combined in different ways to reflect emergent behavior.

The goal will be to meet the universe halfway—some aspects scripted by humans, some algorithms derived from observed behavior, and others generated by trail and error and integrated with the human scripts. This way, we can breed some good behaviors from existing, known good solutions and modify the environment as we go along by changing the variables that surround it, thereby improving the overall available pool.

To maximize these benefits, A-Life principles, such as genetic programming to change the variables, can also be used. If the correct result is not produced, a rule-based problem-solving approach can be employed to try and retain only the "fittest" algorithms. There is also the option to use natural selection and allow humans to participate in the selection process. Some games, such as *Galapagos*, even make this part of the game itself and do not just use it as a tool to achieve superior behavioral traits.

There are several ways to implement this sort of selection. Versus theory can be used, in which human players train the A-Life counterparts by playing against them. This is akin to training armies fighting campaigns in a scenario-based strategy game, or priming a chess computer, but it occurs in the design and development part of the project. (As an aside, if we had access to all chess games played between two Grand Masters, this would provide an excellent and fertile source of behavioral patterns to exploit.)

Direct intervention can also be used by observing clever behaviors and scoring them post-gameplay for selection and recombination (breeding). Recombination relies on the player/developer knowing what behaviors are to be encouraged, which might be easier than deriving an automated scoring mechanism.

On the other hand, one advantage of automatically bred A-Life scripts in games is that it can happen virtually. Without the graphics, sound, or interaction, the whole virtual breeding process can happen much faster than if done manually.

Using these techniques, we can breed scripts from scratch very quickly. The vast majority might be discarded along the way, but there is a good chance of retaining useful DNA strings for delivery with the final package. A substantial investment is made, however, in ensuring that the correct behaviors are propagated. For example, if we are creating a templated system, we need only provide some basic behavioral building blocks in scripted FuFSMs. Each FuFSM contains state changes and decisions, along with some weighted decision-making apparatus to provide adaptive behavior without changing the state machine itself. We can then alter just the fuzzy

decisions, thus changing the degree of behavior, but not the flow. This is genetic programming at work; we change the parameters, but not the actual algorithm, and still obtain adaptive behavior.

On the other hand, the decisions themselves can be changed so that the outcomes vary, but with the same weights in the FuFSM network. This is like using genetic algorithms in which we change the processes that operate on the data to produce variants in behavior. And we can do both. This is rarer and more difficult to control and evaluate, but it still has application in certain circumstances.

However, in a nontemplated system, we let the script building take place without any intervention or priming. Templates are not used, and the starting point is a blank slate, along with a grammar capable of producing a valid FSM or FuFSM from scratch.

Of course, initial variations will be odd and possibly include infinitely looped behavior. We have to decide how much leeway to allow the system in creating a script in order to mitigate this kind of behavior. The only problem in using intervention is that we restrict the power of the system to create scripts naturally.

This ability/power will also be affected by the granularity of the system. A coarse degree of granularity will produce fewer variations, whereas a finer degree of granularity will have more potential for variation—but it will also need more control in the way of either AI management or human intervention. For example, a coarse granularity might allow API (application programming interface, or commands that produce actions within the game universe, in this case) functions such as Move, Shoot, and Fire, coupled with access to state information through functions like Health, Ammo, and Distance to Player. Telling the system to Move will invoke a chain of AI-controlled events to make sure that the move is sensible, and the machine will select the exact nature of that movement, as defined by a rule-based AI system. We can then create simple FSMs or FuFSMs that produce certain behavioral chains based on discrete or fuzzy information that will embody state machine-style behavior. Each one will be a combination of decision statements and the aforementioned API functions. We could further augment this with personality traits that adjust for values used for decision making in a kind of adaptive resonance system.

From easy building blocks, we have already created a number of options, especially if Move takes arguments specifying a location—Toward Player, Away from Player, Toward Wall, Away from Wall, To Nearest Door, and so forth—which provide a slightly higher degree of granularity. Depending on the way that all the possible values and API functions can be combined, we might have hundreds of candidate scripts in each generation.

A finer degree of granularity might take this a step further and provide an API that allows discrete movement and sensing—such as Move Left, Where Is Player—as well as mapping and other AI techniques to provide a more-complete "brain." This will generate thousands or tens of thousands of initial candidate scripts.

At this point, we need to manually weed out "silly" behavioral traits or wait for them to be discarded naturally. The finer degree of control removes some of the responsibility and subsequent ability of the machine to restrict certain chains of behavior.

We might have encoded an algorithm to manage the coarse-grained Move (To Nearest Door) that the fine-grained alternative would require a lot of breeding to generate naturally. In between, there are likely to be many iterations of spasmodic Left, Right, Forward, or Backward scripts that do not progress the solution one iota.

In addition, the description of the "DNA" that makes up the life form becomes as complex as what it is trying to describe. Choosing the right level of granularity is clearly important, as is supplying adequate template behavior where appropriate. The danger (and one reason that advanced AI and A-Life are often abandoned) is that the adaptive and emergent benefits run the risk of getting lost beneath the surface tide of creating a computer within the computer to handle it all.

A-Life Outside of Scripting

The previous section has concentrated on script-based systems, but not all game development projects rely on scripts to implement behavior within the game universe. Some might use strict expert systems and neural networks, for example, to provide similar functionality. Scripting is very useful, but there are also concerns, such as regarding speed of execution and other possible drawbacks, including scripting the interface's openness to user tampering.

Fortunately, we can use A-Life in the development stage, even if scripted intelligence is not being deployed. However, it is a little more complex and less open to adaptive technology by virtue of the fact that it is encoded in a programming language that is usually only understood by programmers—and AI programmers, at that.

Imagine, for example, that we have an expert system to train. Each rule is evaluated, and a path chosen through all the responses, leading to a final outcome that may or may not be correct with respect to the behavior we are trying to elicit. Therefore, we need to gauge the success of the path through the expert system in order to train it. If we want to use genetic algorithms or genetic programming A-Life techniques to try and generate paths that are more or less fruitful, this requires measuring success within the game universe itself, rather than just an observing AI. In other words, we know the questions to ask, just not how they can be connected in order to produce a good answer—but A-Life does. We want to use A-Life to create a path through the question maze.

So we can allow the system to make a series of decisions (initially at random), and this provides the initial sequence. The DNA string can then be used in GA or GP algorithms to produce children that can be tested and rated accordingly. This has to take place within the game, too, in the source code. The advantages are that we can have much better control of the life forms, at the risk of losing some of the ease of flexibility provided by a scripting interface. There are also usually fewer deviations from acceptable behavior, because we use a rule-based system for which the initial questions were devised manually.

However, we have to be able to completely describe their processes within the system from scratch, because we will not normally be in a position to build the actual questions. They should be provided by the programmer, like a script template.

In addition, known algorithms for A-Life based on research and modeling can be deployed and used within the system, rather than custom-built FSM or FuFSM networks. This includes flocking and squad movement behavior, which is akin to the coarse-grained API used in the scripting example above. We might use scripts to provide one kind of behavior, with nonscripted extensions that use classic AI algorithms for others. It also follows that GA/GP can be used, as was applied to the other AI techniques that have been deployed to breed successive paths through the networks, thus taking the place of scripted behavior modeling.

A-Life in Video Game Testing

The simulation ability of A-Life technology means that it can also be used to help us test video games. This might not always be the best way to test an implementation. There will be some aspects that only a human can validate, such as the user interface, flow, sound, and graphics; but for the internal behavior, A-Life has many advantages.

We can get good results by leveraging A-Life techniques within the gaming environment, as long as we have programmed interfaces to all relevant information. This is required, because one of the premises is that we must be able to measure progress and behavior against required norms. It is tempting to look at the technology as a "silver bullet," but we cannot just implement a collection of self-modifying, emergent, and adaptive organisms and expect a great game to bloom from a collection of simple rules.

A-Life in the Testing Phase

If we intend to use A-Life to test our video game, then it needs to be introduced as early as possible. In other words, we are essentially preparing for testing when A-Life is used in the design phase. Anything created to facilitate the testing process needs to be done with the forethought of where we are going, rather than just built as quick solutions that will be discarded.

The A-Life must also be tested before using it to test anything else. Unless we are sure that the A-Life paradigms are well implemented, then it will be hard to prove that the tests have been worthwhile. Sticking to smaller routines with as easy-to-test individual components as possible will help with this. The fact that these will be combined into single API calls in the final product is not relevant; we still need to break down and test each small piece as thoroughly as possible and as early as possible.

Since simulation is a large part of A-Life studies, it makes sense to apply it in situations that will benefit from this approach. If we have decided to use A-Life in a game at the very outset, entity simulations will be created from the bottom up; therefore, testing becomes much easier.

If the game will include A-Life–based simulation, its use can be extended for "mass simulation" of interactions in the system, which provides good input for testing purposes. In other cases, large populations of test data can simply be created, generated using A-Life techniques such as genetic algorithms.

In essence, mass simulation is used; many entities are doing different things to fulfill the testing paradigm. Since we have access to the behavioral units in the game system via the API (as used to implement the A-Life in the first place), we also have easy access for test harness programs.

Mass simulation within the system enables us to use A-Life to test behavior, sometimes more effectively than with alpha-beta testing. This mass simulation might require a lot of processor time, but it can be done noninteractively, meaning that we can let it run and simulate without any real-time display of results, and then review them at the end of the session. Therefore, this kind of testing is usually done without the customary interface, so it is not really appropriate for testing the actual interaction between player and game. Nor is it used for checking the flow and synchronization of the audio-visual aspects.

Test Coverage

The test coverage obtained is increased when genetic algorithms are used to modify the behavior of test entities within certain parameters. At the outset, a random population (however large), risks insufficient coverage of all cases effectively, so we want to concentrate our efforts on simulating more common interactions.

We can assume that the test coverage of interactions of a single entity (modeling the player) against the game universe entities will be higher than if human testers are used. This does, however, rely on accurate modeling of the entire game universe.

When testing, we usually only want to validate changes in one aspect of the system at a time. So for the player's actions in a first-person-shooter, for example, movement patterns can be generated and the reaction of the game universe to those patterns observed. This enables us to improve those reactions (in design testing) and fix errors found during multiple runs with different patterns (test phase).

To do this, we change the movement models (algorithms) or change the input data used to control them. Either the scripted behavior is altered or just the way that the scripted events are applied.

This can be looked at one of two ways—as a chain of commands that can be recombined to generate variations of prescribed behavioral patterns, or as real models of entities that are complete in themselves. The former requires no decision-making in the test entity, while the latter does.

Also, the first approach need not have much interaction with the universe beyond its own status, whereas the second requires that decisions be made based on feedback from the universe. One might go so far as to say that the first approach is not true A-Life, and the second is an actual simulation of life as we know it in the game universe.

Initially, the genetic material can be populated with random data. Each DNA string then becomes a sequence of predefined, randomized movements, or a random script. Placed within the system, each one can be evaluated in much the same way as the adaptive techniques discussed for breeding behavioral algorithms.

Over time, we can mimic a large population of random data by simple repetition. This will emulate a population in which there are some good players and some bad players and some players who just wander around aimlessly. It might be more appropriate to recombine appropriate player DNA strings to create variations, rather than using purely random datasets.

If specific traits can be identified, then this technique can also be used to exercise the extremes of the system. Since "any test result is a good test result" (whether the system passes or fails, it has been worthwhile), that part of the test plan should deal with extremes of behavior within the system. A-Life testing can look into those extremes that will not usually be exercised properly by normal testing. Human players cannot be expected to test all the extremes, after all. They will tend to not try things that are illogical. Or, if given tasks to validate, they might not execute them in a fashion that touches all aspects of the extreme behavior.

So these techniques can be used to offer better test coverage than alpha or beta testing by itself, especially when deployed within other test paradigms. They are also slightly cheaper than hiring real people to do the same job.

There is another golden opportunity for the use of A-Life in the development and testing paradigm. The automata developed to test the game do not make mistakes and do not get bored. Humans tend to do both after a while and subsequently become expensive when used as test automata.

Machines are capable of never missing an error, never doing something differently, once that behavior has been defined. Consequently, the A-Life and AI designers can concentrate on one specific part, safe in the knowledge that the automata in the rest of the system are just doing exactly what they are told, every step of the way. Any behavioral deviances are then likely to be the fault of the part they are changing and not the test automata.

As Mark Morris, of Introversion, recently stated to the author:

"Breadth of testing is important, but you can use testing automata again and again and again, whereas human testing is expensive and less accurate. (People miss things; machines do not.) In my opinion, it is within the QA field that A-Life techniques have their best shot of being used by game developers."

Hence, the coverage is not only achieved in breadth, but also in repeatability over time: There is no need to change anything between test runs, make sure that the A-Life automata get a rest, are equivalently skillful, or any of the other things that we need to check when humans are used. This makes artificial testers cheap *and* effective.

Implementing the Interface

The interface to the game universe should enforce the game rules. In other words, it is not worthwhile to test behavior that would be impossible within the gaming system. Take our FPS example: We can only move in discrete steps where there are no obstacles. Teleportation-style movement should not be allowed, unless there is a

game rule to implement it. This should be automatic, as long as we respect the various mechanisms for offering interaction. API access should only be provided to options that also exist for the player; and if we can build that into the design and development, then it will be easier to manage.

For example, imagine that we have joystick-handling that allows movement through the usual eight degrees of freedom, as well as several firing buttons. Now, the hardware subsystem that manages this joystick needs to communicate with the game universe in order to tell it that the player has performed an action. We can either hook this directly into the game code (bad), or we can provide API calls for each movement, which are accessible from any point in the game engine. With this approach, the API calls are also provided to the A-Life entities in test or debug mode. So we might be able to script an event such as:

```
<Monster>::Up(1) or Player::Up(1).
```

The trigger from the point of view of the player-universe interaction is the joystick handling, while from the A-Life point of view, it is the pseudo-DNA, as explained in the preceding section. We have simulated the interaction, and this can happen in the absence of the joystick.

The bonus is this: We can test the usage of API calls to the game universe as well, so nothing is really wasted. Our effort in making the APIs will make development of the game universe interfaces easier; everyone can use the mechanisms, and they become a shared resource.

The abstraction provided by the universe "engine" also allows the simulation to take place in a way that a player cannot experience. It will have access to detailed run-time data, while the player must rely on visual cues, and is less likely to stretch the universe and rules sufficiently to get high test coverage.

This is what makes the technique more effective and probably cheaper than hiring alpha and beta testers. We have to understand the workings of the system, provide the API and access to in-game data, as well as be able to measure and validate the results. The breakeven point of cost versus benefit is likely to occur where a game that uses AI in a top-down fashion tries to leverage A-Life only in the test cycle. At this point, it would probably be easier to use people than technology to test the implementation, because the underlying mechanisms would not be well understood or implemented.

Finally, AI and A-Life techniques can be used to feed data back into the system, relating to the success of our simulations. So it is not just about testing the game for correctness, but also testing it for balance and improving the behavior.

Some games do this inherently during the play session—*Civilization*, for example—and learn by experience. These will therefore have some mechanisms in place to facilitate A-Life testing, whether they use them or not.

If the results of the test runs are used to improve the AI, a better game might emerge, but we might also create very hard opponents, especially if the AI concurrently adapts itself during the play session. However, this is offset by the opportunity to learn new strategies that the game designer might not have envisaged.

POST-DEVELOPMENT A-LIFE

Our final area of A-Life application is in the post-development phase of a product's life cycle. A-Life can also play a role after the game has been developed and delivered, as long as the principles thus far have been respected. Post-application functionality might include allowing players to:

- Interact with the A-Life to induce changes in the system, or
- Tamper with the AI and A-Life in the system.

Mark Morris is also a proponent of this approach:

"With DEFCON, we are doing exactly this, so the game can be experimented and improved by academia. This is low cost to us but increases the lifetime of the product and also allows for the development of new AI bots that could be shipped in future versions of the games."

Part of this experimentation can include creating A-Life automata to interact with the game, so that both the artificial player and the A-Life inside the package can be changed and played around with in a sandbox system. This advanced use of A-Life is not common, but the use of AI scripting that is open enough to allow new behaviors to be created is more so.

There are a few examples of this in games (usually referred to as "extensible AI"), but it is more or less restricted to entity scripting within first-person-shooters. Games do not usually ship with an A-Life lab with which to experiment, unless that is the purpose of the game itself.

Current games offer extensions to the game universe, which allow the players to participate—from actual changes to the underlying game, to the modification of the game entities' behaviors. These extensions to the game universe extend the shelf life of the product and heighten the product's appeal. This also means that the game can be improved over time; if the initial release was not good enough, hard core fans can remedy that. In fact, there are instances of *Quake* bots that provide challenges (such as *ReaperBot*) that the game's creators did not envisage.

This can be taken one step further. A game with a level designer, for example, is not a novel idea, but a game with modifiable DNA or A-Life is. Depending on the genre and presentation, such a game could possibly sell on this point alone.

The benefit of new scripted entities, introduced into the game universe over time, is well documented. These include entities/objects that the player builds and "owns" and that the universe and other players must "learn" to exist with. *The Sims*, for example, allows the player to add new elements. Each one is created within the game universe rules and informs the universe of its purpose.

Some extensions are applicable to single-player games, while others are available to multiplayer games. Multiplayer add-ons require more control, because there is a greater risk that the illusion created by the game will be broken. Even so, if online games like *Unreal Tournament*, for example, did not include bots of some kind, their gaming experiences would not be as rich.

The goal is clear in action shooters: create better or different adversaries, improve on "good" game AI, and so on. Even strategy games like *Age of Empires II (AoE)* often ship with "bad" AI in some areas. In the case of *AoE*, the main criticisms cited poor pathfinding and odd and stupid attack modes.

Naturally, A-Life technology in the game's post-development phase can also simply become an extension of the testing process; after all, the same technology is being leveraged to produce similar results. If it can be reused in this way, the developer might be encouraged to invest in the technology.

SUMMARY

You should now understand where A-Life could be useful in video games. Quite a few examples of theoretical use have been covered, but they all stem from the basic premise of A-Life implementations as simulations.

In order to summarize these approaches, that following are some real examples of how and where A-Life has been used in games. The rest of this book will detail how these techniques can be implemented and provide some demonstrable source code, as well as concrete algorithms.

Examples

Black&White, created by Lionhead Studios and brainchild of veteran heavyweight Peter Molyneux, is essentially a game that involves "training" creatures in 3D so that the universe and everything in it follow the player's whimsy. It is beautifully rendered and makes use of A-Life from the ground up.

The A-Life behavior is modeled in a dataset that represents desires, learning, and object attributes. The connections between these and basic instinctive actions provide the behavioral patterns that can be learned. For example, on the *Black&White* Web site, Molyneux described the satisfaction of hunger in a creature: If a creature is hungry, and if that hunger is the strongest desire that it currently has, it will seek to eat. If a fence does not satisfy its hunger, it will learn this and move on. If it finds a human to eat and is nourished by it, then it will also note that. If the player rewards the creature for eating humans that are under the player's control, it will continue to do this; however, the player could also train the creature to eat other people's humans, or none at all. All these decisions affect the game universe.

This is a quite simple example of act-and-reward versus act-and-punish reinforcement and is a quick and easy solution to an elegant learning problem. These mechanisms are used in the game to teach the creature how to behave. There will also likely be some basic fuzzy learning and lookup tables that emulate neural networks to augment this. The final result is good, but it quickly becomes repetitive. The game seems to lack an all-encompassing goal—which was also the principal problem with *Creatures*.

Beasts, however, corrected this by taking the *Creatures* format and adding missions in order to answer criticism that the game had no concrete goal. Since the A-Life was built upon the *Creatures* chemistry, that part of the game works well, as does the concept of missions that center on Yeti-style beasts.

In *Cloak, Dagger & DNA (CDDNA)*, this is taken a step further. The A-Life techniques used also act as unsupervised learning. The implementation uses genetic algorithms to provide an evolving opponent, which is reminiscent of *Risk*. The environment is prestocked with strands of DNA. Each strand contains rules that govern the behavior of the simulated adversary. This is a good example of templated scripting behavior. The success of each DNA strand is scored, and rules allow them to combine and mutate to produce new strands of behavior. This approach, the Darwinian select-and-reproduce paradigm, works well. Reportedly, the AI gets better over time and illustrates a very powerful strategy for adaptive, emergent behavior.

Creatures also uses GA techniques. When it went to market, it made more use of A-Life than any other game. *Creatures* is set against a 36-screen world filled with objects. The player's job is to introduce a creature to objects, feed it, and punish it where appropriate to mold its behavior.

There is some GA technology for breeding new creatures, which are then based on the player-trained parents. The base behavior is managed using neural networks to simulate a brain. Then, the A-Life chemistry takes over and feeds information back into this simulated brain, such as data on pleasure, pain, physical needs, and so on. The end result is processor intensive, but ultimately successful for at least one generation of entities. It can, after that, get a bit stale. However, there were three or four games in the series, plus the aforementioned *Beasts*.

Another game in which the central protagonist is represented by an A-Life entity is *Galapagos*. To quote *Next Generation (NextGen)* magazine:

> *"More that any other title ever previewed in* Next Generation*, the technologies pioneered in this title may significantly change the way we play games in the next several years."*
> *[NEXT95]*

What makes the game interesting is that the player induces effects into the environment, and the character will adapt to these changes, rather than the player taking control of the central character directly. Coupled with this, the entity is also an independent thinker and might go off and do something unexpected, against the player's wishes. This produces an interesting (if ultimately not commercially successful) game idea.

Nooks & Crannies also has A-Life at its core, but it was not a great success. The underlying game idea is that players have to breed life forms that are then pitted against each other. Essentially, it involves making better war machines. Again, as in *CDDNA*, each life form has a strand of DNA that describes its actions. Breeding occurs when the entity has enough food to divide (reproduce), and the result is a new population from which the player can choose which will survive.

Finally, *Half-Life* took gaming a huge step forward (as was noted in Chapter 2) in its use of AI. However, it also used some clever flocking and squad-movement algorithms that come straight from A-Life textbooks. Coupled with the openness of the AI scripting and its massive appeal, *Half-Life* has proven to be one of the truly commercially successful applications of A-Life technology to date.

To sum it all up, the previous chapters illustrated the impact of the pure AI and how these algorithms are important in video games. To a certain extent, this is used to manage the effects of the emergent behaviors dictated by the game's A-Life.

A-Life has the potential to expand on the possibilities, but it is also more sensitive to changes in variables, as well as being error prone if not implemented carefully, gradually, and correctly. In the rest of this book, we will examine the coupling of AI and A-Life, methodically and in several steps, to ensure that implementing these techniques in a game project will avoid common mistakes.

References

[NEXT95] *Next Generation* magazine, December 1995, p. 116.

CHAPTER

4

THE A-LIFE PROGRAMMING PARADIGM

In This Chapter

- A-Life: The Game within the Game
- Evolving Behavior
- Planning the A-Life Implementation

This chapter will introduce the paradigm that should be followed when designing and programming A-Life in video games. It will prepare you for Chapter 5, "Building Blocks," and Chapter 6, "The Power of Emergent Behavior," which will build on this preparatory information. This chapter will begin to show how A-Life can be worked into most video game projects. Then, Chapter 6 will present some actual nuts and bolts of AI and A-Life mechanisms, with Chapter 7, "Testing with Artificial Life," tackling the implementation of the entire gaming system.

The first step is to understand the thought process that needs to be followed in order to define, design, and build A-Life. It requires embracing some concepts that may seem, at the outset, to be an added burden on the team, but the benefits are clear.

The decision to add A-Life to any programming project involves creating a game within a game, or program within a program. In addition, all the various tools and utilities needed to support the A-Life also must be created. It is a reusable solution, however, so most of the code that is created and the design lessons learned can be applied to future projects. In addition to the in-game management, it is also necessary to spend some time designing the A-Life protocol. This includes defining data storage, communication, and what AI units will be used to manage the A-Life, as well as choosing appropriate representations for the individual entities.

Having specified how the A-Life will be included (from scripting to data representation and object interaction), it is then necessary to define the framework into which A-Life will fit, within the scope of the video game. This occurs in several stages and is not something that can be added at the end (which we have tried to make clear throughout this book). At the design stage, the A-Life framework has to include the appropriate mechanisms to allow the future addition of features. In other words, it is useless to create entities whose behavior is prescripted in the source code and cannot be modified if we want to create scripted behavior that can adapt by changing the behavioral model.

During development, each application of A-Life will need to be supported by appropriate resources. If this sounds a little obvious, then remember: With A-Life, some behavior is allowed to emerge, but some is scripted. That scripted behavior is as much a development effort as is creating the framework in which it is possible.

Finally, testing time has to be allocated specifically to the A-Life part of the project. This might comprise testing the A-Life itself, or it might mean using A-Life to help test other parts of the system. Both of these uses will be covered in this chapter's description of the A-Life paradigm, as well as the following, in detail:

- Individual behavior: Create in-game objects or entities that use A-Life techniques to derive their behaviors in isolation of any other game elements.
- Group behavior: Collections of entities bound by A-Life–style rules will mimic real-life dynamics.
- System behavior: The system will use A-Life to create an environment that caters to the player.
- Digital genetics: Paradigms can be leveraged to mimic some of the genetic processes that create A-Life behavior from simple building blocks.

- Granularity: Determine the level at which different parts of the system need to be modeled in order to make the best use of A-Life.

The final section of this chapter will deal with the practicalities of the A-Life paradigm and how to avoid pitfalls that can stem from its application. In particular, we deal with the obvious problems of resource sinks, especially when resources run the risk of being overused in the effort to implement extended A-Life systems.

A-Life: The Game within the Game

Instilling A-Life into a software project can be viewed as creating a system within a system—a system that reprograms itself periodically within certain limits. In other words, we are creating a program within a program, or a game within a game.

Like any other program, the A-Life system will be composed of data and instructions to manipulate that data both within the A-Life system itself and within the game universe. In other words, we need:

- Instructions that represent the A-Life,
- Instructions that manage the A-Life, and
- Instructions that interpret the A-Life.

So when A-Life is created, a paradigm must be followed that allows programming at several levels. Each piece of the puzzle needs to be treated as a program; it will need to be designed, created (implemented), and tested.

The A-Life itself must be tested, as well as the instructions that manage the A-Life and the instructions that interpret the results as actions in the game universe. This semirecursive mentality can be difficult to grasp at first, as it runs slightly contrary to normal programming paradigms.

The cornerstones of A-Life are genetic algorithms. Our A-Life will be represented as data, instructions, or a combination of these, which will be manipulated using:

- Genetic algorithms—in which the data is manipulated across known instruction sets that represent a possible solution, or
- Genetic programming—in which the instruction sequence is manipulated to generate new sequences that might or might not provide a solution.

One last point: In an attempt to keep the discussion focused on our use of A-Life in video games, some theoretical corners have been cut. In other words, this is not strict science, but an interpretation of scientific principles that have been researched in recent years. We will be applying AI and A-Life to video games and, to a certain extent, general programming problems that require some semblance of intelligence or problem-solving ability and adaptability. Our discussion, though, will exclude some of the complexities that make up an in-depth scientific study of artificial intelligence/life.

Scripting Interfaces

If we concede that A-Life is needed in our video game, then a way is also needed to represent that behavior in-game without prescribing it in source code. In order for A-Life to function properly, it is essential to keep the entity or behavioral code separate from the core game code so that events, behaviors, relationships, and actions can be scripted at a higher level. Otherwise, it will be difficult to manage the project and almost impossible for the A-Life to perform and will deny any emergent behavior ability within the game itself. Without external representations of the A-Life, we would need to recompile the source code for every small change.

External scripts also gives the player the ability to extend the AI and A-Life with his own scripts—something that we might or might not want to allow. Scripts can take various forms—from a C-style programming language to basic binary code—depending on its use in the system. Scripts are usually required in one (or more) of three categories:

- Discrete, prescribed, behaviors
- Adaptive or emergent behaviors
- For use in genetic algorithms/programming

Each category uses scripts in a slightly different way, and each can be simplified for easier implementation or made more complex to enable a better human interface. Generally speaking, the closer to natural language the scripting is, the more difficult it will be to implement in the game engine. Conversely, the easier a script is to implement in the game engine (even as basic as pure bit-code notation), the harder it will be for a human to create the script. There is a reasonably good compromise, however, that will convert the language representation into "byte code," which is easily interpreted and manipulated by game engines. For example, Table 4.1 compares three possible representations.

TABLE 4.1 SCRIPT REPRESENTATION COMPARISONS

Script Language	Byte Code	Numerical Form
`If EnemyWithin (10)`	`LDX EnemyWithin`	`1 1 0 1 2 10 2`
`  Fire ( EnemyLocation )`	`LDY 10`	`1 2 3 4 1 0 1 1`
`Else`	`SUB X, Y`	`1 2 4 3 4 1`
`  MoveAway ( EnemyLocation )`	`BNN +4`	`0 4 1 0 3 4`
`END`	`LDA Fire`	
	`LDX EnemyLocation`	
	`INT`	
	`JMP +4`	
	`LDA MoveAway`	
	`LDX EnemyLocation`	
	`INT`	

From left to right, Table 4.1 contains a script-language version of a simple decision process: If the enemy is within 10 units of distance (in any direction), then fire toward the enemy's location. Otherwise, move away from the enemy's location.

The second column contains a byte code equivalent, based loosely on 6502 assembly language, in this case. Registers `X` and `Y` are loaded with suitable values, and then one is subtracted from the other. If the result is not negative, the script branches to the instruction four steps away; otherwise, execution continues.

The final column takes the byte code and translates it as a string of digits that require a lookup table to be correctly interpreted into in-game actions. Each of these approaches has its use in scripting, depending on what level of AI or A-Life it is being implemented at.

So for discrete, prescribed behaviors, we can use an easy-to-write scripting language, such as was presented in the first column of Table 4.1, which is easily understood by humans but is also quite sensitive to interpretation. If the human makes a mistake or chooses to write the code in a different format (layout), the engine has to be flexible enough to be able to interpret the script and robust enough to handle any errors.

For adaptive behaviors, the representation has to be easy to manipulate, so we tend toward numerical encoding. Each instruction and variable (data) is represented as a number and is, as such, very easy to change when altering the script by using adaptive, emerging A-Life or a learning algorithm.

The byte code representation, in the second column of Table 4.1, has its uses when encoded as a tree structure in genetic programming. The genetic programming approach takes the tree structure and generates successive versions of it, and the script is evaluated slightly differently for each individual, programmatically created script. Arguably, a purely numerical or byte code representation could be used for that; however, the byte code has an added advantage in that it can be read and used by a programmer.

Genetic algorithms (as opposed to programming) are also in the byte code or numerical representation category. Here, we are modifying the parameters of the solution as opposed to the instruction sequence, so we have a choice between the two representations.

Each entity within the game will need to be scripted in some way, even if it is just a simple finite state machine or collection of connected decisions. These entities are not limited to the various enemies that the game might be made up from. Everything, from objects to weapons to the system itself, anything that has some level of in-game interaction can be scripted. The more scripted behavior we implement, the better the final AI and A-Life *can* be, even if we choose to restrict its final use to easy-to-achieve yet effective solutions. So different levels of entities will have different scripting formats, as will different levels of AI and A-Life, but there is nothing to prevent us from turning one representation into another (compiling inside the game engine) for different end uses.

Finally, the AI that governs the A-Life will also be based on one of the scripting languages, as it will need to process the entity's instructions and data. For example,

the neural network that lies at the center of a genetic programming, adaptive A-Life system models the tree structure of a program that is being genetically engineered within the system.

To sum it up, the first part of the A-Life programming paradigm is the scripting interface(s). These allow us, for different categories of A-Life, to specify the default behaviors or evolving behaviors, or even enable us to generate completely new behaviors.

Categories

The preceding section explained how different entities require different scripting approaches and how the selection of AI and A-Life algorithms have an impact on the kind of scripting language used. However, the actual behaviors themselves also fall into fairly neat categories:

- Life-like behavior
- Synthesized behavior
- Adaptive behavior

Deciding the right behavioral categories for specific parts of the game will also have an impact on the scripting language chosen. Our game within a game needs to provide for these behaviors at either the source level, the scripting level, or a combination of the two.

The life-like behavior category tends to contain things such as the opponent AI and other prescribed behaviors that might be design-time emergent but are otherwise fairly static in either the source code or external script. The end result is lifelike and predictable within these parameters, but is not designed to change during the game process itself.

The extent to which the script is applied might change, however; therefore, the entity using the script can be put into a game-specific behavioral category. This also allows the use of genetic algorithms to enable loosely adaptive behavior.

Imagine that the example in Table 4.1 describes part of an FSM for an opponent tank. A back-off distance set at 10 units is the default behavior, placing the tank in an aggression category of "mild." The behavior is prescribed—attack or run away—but the extent can be changed. If a population of tanks is then created with various aggression levels—back-off distances that vary between 1 (highest) and 100 (lowest)—we can observe their success over time and pick the best trade-off between aggression and other components of the FSM during a game session. This offers some flexibility in a constrained, safe manner.

The synthesized-behavior category is similar, except that the rules are well understood and are aimed at producing behavior that is not only lifelike (it reflects a specific category linked to the behavior of entities within the game), but also synthesizes observed behavior. One example of this is flocking, in which the group dynamics might override, or at least complement, individuals' behaviors. So, we might end up with a population of very aggressive tanks swarming on a target, or

we could have a variety of aggressions that cancel each other out, depending on to what extent the flocking algorithm is allowed to affect individual behavior. The basic flocking algorithm (or at least the rules governing it) is static, but it can produce behavior that is more or less unpredictable within certain limits.

This leaves us with the final behavioral category—adaptive behavior. Now, it was noted that being able to model some aspect of life-like behavior (such as aggression) allows flexibility in implementing adaptive *scales* of behavior, but true adaptive behavior allows selection of possible approaches.

Adaptive behavior includes learning algorithms based on trial and error, in which behavior can be modified to various extents or a behavioral profile chosen based on past experience and current observation. This might be as simple as a rule that states "I will choose an aggressive behavioral profile if the opposition seems to be adopting a less aggressive stance," or something based on an AI management algorithm, such as an expert system or neural network.

Quite clearly, this illustrates the levels within the A-Life paradigm; not only do we have to alter the behavior itself, but we also need to provide for a general, overriding, or controlling system behavior that can adapt. These will be referred to as "AI management routines" and will be covered in more detail in Chapter 5.

Adaptive behavior, then, allows for some loose rules that allow a certain degree of freedom in choosing solutions at one end of the scale and genetic algorithms allowing the construction of completely new solutions at the other end. Both extremes are dangerous—the first because it is no better than existing solutions, and the second because it allows a little too much freedom for embarrassing mistakes by the system. To get around this, a system based on behavioral modification can be specified—one embodiment of adaptive behavior, which allows the game to modify the default entity behavior. Behavioral modification is a part of the A-Life programming paradigm that warrants further expansion within our problem domain—video games.

Behavioral Modification

Behavioral modification allows us to implement many aspects of A-Life and is an integral part of the programming paradigm. It works at several different levels, from the ability to swap between behavioral models (prescribed, adaptive, synthesized, and so on) to allowing scripts that implement changed behavior. This means that the player is allowed to alter specific entities' scripts to improve or change their behaviors. More important, using a similar mechanism, we can enable AI management routines to adapt the scripts or parameters and thereby dynamically change behavioral patterns.

Again, this illustrates the multifaceted nature of the A-Life paradigm. Behavioral modification can work at several levels. Take the example of allowing an externally scripted interface to be specified, which will make this much clearer. Bear in mind that whatever can be done externally can also be done internally; so if the user is allowed to introduce a change, then the managing AI can also be allowed to make that change. With this in mind, assume that we want to allow a fully scripted behavioral-modification system. To do this, we need to expose:

- The entity-prescribed behavior scripts,
- The adaptive behavior scripts,
- Any synthesized behavior scripts, and
- The behavioral model-selection scripts.

In our tank example, a fight-or-flee selection mechanism was specified. This is the prescribed behavior, which was exposed via a C-type scripting language. Here is the script again, with the addition of a function header:

```
Function FightOrFlee ( BackOutDistance )
If EnemyWithin ( BackOutDistance )
    Fire ( EnemyLocation )
Else
    MoveAway ( EnemyLocation )
End
End Function
```

Now, if we wish to give our tank some adaptive behavior while remaining within the fight-or-flee mechanism, it can be based on a weighted historical learning algorithm. We will examine exactly how this is implemented in Chapter 5, but it essentially allows a decision-making process that selects a previously successful outcome, given similar circumstances.

The algorithm needs to be exposed via scripting that has access to the various weights and historical data that give rise to a trigger condition. We call `FightOrFlee` with an appropriate value in the `BackOutDistance` parameter. This value will adapt according to the result of the basic learning algorithm.

The algorithm needs three items—a threshold value, a success rate, and a measure of success. The threshold is a variable value; for each historical event that takes place with a given threshold, a record of success against the measure of success is kept. If the current situation is predicted to produce, on average, an outcome that is better than the success rate specified, then the event is triggered. We might choose to write this in a separate script:

```
Threshold 10, Success 75, Measure Damage : FightOrFlee ( Threshold )
```

Now the only part missing is the snapshot of the situation under which the algorithm is applied—a selection of in-game variables that need to be within a certain percentage range in order for there to be a match. Though outside the scope of our current discussion, these might include statements such as:

```
Damage 0 < 50
```

This would be read as "damage between 0 and 50." The neural network that manages the adaptive behavior must be able to distinguish between circumstances, outcomes, and measured (weighted) success.

Next, the synthesized behavior scripts that exist need to be exposed. Again, these are generally beyond the immediate scope of this chapter, but they might include parameter files that manage flocking behavior—sequences of numbers that tell the algorithm what distances the entity should maintain between it and other entities/objects.

Finally, the behavioral model selection scripts also need to be exposed. These decide how much of an effect on the final outcome the three behavioral models will have. Implementation can be handled in a variety of ways—from a guided expert system to a full neural network. In either case, a list of parameters would probably be preferable to an actual scripted encoding of the algorithm. The question of efficiency is also part of the equation (we will revisit this later); some routines are simply too processor intensive to be allowed as part of a scripted interface.

All of the preceding are valid for user (gamer) generated scripts, as well as machine-generated scripts. What will vary is the kind of script, its representation, and its target for management, which could be one of three in-game entities:

- An avatar,
- An agent, or
- The system.

We will discuss system management later on. For now, let us take a quick look at avatars and agents within the video game.

Avatars, Agents, and Autonomy

When deciding how to implement certain behavioral modifications in an A-Life–aware video game, we need to consider the type of in-game entity it will be applied to. Anything other than the system itself should fit into one of two categories: avatars or agents.

An avatar is generally a passive component that provides information from the game world to the player. It helps the player in his decision-making process by offering advice and timely feedback or providing some other passive service.

Avatars need to be able to adapt their behavior in response to several stimuli—such as the pace of action, upcoming events, or observed reaction of the player—without ever losing their usefulness. For example, an avatar can track the progress of the game, prompt for actions into the future, but also know when the player really does not need a specific piece of inconsequential information.

In a game like *SimCity*, there are several such avatars that can be called up to give advice on anything from budgets to transport and waste management. They will also pop up from time to time to warn the player and give advice and praise.

If, for example, the player routinely ignores an avatar that offers advice on when to lower taxes at a time when the player is spending excessively, then the avatar should "learn," after being rejected a specified number times, that the player does not welcome the intrusion. This is a fairly benign example of behavioral modification based on adapting to simple in-game data. If the avatar goes a step further and actually lowers the taxes itself, then it changes from being an avatar to an agent.

Typically, agents perform real actions, on behalf of either the player (helper agents) or the system, or independently as opponents or enemies of the player(s), the system, or each other. Agents have the power to interact, either on the same basis as the player or in a more god-like way, depending on their ordered place in the game universe.

What keeps everything on an even keel is the degree of autonomy that we allow these in-game entities. By autonomy, here we refer to two kinds of specific freedom: the freedom of choice (which algorithm to apply, for example), and the freedom of movement (actual displacement within the game universe). One might lead to the other; certain freedoms in the decision-making process will lead to higher or lower degrees of movement within the game universe. We might call these two freedoms "behavioral autonomy" and " autonomy of activity." An autonomous agent can be an in-game robot, opponent, or helper, whereas an autonomous avatar is an adaptive information source. Depending on the behavior and role of the system, we might choose to make agents and avatars semi-autonomous and only let them be partially responsible for their actions. Not surprisingly, both the game design and part of the A-Life programming paradigm have to provide limits on the autonomy of in-game entities to avoid chaos.

The autonomy of activity relates not only to physical movement from one part of the game universe to the other, but also actions that might be carried out within the game universe that have an outward effect on it. The behavioral side just makes the decision as to what to do next, and the activity side actually carries something out, however complex or simple that action might eventually be.

System Behavior

The system has been defined as an entity in the game universe, so we should try and be a little more concrete as to its exact role. Aside from all the graphical, audio, and controller logic that the game engine contains, the system also manages the game universe, the events that take place within it, and the rules by which all entities are bound. One example is in difficulty ramping, which is not unique to any one genre, but the way that a racing title handles ramping will be different than in a first-person shooter. Chapter 3, "Uses for Artificial Life in Video Games," showed many different areas, in addition to difficulty ramping, in which the system can have an effect on the gameplay and be managed by AI and A-Life algorithms.

Generally speaking, when we talk of altering system behavior using AI and A-Life (like difficulty ramping), we assume that the result changes an in-game agent or avatar. These new changes are separate from any autonomy given to these entities to adapt their own behaviors.

Even if a game's system could be all-controlling and no autonomy given to the in-game agents or avatars beyond carrying out the scripted AI that they have been created with, there are clearly degrees of system manipulation that can be imagined. This prompts two questions: Does the game come first, or does the A-Life come first; and how do we use feedback (in the context of the video game) to make sure that any system behavior modification (of itself and in-game entities) is achieving the

right result? The answers to these two questions are very important parts of the A-Life programming paradigm and will also affect the eventual design of the game AI and A-Life engine.

Feedback from the System, User, and Other Entities

The role of feedback in A-Life is very simple: It enables the algorithm to measure an in-game variable and add the weight of that variable to future decision-making processes. It is invaluable in "unsupervised learning" algorithms, in which implementation is designed to be executed in isolation of external, third-party result-scoring.

Players use feedback all the time—from listening to the sound of exploding enemy spacecraft to watching people leaving failing towns and/or soldiers fleeing battlefields. Part of the responsibility of game design in general is to give the player enough feedback so that he can make gameplay decisions. The same applies to the opponent entities' AI in video games.

In basic terms, feedback is usually a measurement of success or failure. The system needs some way to measure the value of a decision or situation. By using feedback, we can begin to structure risk/reward algorithms that balance anticipated success against possible failure. Players do this naturally, but it is something that needs to be built into our AI routines so that they have this added tool for decision-making.

This sort of feedback might evolve by itself (as a result of mechanisms that will be discussed in Chapter 5), depending on the design of the system, but it might not—which is difficult to know in advance. Therefore, we need to tackle it head on and provide a way for the algorithm to score the result of a specific interaction with a specific entity. The very simple adaptive system that we previously built also used feedback to adapt itself according to the success rate of a given approach; in this case a number relating to the distance to an enemy unit.

Applying feedback can have several components. There needs to be a way to enrich the algorithm (or the data that the algorithm is operating on) by updating it with the result that is being fed back, and there also might be a need to track the circumstances, depending on the exact algorithm being used. A final option is that the entire feedback system can be discarded in favor of a simple random-selection algorithm, possibly augmented by probability data that is drawn from analyzed previous outcomes, which is effective in areas where unpredictability within certain prescribed limits is required.

Evolving Behavior

Part of the A-Life programming paradigm is that we would like in-game behavior to emerge in a bottom-up fashion, as opposed to being dictated at design time using clumsy pseudo-AI techniques. We say "would like" because there will still be some necessity to direct some influence down from the top; not everything can be left to emerge autonomously, except in rare cases.

So A-Life does not replace traditional AI. The basic behavioral structure, as we have seen, still relies on AI mechanisms that are (thankfully) quite well understood. These building blocks make up much of Chapter 5 and include the following:

- Neural networks (for brains)
- Finite state machines (for discrete behavior)
- Fuzzy FSMs (for adaptive discrete behavior)
- Expert systems (for situation analysis and modular decision making)
- Extrapolation techniques (for prediction)

There are other AI techniques that can be used, but they are generally all derived from or based on items in the previous list. These have been covered in earlier chapters of this book, as well as the ways in which they can be applied to various genres of video games.

In A-Life, what ties everything together is the ability to use genetic algorithms and propagated/emergent behavior side by side with traditional AI techniques. In doing so, behavior can evolve at three important stages of video game development:

- Design,
- Execution, and
- Testing.

Part of the A-Life programming paradigm means understanding the terms and where they can be applied. We will start with genetic algorithms and then continue with propagating and emergent behavior—all in terms of the paradigm. (These topics, including their implementation, will be covered in detail in coming chapters.)

The Role of Genetic Algorithms

In video game development, genetic algorithms take predefined behavior and generate new behaviors that we hope will follow existing patterns, improve on them, or create valid evolutions of patterns. We accept that along the way, behavioral patterns will be generated that are invalid, less successful, or otherwise impaired.

Our application of the A-Life paradigm needs to have mechanisms for selecting the good and removing the bad from the digital gene pool. This can be done manually at design time; at test time, it can also be done by a human, or it can be done by the system through natural selection—flawed behaviors being less likely to survive.

However, at execution time, we need to be able to select and remove flawed behavior before it is introduced into the game universe—and before it is revealed to the player, which will usually not be a pretty sight. The player should not see works (entities) in progress, such as tanks spinning around in circles or demented soldiers slaughtering each other.

Before we look at how these works in progress are derived, as well as what to do with the cream of the genetic crop, let us recap the two main genetic algorithm processes that are included in this A-Life discussion: genetic algorithms (GA) and genetic programming (GP).

The reader will remember that the key differences between GA and GP are that GA modifies the inputs to an algorithm that remains fixed, while GP modifies a sequence of instructions in the hope of finding a sequence that solves a problem. They are both measured in terms of their fitness against some notion of a perfect solution, should one exist.

The GA process changes parameters to be fed into a solution in an effort to derive the best combination for an optimal solution. Along the way, we will likely encounter parameter sets that are worse than the first possible solution, as well as some that are technically as good, but defective in some other way.

GP, a form of GA, starts with potential solutions/candidates (if any); a solution/candidate is a script or program, plus the data that can be fed into it. We derive new solutions that may or may not be successful as defined by our measure of success.

Success, in both GA and GP, might just be to derive some behavior that is different yet similar, or it might be more concretely aimed at furthering the goal of the player or opposition. The two cornerstones that enable GA and GP to work at creating or evolving behavioral variations are "recombination" and "mutation." These are derived from observed, real-world genetic study. They allow us a certain degree of flexibility when using two or more parents to create children and their siblings. It is also worth noting that in certain circumstances, none of the resulting behavioral patterns will be discarded if the solution domain is restricted enough so that there is no risk of unacceptable behavior.

Recombination

Genetic code recombination takes the essence of two parents and creates a child that contains some of the properties of each parent, such as half of each, or one-third of one parent and two-thirds of the other, or any other combination that makes sense within its application in the problem domain (see Chapter 5). Part of the A-Life paradigm is knowing whether there are natural behavioral boundaries that allow us to select different aspects of the behavioral pattern as genetic material. For example, we might take the `FightOrFlee` function as one part of the pattern (aggression) along with other functions that govern other behavioral aspects as our boundaries.

These pattern parts are recombined by selecting among them (the parent), rather than breaking up a specific function and trying to recombine the pieces to create offspring. In this way, the possibility that rogue behavior will occur is limited; we have the aggression of one entity, the navigation of another, coupled with the squad or flocking behavior of the first.

Recombination strives to change the behavior of successive offspring. We can either recombine the results or just be content with one first-order variant and continue the game. At some point, the genetic record of the "longest survivors" can be kept (fossils) and scored for future use when, for example, an opposing entity population needs to be respawned.

Without recombination, no A-Life will emerge. It is, in a sense, fundamental to the whole exercise when applied to emergent behavior over time. Clearly, we can

have emergent behavior on the basis of discrete entities, but to give rise to populations over time, recombination must be used.

We can still have behavioral variety, but it will not be able to adapt in the same way. Even a flocking algorithm uses adaptive behavior by altering genetic code represented by inputs to the behavioral mechanism. These inputs then change the weights that govern individuals' movements within the flock.

However, this is not the same as generating successive populations (generations) of subtly different flocking entities in an attempt to produce the best possible solution space. In this case, the results might not be wildly different or entirely appropriate; but for more complex entities, it is a very useful mechanism.

Taken to an extreme, we can use it to evolve the opposition to the player, or evolve system behavior to challenge the player (should there not be any direct opposition, such as in simulation games). During the play session, the entities might evolve further using GA or GP, and we could even create and insert test players into the system during the test session.

While recombination is powerful, and while there may be many possible recombinations of genetic material, depending on the granularity of the system, we might run out of possible variations. Therefore, we need to be able to generate (possibly unlikely) behavioral patterns that would not necessarily occur naturally. Here, mutation can be used as a way to extend the results offered by recombination.

Mutation

Mutation is another aspect of genetics that is derived from observations of naturally occurring genetic reproduction, but mutation is different from recombination in that we randomly tweak the genetic material as it creates the child. In life, we can never be sure what mutation will arise, because the changes, as well as the place in which they take place, are purely random.

However, we can also use directed mutation (that is, altering digital genetic material to our advantage) to alter the children. This directed mutation is the digital equivalent of genetic engineering and can be used as a way to prevent rogue behavior, or just as a way to propagate new behaviors.

The underlying mechanism is to change a random portion of the digital genetic material that represents the behavioral pattern being modeled, at random intervals. Each genetic recombination is not changed, because that would defeat our purpose. The same restrictions apply as in the preceding discussion of recombination regarding granularity and choice across behavioral boundaries. However, we might be able to interfere at a lower level and within individual patterns, such as the `FightOrFlee` function, and tweak individual lines within these patterns without risking the emergence of rogue behavior.

If we only change those parts that have no direct effect on the *result* of the mutated behavioral pattern, just the manner in which the behavior manifests itself, we reduce the risk of rogue behavior. Similarly, if we only swap out action scripts and avoid changing anything that dictates decision points in, or flow of, the script, then we can further reduce this risk.

With all the risk mitigation in place, we can then embark on either natural mutation, which is purely random, and see what it brings, or directed mutation, in which we try to engineer something that provides more advantage to the game than it does the player.

Propagating Behavior

Propagating behavior refers to keeping behavior that will be representative of in-game entities while discarding unwanted behavior. This can be achieved using GA and GP, as was previously outlined, but it can also be performed in other ways as part of the A-Life paradigm.

Scoring and Propagating Behavior

As in the GA/GP approach, we need to know how to score the behavior that is being witnessed so that it can be tweaked to improve or downgrade it, depending on the outcome of the scoring mechanism and the goals of the propagation algorithm. These must be defined within the game rules and appropriate algorithms put in place to make sure that the behavior of similar entities is in line with the expected general behavior. There will be variations between entities with different "characteristics," but they should all exhibit a basically similar behavioral trend.

For example, in a racing game, we might decide that lap times and crashes or collisions are the criteria by which the success of a given entity will be judged, as compared to the player's in-game persona. So the behavior of the weakest winner in terms of lap times would be propagated, coupled with the driver with the least collisions to provide a fairer field for the player to compete against.

In a first-person-shooter, we might isolate hits, hit ratio, kills, and misses as our criteria and adapt behavioral patterns according to those entities that represented the best or worst in each category. In this way, a difficulty level can be provided that adapts itself during the play session, rather than using the standard static difficulty ramping that most games employ. Finally, the FPS could also take into account the damage, kill ratio, and ammunition expended of all entities in the system and attempt to propagate behavior that is slightly superior to the player's. In each case, as long as we can measure and score behavioral patterns, we are able to use various techniques to ensure that the behavior can be carried forward—and in real time—for example, in adjusting group dynamics.

Group Dynamics

The premise for manipulating group dynamics is that we wish to treat a collection of otherwise independent entities as a unit that exhibits specific behavioral patterns. The behavior of these individuals is governed by the behavior of their peers—each one having entirely predictable behavior. Individual elements might exhibit varied extremes of this predictable behavior; in other words, there will be a wide variety of

behaviors within the population—some on the edge of what is acceptable and others well inside. These behaviors are communicable because each entity governs its behavioral patterns with respect to its peers.

So these behavior patterns ripple through the group being either enhanced or counteracted by the applied behavior model. A good example is the flocking group dynamic. Like birds all trying to avoid each other in level flight, they move within an envelope that enables them to flow around obstacles and all take different paths, but they move as one unit (group) in a single general direction.

Another group dynamic that works in this way is "mobbing," in which an escalating borderline behavior propels the entire group toward a single goal. In a mobbing dynamic, otherwise fairly benign behaviors can be altered toward that goal as the behavior feeds into individual patterns.

Squad play also fits into the A-Life programming group dynamics category. It is slightly more advanced and requires simulated communication between individuals in a way that transcends simple flocking or mobbing. Flocking and mobbing rely more on reactions without specific communication between individuals; their individual observation of behavior is used, rather than a formal exchange of intention. This also adds a slight time lag to the result, making for a more naturally flowing result.

To implement this, some specific traits must be designed into the system, using techniques such as object-oriented design and programming, which allow external interfaces to be exposed. The interfaces to the object classes allow individuals to read information about each other, as well as send each other triggers that can be fed into the behavioral algorithms.

Implementing group dynamics can easily lead to emergent behavior; the group becomes an entity in its own right. Or, as another example, individual elements in a system work together to produce the end result. The key to emergence is that we cannot necessarily divulge the exact reason for behavior from looking at either the group or its individual parts.

Emergent Behavior

When we add all the pieces of the A-Life system together, behavior is generated that is beyond what we might expect, and new behavioral patterns that were not necessarily predicted are witnessed. In other words, it is the individual behaviors that feed back into the system and drive new behaviors, which are sometimes unpredictable.

This kind of behavior is also prevalent in learning and adaptive behaviors, which can also be unpredictable, particularly with unsupervised learning algorithms. This will be covered in greater detail in Chapter 6, but as part of the general paradigm, we need to be aware of the functionality that it provides.

Technically, there are two types of emergent behavior: strong and weak. Strong emergence precludes the ability to predict how the behavior will occur by looking at the component parts, whereas weak emergence ensures us that any behavior generated is entirely within our control.

Strong emergence, stated simply, is the inability to derive the outcome (a state in the system) from the inputs (input data or states), and hence the inherent inability to predict effect from cause, even when we know the full effect of each part of the system. For example, nobody can calculate, in a single step, where the balls on a pool table will end up after the break.

It is only by carrying out the simulation and calculating the effect of each ball on the other at every step of the simulation (either in real time or at higher speeds) that we can determine the result from the causes of that effect.

On the other hand, weak emergence is characterized by the ability to spot which of the causes contributed to the effect and by how much. In this way, once the "rules" by which the effect has been caused by the inputs to the system are known, the emergence can be predicted. Craig Reynolds' flocking systems fall into this category. We know how the flock will react, even though it is an emergent system where each entity contributes to the behavior of the flock as a whole.

In both cases, the small rules work together at a low level—combing to produce the actual behavior. These rules can be prescriptive, such as state machines or neural networks, but we can allow them to be adaptive or self-modifying, depending on the A-Life paradigm's requirements.

Which Comes First—the Game or A-Life?

In programming terms, and in the context of our A-Life paradigm, the question is one of implementation. Clearly, without the original game idea, there would be no use for A-Life; so therefore, the game comes first.

However, the game merely provides a context within which the A-Life paradigm can work its magic. From this point of view, we have a need to make sure that the A-Life implementation comes before the rest of the game. It can take place alongside other design and development efforts that are geared toward the outward appearance of the game—such as the graphics, sound, and physics engine—but the A-Life needs to be planned before the final rules are implemented.

No single aspect of a game can be considered to have a higher value than another. All aspects must work together. However, in terms of the approach and timing, there are great rewards to be had by developing A-Life before most of the rest of the game. This does not make it more important, but it prioritizes it in the development chain slightly. Your approach will differ, depending on whether the game's AI/A-Life component is small (a shoot 'em up) or medium (squad-based single-player first-person-shooter or a racing game) or forms a large part of the game; in essence, the A-Life is the game, such as in *Creatures*, *Beasts*, *Black&White*, and other god games.

The reason for putting the A-Life first is because we must make sure that the in-game entities are correctly abstracted from a behavioral point of view. As previously noted, this means that they cannot be governed by source code, but must be interpreted by the system, which enables the code to be interpreted to and interfaced with other parts of the game universe; this has to be in place before the rest of the game. This runs counter to the traditional AI video game programming mantra: "We cannot test the AI unless there is a functioning system to test against."

As you should now start to appreciate, this book does not subscribe to the "system-first" view for A-Life in particular and AI in general; it is preferable to make sure that the AI and A-Life components are part of the game, rather than optional extras. This means that the algorithms must be intertwined with the system from the initial design so that managed action and feedback can be provided.

Again, though, the game comes first. Playability and the pursuit of fun, in the eyes of the player, are the goals of the whole endeavor, and putting A-Life so high on the list of things to do does not change that.

Design Time

At design time, the A-Life components need to be designed in chunks. The size and complexity of each chunk will depend on the level of granularity required for that specific aspect of the game engine. Some aspects of the game will require much higher levels of granularity and therefore more chunks to represent the behavioral traits that we require.

These elements are then combined in interesting ways—connected by a variety of mechanisms that give rise to emerging behaviors. The resulting design should be implementable as A-Life, rather than a collection of individual routines that ignore all other behavioral inputs and act on their own.

So the combination of the various elements will allow us to select behavioral profiles that emerge as a result of the design. However, some aspects will only be visible in the run-time portion of the video game.

Run Time

In the run-time environment, there will be some behaviors that emerge as a consequence of the way they have been implemented. When the A-Life is achieved during actual gameplay, only the building blocks for the A-Life system have been provided, and the behavior is allowed to emerge during the play session.

The danger here is that we lose some element of control, and the emerging behavior can quickly spiral away from what is considered to be viable. We call this "runaway" behavior, and part of the general A-Life programming paradigm is to put checks in place to restrict this.

Whereas the selection of behavioral patterns (as discussed in the previous sections on GA and GP techniques) requires that we compare the result of that behavior with a scoring (or profiling) mechanism, restricting runaway behavior requires that we modify the interaction between individual elements. To do so we need to be able to:

- Spot evidence of runaway behavior,
- Identify the principal cause, and
- Restrict its effect on the population.

So part of our design has to allow for these three mechanisms. It must also include criteria that make it possible to ensure that the emergent behavior is within expected limits.

This is perhaps the biggest issue in the use of A-Life in real projects. In Chapter 3, we mentioned that QA was a real headache in artificial life systems, and the case for string emergence proves the point. A strongly emergent system offers some clear advantages in terms of the game's realism, interest, and replay rate, but it also becomes nearly impossible to manage correctly.

So to try to combat this, we restrict the emergent behaviors so that they only fall within predetermined, acceptable limits. We also try to hobble the emergence at the source by designing a solution in which strong emergence can never produce runaway behavior.

If we choose not to put any of these safeguards in place, then managing the emergence becomes a more complex proposition. It becomes so complex, in fact, that it is a field of study all to itself, and only games that use higher A-Life functionality need concern themselves with it. Consequently, we will only deal with the general case here: identify, restrict behavior so that it fits within our needs, and live with the fact that this will make it less effective than it might otherwise have been.

The result will, I believe, still be ahead of what could have been achieved without the application of A-Life in the system at all.

Planning the A-Life Implementation

Finally (and before covering the actual nuts and bolts of our A-Life implementations in video games), we need to briefly discuss how an A-Life–aware implementation can be planned. A-Life is special and different from other paradigms; we must be clear from the onset exactly what elements need to use it and in what ways.

There are many potential pitfalls; chief among them is the danger of "evaporating time" during the design and development cycles. It is quite easy to get so involved in the A-Life—we tend to bite off more than we can chew—that time (originally slated for other portions of the project) simply disappears.

These time sinks need to be avoided or they will engulf the project and potentially lead to the abandonment of any kind of advanced AI or A-Life implementations. Identify areas at risk early on, and plan to reduce their impact. Time sinks can occur in the following:

- Design time
- Development time
- Emerging (new) technology
- Testing

The goal of the final pages of this chapter is not to discourage you, but to encourage realism with respect to what can be achieved in a normal video game project.

Design Time Considerations

The first time sink is in overstretching the available technology at design time, which leads to trying to find workarounds. If the platform is handheld or another restricted-resource platform, it does not make sense to try and use processor-intensive routines, such as in an action game.

The design team has to know where to draw the line when it comes to the target platform and reduce their efforts to a more benign (and perhaps less effective) but realizable goal that is within the guidelines of the technology being considered. This is linked to the second time sink—staff not fully appreciating how A-Life is *supposed* to work.

Many designers and programmers understand (or believe they do) AI. Some might have heard of A-Life, but it is rare that they really appreciate what it can achieve. So education will be needed, or they will waste time in trying to apply half-understood techniques—and then wonder why the expected results are not generated.

Of course, half of the issue is that we want to allow some aspects of the behavior to occur naturally, so our design must place restrictions on the system, which is the equivalent of trying to manage the A-Life using inflexible AI-style rules. In other words, we need to appreciate that random prescribed behavior is not the same as adaptive/emergent behavior, and that slightly different rules apply.

Linked to this is a final point: Many designers will spend an inordinate amount of time trying to mold A-Life around the game idea. A solution might be to try and attack the problem from a different level of granularity. This means starting with simple goals that are reflected in simple entities. In other words, to avoid time sinks, work in a bottom-up A-Life paradigm fashion and restrict goals to reasonably achievable targets.

Achievable Goals

To begin with, the goals that A-Life is supposed to deliver should be limited. This might mean that some of the system is designed initially to work with simple rule-based systems or some form of sequential AI. This can be seen as a low risk option; it is evidently achievable, but the results might deliver a better end experience for the behavior if a higher risk (higher reward) option is taken and A-Life added to the initial AI.

If only one aspect of the initial design uses A-Life, then that gives us a starting point for more complex behaviors. We need to remain realistic yet allow the design to be adaptable so that A-Life can be used at later stages.

It is better to get a good A-Life system in place for some aspects than have to abandon the design completely because it has grown out of control. We might have to develop a "flexible" design that is capable of adding new bits of A-Life–enabled algorithms as we go along. However, this will not work if the underlying game engine is not built in a way that is conducive to extending A-Life. This is a development consideration that must be taken into account to make the project's future life easier.

Designers need to make sure that they concentrate the effort on areas where the A-Life will bring that improvement in the experience for the players. When I mentioned A-Life for the sake of it in Chapter 1, "Introduction," that is what I meant. If A-Life does not immediately enhance the experience for the player, it is not worth concentrating on.

The effort has to go where the benefit is highest. That goes for design as much as it does for development.

Development Considerations

There are many varied time sinks that have as much to do with regular development as they do with A-Life—most of which are outside the scope of this book. However, two key points remain. First, it is very important not to try reinventing solutions that already exist.

The two areas where the most time will be saved are in using existing scripting solutions and in using existing AI design and development patterns. Even if it means copying (the observed structure thereof, of course) an existing design and implementing something similar, the time saved is better spent on other aspects of AI and A-Life development.

The second key point is that a good game is unique enough in the first place, so trying to add on extensive AI and A-Life might not be the best solution if it is an attempt to replace another aspect of gameplay. Bear this in mind: Do not try to overextend the AI/A-Life to compensate for failings in the game itself. Of course, there are games that hold A-Life as the cornerstone of their development—that is, the A-Life is the basis for the game and provides its originality.

Emerging Technology

A smaller but nonetheless reasonably important time sink can occur when trying to continually adapt the design and development to take advantage of new advances in AI and A-Life techniques. In a sense, this book itself is part of that danger. To avoid this, try to limit the number of new ideas that crop up in the course of the design and development cycle—at least to new techniques that can be easily mastered and incorporated. In other words, the rudimentary building blocks of AI and A-Life can be implemented as a base and then the algorithms added that manage those blocks. That is, start with a system of objects and entities that are designed to be used in an A-Life fashion, but which are managed by strict rules.

For example, imagine vehicles in a racing game that always try to drive a perfect course, and that do not use of any kind of behavioral framework. These vehicles might crash a lot, or they might slow down until they can safely follow the racing line. If we want to add some rudimentary characteristics to the drivers, such as aggression, we can implement adaptive behavior that is the racing equivalent of the "FightOrFlee" script covered earlier. New techniques can then be built in that more easily follow the emerging technology; but bear in mind, unless we keep it simple, there is the danger that they will not be built at all. In addition each new added piece needs to be tested.

Testing with A-Life

Anything that we create has to be tested. This applies equally to scripted behavior as it does to more advanced techniques, including network and emergent behavioral patterns. Not only that, solutions created to help manage these patterns also need to be tested.

We can, however, employ the A-Life that is intended to add value to the game and use it to help us test the system. This will be covered in detail later on in this book, but for now, the general idea is that we reuse pieces of the design to help us test. The same techniques used to create the digital appearance of life can also be used to test it. Even the player can be modeled within the system and then let loose as a virtual entity in the game universe. This is best done before the interface is in place because the execution time will be quicker. If the graphics are already in place, then testing will likely take place in real time, and if we want to cover all permutations, the process will be time consuming.

SUMMARY

This chapter is chock full of vital information; it sets a framework into which we can drop the basic building blocks that will create our final system. This can be viewed as a game within the game, and it is important to realize from the onset that the more levels there are, the more implementation and testing will be needed.

When deciding on the granularity of individual components, the complexity can be limited, but we also have to ensure that our options will not be artificially reduced at a later stage. Our system will benefit from a proper design that takes into account the A-Life paradigm, but it might not immediately take advantage of everything that the paradigm encompasses.

In an effort to make our lives easier, we can concentrate on modeling individual behaviors, much as we would in a "non-intelligent" game design and development project, but leave the implementation open to adjustments. Scripting mechanisms can be used at this stage to open up the behavioral models, be they based on actual reinterpretations of scripts or the parameters that govern the scripted behavior. Either of these can then use GA and GP techniques to further enhance the playing experience through adapting standard behaviors, and at very little extra cost. These two techniques alone can help to create new behaviors that can be quickly optimized either at design time or at run time. Run-time execution will, however, necessitate some kind of scoring and evaluation mechanism.

Groups of these objects can then be linked together to enable interaction at a variety of levels and give rise to emergent behavior that takes your video game to a new level. AI and A-Life are often no more than buzzwords in the video game industry, but some games truly embrace emergent behaviors (if the preceding breakdown of the system into appropriate units at the design phase is followed).

All of this needs to be placed into an appropriate planning and testing framework. Team members must be kept informed about the various decisions that have

been made and how they fit into the grand scheme of the video game project. This requires a bottom-up design and bottom-up development approach. Goals must be kept realistic and simple but allow modest beginnings to be built upon.

Finally, everything that is developed needs to be tested. This includes the enabling technology as well as the actual scripted entities themselves. More than in any other project, each piece that is added needs to be adequately tested; but fortunately, the very same techniques used to create them can also help test them.

This page intentionally left blank

CHAPTER

5

BUILDING BLOCKS

In This Chapter

- AI and A-Life Building Blocks
- Deploying the Building Blocks
- Ground-Up Deployment

This chapter, which is positioned between "The A-Life Programming Paradigm" and "The Power of Emergent Behavior" chapters, introduces the building blocks that are available for implementing lifelike behavior and AI in video games. In the previous chapters, we have looked at various games and the extent to which AI and A-Life have been used to enhance the playing experience. In doing so, the ways in which some of these techniques work have been discussed without really explaining how they can be implemented. At that point, it was important to understand the way these techniques can be leveraged.

So now is the time to make these concepts concrete and set the scene for allowing emergent behavior to become a reality. We will look at the major AI and A-Life algorithms that can be used in video games, which specifically offer capabilities for:

- Learning,
- Behavioral modeling,
- Environmental modeling,
- Chemical and biological modeling, and
- Decision-making.

Learning is important because it gives the game the ability to react differently given similar (or identical) circumstances and possibly have an effect on the outcome. When we record what deviations from predetermined (and possibly random) behavior have successful outcomes and which ones do not, this constitutes learning by trial and error, which is a powerful addition to AI and a building block in itself.

Behavioral modeling determines, without weighted or other considerations, the actions that an entity takes—in other words, the digital equivalent of instinct, at least as far as video games go. Several building blocks can help create these models, and we will see how this basic instinct can be honed by using learning and other unsupervised modification techniques.

The counterpoint to behavioral modeling (which deals with entities) is environmental modeling, which works in two ways. First, it can exist as a mitigating effect to emergent behavior that arises from collections of interacting entities. Second, environmental modeling can be used to change the way that all entities make their decisions.

Chemical and biological modeling are very specialized instances of modeling in which we use carbon-based life as our basis. Not surprisingly, it does not find use in many games, and where it is present, the designer has to make some careful decisions as to the extent to which it is deployed. We will look at some examples, as well as ways in which appropriate processes can be modeled—for example, to model poison, food, energy depletion, and so on to augment behavioral models or decision-making processes.

That brings us to the crux of AI and some parts of A-Life: the power to make decisions as opposed to simple behavior. Most of the behavioral models examined will give rise to lifelike behavior through emergence or the simple combination of different models. These do not require an explicit choice of actions, such as choosing a path from several options.

Decision-making often requires the entity to make an informed choice and measure the effectiveness of that choice. This leverages several different models and building blocks that enable the entity to recognize the current situation, evaluate the *desired* situation, and make correct choices that will lead to a solution.

After deciding which parts of the paradigm covered in Chapter 4, "The A-Life Programming Paradigm," are needed, we can then take our pick from the various building blocks to implement AI and A-Life to an acceptable level. We also need to decide on the mix of emergence, adaptive behavior, prescribed behavior, and reasoning systems that need to be introduced. This will largely depend on the nature of the game and the role that AI and A-Life should play in it.

Even for those designers who are enthusiastic about adding these advanced technologies, practicality dictates that they will not receive more than a small slice of system resources. Implementing more sophisticated behavioral modeling and intelligent adaptive systems will not be a priority if it is at the expense of other elements of the video game, such as graphics and sound.

AI AND A-LIFE BUILDING BLOCKS

Depending on the game's genre and the kind of A-Life intended for it, there are many different types of building blocks that will provide the foundation for emergent behavior (A-Life systems). However, let us start at the beginning.

The way that we leverage A-Life in video games (in this book, in particular) is not as a collection of specific A-Life algorithms. Rather, A-Life results from the way that AI algorithms are put to work in the game. Chapter 4 put this in terms of the A-Life paradigm—the way that AI routines are processed and the results of various approaches. This gives us a framework and a decision process, which begins with deciding on the granularity of AI in the system. In other words, the first step toward defining what building blocks are needed is to map out the system in terms of the smallest unit required. For example, this might be a simulated biochemical reaction if we are building a part of the system that processes food into energy, such as in *Creatures* or *Black&White*. On the other hand, we might be building an organizing unit that will dictate the movements of a squad of entities in a battlefield simulation, like a general directing the action, or orchestrate the flocking behavior of individual soldiers.

Having decided the granularity level of this part of the system (and remembering that different parts of the game can have different A-Life granularities), we then need to pick a behavioral model. A system comprising simulated biochemical reactions is probably best slotted into a model that allows for synthesized behavior.

The group of soldiers in our medium-granularity example falls into the category of emergent behavior. The observed A-Life behavior results from the action and reaction of individuals as they try to fulfill commands handed down from higher up. However, each individual will itself probably fall into the category of prescribed behavior.

Because he is at the top of the granularity tree, the squad leader will be governed by an adaptive behavioral model and try to choose the best options from a collection of alternatives. This will likely be based on a prescribed starting point—a preset collection of strategies—that can be honed through experience.

As you might guess, the algorithms at the coarse end of the granularity scale are more complex than those at the finer end, on a unit-by-unit basis. Put simply, it is easier to implement a single biochemical building block than it is to devise an adaptive learning algorithm.

On the other hand, smaller units can be used collectively to create more sophisticated behaviors (and more unpredictable outcomes) than larger ones. If the resources are at our disposal, the emergent behavior of a collection of fine-grained AI blocks that are marshaled using an A-Life paradigm is easier to build up and more sophisticated than a straight behavioral-model algorithm.

Clearly, choices made early in the design stage are crucial to being able to achieve the depth of gameplay that AI and A-Life are capable of. There are four broad categories that will dictate which building blocks are chosen to create specific instances of A-Life:

- Granularity: What level of AI and/or A-Life is required?
- Reproduction: Should the resulting algorithm breed?
- Behavioral model: Is the model prescribed, adaptive, synthesized, or emergent?
- Learning: What level of feedback is needed?

In each of the above, there is the reliability of the resulting system to consider. This might turn out to be a trade-off between the AI and A-Life behaving in a way that is completely predictable and sensible and using complex but less controllable A-Life to produce a richer environment. We have to make sure that if we dispense with the possibility to control that reliability, it will be acceptable in terms of the gameplay at the point at which the less reliable A-Life is deployed. Either that, or we need to exert higher levels of control; which will tend to make the deployment of the AI or A-Life less efficient.

The decision will weigh into each of the preceding areas. For example, a lower granularity may tend to restrict the A-Life such that it becomes more reliable, whereas a highly grained system, with many component parts all individually simulated, may tend (because of the emergent nature) to be harder to control and less reliable.

Reproduction, as you might imagine, can lead to less reliable systems, as it is harder to control. This can, to a certain extent, be tempered by the behavioral algorithms chosen—a reproductive system that works in tandem with a pre-programmed AI with limited choices can, even in a finely grained system, be made 100% reliable.

Any learning algorithms used will also need to be deployed with care; again, it is a question of reliability over richness of experience. If the learning is not adequately controlled, we run the risk that the "wrong" thing is learned, thus destabilizing a reliable system.

All of the preceding needs to be used in the decision-making process from the start. Each also needs to take into account its effect on the reliability of the system. At the same time, for each piece of the system, it may be possible to decide how reliable the result really needs to be. As long as both of these questions can be dealt with, the question of quality control and assurance can be resolved satisfactorily.

The two items that have not yet been discussed in terms of the building blocks are reproduction and learning. The reproduction aspect is easy to understand, but it has wide-reaching consequences. Essentially, we are asking a very simple question: Do we want the resulting algorithm (script, representation, numerical matrix) to be capable of reproducing itself with respect to another version of the same algorithm? If the answer is "Yes," then we open up a range of options for creating new versions of the behavioral model—perhaps improving them, perhaps downgrading them—which will eventually lead to slightly different kinds of emergent behavior. If the answer is "No," then we preclude the ability to select from a variety of versions of that entity, thus reducing the entity's effectiveness in certain situations. Referring back to our previous discussion on granularity, it is often simpler to breed a good result than it is to try and arrange for it via an appropriate algorithm.

So a simple question often has complex answers, and this is part of what separates AI from A-Life. AI is top-down; it tries to find that algorithm that leads to the desired behavior. A-Life, though, takes a starting point and tries to recombine its essential elements until a better version is achieved. A-Life might not achieve the most correct solution, whereas AI might achieve the most efficient, correct solution. However, A-Life will generally yield a more realistic, lifelike, and sometimes better-fitting solution for video games.

And it all starts with the building blocks—how we choose them and how we put them together—and it is "learning" that helps us to wire them together correctly. There are different kinds of learning and simulated learning algorithms, all of which can be used in AI and A-Life to provide input to the building blocks and mold the system.

The kind of learning used will, to a certain extent, depend on the behavioral model. For example, pathfinding that uses backtracking is a well-understood AI routine for solving a maze or best-path problem. Backtracking is fine for behavioral models that rely on this kind of prescripted decision-making—for example, when creating a river on a terrain (environmental modeling), water always chooses the path of least resistance. Another example is weighted decision-making, using formulae to determine the base outcome and then applying experience (based on weights) to reinforce the outcome, possibly changing it in the process.

Both of these are valid, and both are reasonably "clever." Neither is particularly lifelike, although they can be used to create some aspects of lifelike behavior. However, the learning aspect is very important as a controlling force in our AI or A-Life implementation; it warrants discussion before we take a detailed look at the various building blocks. It is the learning aspect of the behavioral models that will enable us to leverage the building blocks more effectively and provide added depth to the video game—and thus, more value for the player.

Learning Defined

Learning means different things to different people, but scientists agree on one aspect of learning (as illustrated by Pavlov) that remains at the core of many education systems today: reinforcement. Pavlov worked with "conditioned responses," automatic reactions tied to external stimuli. In his classic experiment, dogs were conditioned to expect food at the sound of a bell ringing. Over time, the mere ringing of the bell was enough to make the dogs salivate, whether they received the food or not. The response had become "ingrained."

In human learning systems it is possible to create similar links between actions, reactions, and rewards; even when the rewards are not immediately forthcoming, we can still be quite sure of eliciting a desired reaction in response to a given situation—at least for a while. This is called "training."

Neural networks can be trained. In fact, it is this training aspect that makes them useful in creating AI and A-Life systems. The artificial neural network is modeled on the observed physiology of human (animal) brains that we believe allows them to learn effectively. The training is possible because of the underlying physiology, which leads to something observable because cause and effect become linked in the brain.

Weighted Historical Reasoning

We also need to employ another learning algorithm that is less resource intensive—weighted historical reasoning (see Chapter 4 and the simple `FightOrFlee` tank function). Note that weighted reasoning is a reasonably accurate way to train a system using reinforcement, but is not really learning, per se. If intelligence is the ability to apply old solutions to new problems, then learning that is implemented as historical weighted reasoning is probably not going to be enough.

However, in terms of the kind of restricted-situation learning needed for video games, learning by weighted experience is good enough. As was pointed out in Chapter 1, "Introduction," even the most rudimentary biologically intelligent system has more resources available for information processing than can be imagined, so we need to match that breadth with an appropriate algorithm. That algorithm is what we call "weighted historical reasoning"—the ability to affect the outcome of a decision, based on an evaluation of previous decisions. Its origins are in maze-solving, which is a good illustration of a very coarse-grained weighted historical reasoning system.

We say that it is coarse grained because each decision point is a yes/no proposition. At each fork in the maze, there are usually only two choices. We cannot partially go down one corridor and partially down another. This makes the evaluation process easy to implement using backtracking. As the algorithm progresses through the maze, it places each yes/no decision on a stack. When the algorithm can go no further, it stops and tests whether or not it is at the end of the maze. If it is not at the end of the maze, it marks the last choice as bad, and unwinds (backtracks) the stack to the previous yes/no (decision) point.

If another choice is possible, the algorithm takes it and proceeds down the new path. Each time it chooses between options, it does so arbitrarily and puts the decision on the stack. When it reaches the end, it performs the test for success—or

notes failure and backtracks to the previous decision point, marks it as bad, backtracks to an earlier unmarked point, and takes the new path. Eventually, the algorithm will have visited all the points in the maze and will have tested all available paths (in the worst case), but the maze will have been solved. We will call this a system of "re-decisions"—a decision that is repeated, based on the outcome of a previous decision taken at the same point in a previous sequence.

This is an example of pathfinding and a handy approach to solving maze-like problems, where the decision points are fairly coarse, yes/no-style decisions. Where weighted historical reasoning borrows from this is in the retention of the decision processes and the relative success of each decision point.

The size of the state space will also have an effect on the eventual success of the algorithm. The more states that there are (places to visit), the more decisions there are to make, even if those decisions are simple yes/no or right/left values. In fact, in a binary system of fully connected states in the state space, adding a single state dramatically increases the number of possible decisions.

Clearly, again, there will be a performance and design trade-off to be made between the number of states, the granularity of the decisions, and the available resources that can be dedicated to the algorithm during the play session.

Extending the algorithm so that fuzzy decisions can also be taken is key to the success of the weighting part of the basic algorithm; it allows us to apply a decision with a certain strength. In effect, the decision-making process can be "trained" by using the weight of evidence from past decisions to influence how much of each choice to make when that decision crops up again. If there is enough data and enough resources, chains of weighted decisions can be simulated, and backtracking and re-decisions used to arrive at an optimum solution that is nondiscrete at each decision point. This also means that although the general "path" will be the same each time, the extent to which each solution is explored will potentially change.

The weighted historical reasoning algorithm replaces the learning of any of the (relevant) building blocks that follow. In effect, it is the first of the building blocks, and when implementing the AI/A-Life part of a video game, we have to decide whether historical reasoning will be enough, or whether a neural network will be required.

Whereas the weighted historical reasoning algorithm will need to be adapted to fit the building blocks for a given project, its implementation should be described to make it concrete. Thus far, this algorithm has only been discussed in a very abstract fashion. At the core of its implementation is an event-action matrix that contains a mapping of all possible events to all possible actions, based on experience. Each event-action pair also contains a weighting factor that can be adjusted, based on an evaluation of a given outcome.

As in the maze example, the outcome could be near term or far term. In other words, if a racing game has an event such as "100 meters from 45-degree right turn," there might be several possible actions—speed up, slow down, turn left, turn right, and so on—from which the actual response to the event will be built. The outcome might be near term—an immediate crash, for example—or far term—the lap time or final race destination. Given a starting matrix by the design team (which

would logically include for this particular example some slowing down and possibly some movement to the right, into the apex of the bend), it would be tempting to fiddle with some of the weights to see what happens.

So we take a copy of the entry in the matrix and change one of the weights and test the outcome. If our car crashes, the entry is reset. If our car does not crash, then we note the decision taken and move on. When the lap is finished, we might decide to upgrade the matrix slightly to reflect the decisions made during that lap.

All the decisions to the new weights are not set blindly, however. The weights are just slightly adjusted (there might have been some bad decisions) to reflect this case's new lap time. Over time and through many laps, we might eventually settle on a more or less stable matrix that is an improvement on the initial one.

That is how a weighted historical reasoning algorithm is implemented. In short, it has several components:

- A way to map actions to events,
- An evaluation of cause and effect (outcome), and
- The upgrade/downgrade algorithm.

The result is basic learning by reinforcement. As such, it cannot be applied to new situations; each event-action-outcome triplet has a relationship that only makes sense in a particular gaming scenario. In addition, it is difficult to create an abstract of the algorithm that can be applied to all situations, but it can be adapted to fit each building block.

If there are links between contexts, however, the matrix can be tried on new situations—for example, in our racing example, we could use the same matrix for different vehicles. If hard links were then created between different gaming contexts, the matrix might be able to be shared in new situations, such as for aircraft or boats.

If this is the kind of learning and reasoning system that is to be put in place, there is a decision to be made on the granularity of the state space. Put more simply, there needs to be a fine enough level of granularity that the states and decisions can be adequately represented such that the system is capable of learning and reasoning, but it needs to be coarse enough that the state space (number of states/decisions within the network) is kept to a practical size.

Even a simple learning and reasoning system in a complicated state space will take up a large amount of resources. As far as it can be, the learning and reasoning state space needs to be abstracted away from the complexity of the underlying system. The game designer needs to realize that not every state in the game needs to be reflected in the learning system—only those states that are not inevitable.

Some states and state changes will take place no matter what happens. If the actor in the system cannot deviate the path between several states at a given moment in time, there is no point in trying to map each state to a part in the reasoning system. As far as is possible, the states should be reduced to an effective minimum—there simply will not be the resources in such a complex real-time control system to represent the entire state space.

Finite State Machines

A finite state machine is best described as an algorithm that has an internal state that can be in one of a variety of states at any given moment in time. Each of these states is discrete—in other words, an FSM can only represent one state at a time and will remain in that state until another state is imposed.

Each of these states may have actions associated with them: an entry action, an exit action, or both. An entry action is something that happens (or is caused to happen) when the state is entered, and an exit action is something that happens when the state is exited.

FSMs are computationally inexpensive as well as easy to understand and a good model of real life. Therefore, they are deployed in video games on a fairly regular basis, especially when behaviors can be broken down into discrete states.

The algorithm moves from state to state by accepting triggers (events) that cause it to change states and perform appropriate actions if necessary. For example, if we want to model an electric sliding door, we can construct an FSM that couples the door and sensor mechanisms. The door can be either open or closed; opening the door is triggered by the player pressing a button, and the expiration of a hardware timer (triggered by the open door) triggers the door's closing. We can then model this with a state transition diagram (Table 5.1).

TABLE 5.1 FSM STATE TRANSITION DIAGRAM

	Closed (A)	Open (B)
Button (X)	B	—
Timer (Y)	—	A

From this we can determine the basic movement between states, but we cannot discern the various actions that can take place when the transitions occur. In addition, there is no link between the timer and the door being open/closed. The notation can be extended to use a state transition table that contains additional information relating to these actions, as shown in Table 5.2.

TABLE 5.2 FSM STATE TRANSITION TABLE

State Name	Condition	Action
Door Closed	Button Pressed	Start Timer
Door Open	Timer Elapsed	Close Door

Table 5.2 is also not entirely satisfactory. For example, the exact nature of the relationship between the door and the timer is still not immediately clear. So we can depict it in a graphical form, as shown in Figure 5.1.

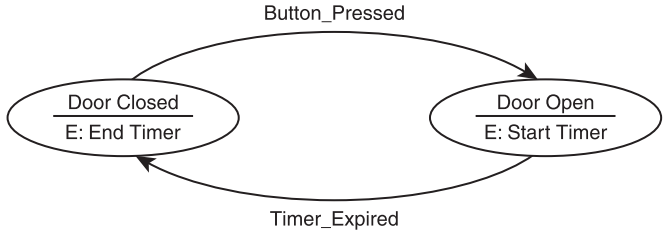

FIGURE 5.1 State transition diagram (graph).

In Figure 5.1, we can clearly see that there is only one exit action—Start Timer. The timer itself is not contained within the FSM and is assumed to be an external object.

This is an example of a *transducer*; output results are based on inputs and states combined with various actions. They are very useful in behavioral modeling that does not necessarily follow a sequential pattern, but allows (random) movement between states, based on triggers. Within this category there are two traditional models. The Moore model uses entry actions, with the output depending on the state. This allows us to vastly simplify the behavior, but will yield a proportionally higher number of steps. The alternative is the Mealy model, which uses a combination of the input and state to generate the output. This reduces the actual states; each state is more complex, which can be both an advantage and disadvantage.

A mixed model was used in Figure 5.1, a practice that is fairly common. For our purposes, the pragmatic view is preferred—flexibility is better than correctness or adherence to a single model.

There is another type of FSM, a *sequence detector*, in which each state merely tests a Boolean (or otherwise binary) condition. If the condition is evaluated as true, the next state is achieved; if not, then the routine moves to an error state. The evaluation of the last step either returns to an error condition or flags success. Either way, the FSM will not be in any state at the end. This is different from most FSMs in which each state is transient.

Of course in video games, we have a slightly different need. Our FSMs are often used to map behavioral models that will include a final destination—the destruction of the entity being managed by the FSM. Therefore, even our transducer FSM networks will have a terminating state. Sequence detectors might also contain an accepting state that defines the point at which the FSM has successfully completed analyzing the input data. It can also be the starting state, and as such is useful in modeling certain kinds of deterministic behavior in which we need to test for a given sequence and return when that sequence has been satisfied.

We can use FSMs to emulate behavioral models at several different levels. Bear in mind that these levels can be scripted or encoded. In other words, a scripting mechanism can be created to specify the behavior in terms of an FSM (a technique introduced in Chapter 4). Extending this basic usage, an FSM can be used at the highest level to describe a mechanism that allows us to change between different behavioral modes. For example, imagine a guard with five behavioral states—patrolling, active patrolling, attacking, defending, and dead. Moving between these five states can be governed by various outside triggers:

1. The player being noticed: patrolling ➔ active patrolling
2. The player being identified and targeted: active patrolling ➔ attacking
3. The ensuing battle: attacking ➔ defending ➔ attacking …
4. The end result for the guard: defending ➔ active patrolling, *or*
5. The end result for the guard: defending ➔ dead

Each of these behavioral modes may well have another FSM inside it that describes the actual state during that mode. So in the attacking behavioral state, the guard may be transitioning between a number of aggressive states.

Where an FSM can have its drawbacks is in requiring that the system be represented in terms of purely discrete states. As a behavioral mode selector, it might be perfectly adequate, but some actions, such as driving a vehicle, might not be so easily represented in such discrete units. This is where the fuzzy FSMs offer slightly more flexibility over regular FSMs.

Fuzzy Finite State Machines

A fuzzy FSM (FuFSM) allows for nondiscrete states and transitions. Unlike the FSM from which it is derived, a FuFSM removes this restriction; the machine can be partially "in" a number of states at the same time.

It also removes the restriction that requires the selection of a single new state to move into by allowing us to specify possible multiple new states. Therefore, we can divide the FuFSM category into two subcategories:

- Fuzzy state FSM
- Fuzzy transition FSM

The fuzzy state FSM algorithm allows the current state to comprise membership of more than one state to differing degrees. This is helpful when describing states in a model such as driving a vehicle, where we might be simultaneously turning, braking, or gearing up/down. Granted, this might overcomplicate what should be an easy building block to conceptualize, design, and implement as part of a video game. Therefore, we should break down the system into smaller blocks rather than introduce FuFSMs. In the case of the vehicle-driving example, this would mean that turning, braking, and gear selection are treated as three separate FSMs, possibly linked by an overseeing FSM, as previously discussed. The designer should at least be aware of this option and is free to decide which method is most efficient.

In the FuFSM model, triggers then become events plus a weighted portion of a given state, rather than just prompts. So the transition from one state to another, including a partial transition to other states as well, becomes a case of evaluating the extent to which we wish to apply the current logic in these other states. This allows us to trigger state transitions based on varying values, such as health, simulated stress, hunger, aggression, and so forth. Again, this is quite useful in video games when we need to emulate animal behavior, which is more or less by definition fuzzy, and which might not always fall into discrete behavioral patterns.

Like "crisp" FSMs, fuzzy state FSMs can be used at the highest level to trigger state changes between behaviors—for example, the guard moving to a higher degree of curiosity when a threshold input value is reached, rather than just moving crisply between the two states. This allows a greater sophistication in the ensuing behavior; the guard becomes increasingly curious and more likely to engage in behavior (within that state) that indicates this. This mirrors human behavior in a similar situation. Rather than changing from a low to high state of alert, we tend to just increase our attention to the details until we find out what has alerted us in the first place. Unlike crisp FSMs, it is quite difficult to use fuzzy state FSMs at the lowest level because, when we get down to individual actions, the problem domain (the states) tend to organize themselves naturally into discrete units.

The second subcategory of FuFSM, the fuzzy transition FSM, is slightly easier to understand and deploy. As mentioned, it allows the FSM to consist of multiple possible state transitions. Only one can be followed at a time, but these transitions can be selectively triggered.

When implementing a fuzzy transition FSM, a state transition is chosen based on an applied weight (or even random selection) that reflects an arbitrary decision-making process that does not adhere to a strict transition table. Some transitions are impossible (and those restrictions remain), but the rest can be chosen by the algorithm during the play session. Our guard, for example, in moving to a more alert mode, might switch states from looking one way to another or moving forward or backward, with his eyes scanning from side to side. Each of these can be encoded in a state diagram that would dictate a more or less sequential process.

Were we to apply a fuzzy transition FSM, the behavior can be made more natural by introducing some randomness into the transition selection process. This means that some of the time, an unexpected action will be chosen. As a mechanism to increase realism, this works quite well. However, it can also be applied as part of a learning system in which we feed back success rates into the fuzzy transition FSM. In this way, we tend to lean more heavily toward one transition than another. This kind of pattern-recognition learning by reinforcement is more closely associated with the next building block: the neural network.

Building a Neural Network

A neural network is modeled to imitate animal thought processes (including memory). From our perspective, this kind of building block is usually deployed at the

system level. It is there to guide the system and is not usually implemented within individual entities.

Of course, there are exceptions. As we shall see later on when we discuss choice and deployment of building blocks, there are some games that revolve entirely around the use of AI and A-Life, and they may well deploy simple neural networks as part of their simulations. But for the most part, it will be assumed that neural networks will be deployed to help the system learn, recall, and predict with greater accuracy than is possible by using FSMs, FuFSMs, weighted decision-making, or the other, predominately static building blocks that have been discussed up until now.

That is not to say that they cannot be used in conjunction with these other mechanisms, but neural networks are resource intensive enough that they will take up too much of the available resources and are best deployed with care. It is also worth noting that of the many games that start out slated for neural networks, many shelve the idea as it rapidly becomes hard to understand and quite complex at the implementation level. So here, we are going to remain with easy-to-understand, easy-to-deploy uses of neural networks, starting with the basic building block: the neuron.

Neurons

A neuron is a simple enough logical device. It accepts one or more (usually several) inputs and provides a single output. The inputs can be wired to the outputs of other neurons, and its own output can be wired to other neurons or read directly. The output fires (produces a signal) when the internal state of the neuron reaches a given threshold value. This internal state is calculated by taking into account the input value, which is weighted. That value represents the strength of an incoming signal's importance. The weights should add up to a given maximum value that should be equal to or greater than the threshold. If we multiply the weights by their inputs and add the results together, that value must be higher than the threshold in order for the neuron to fire.

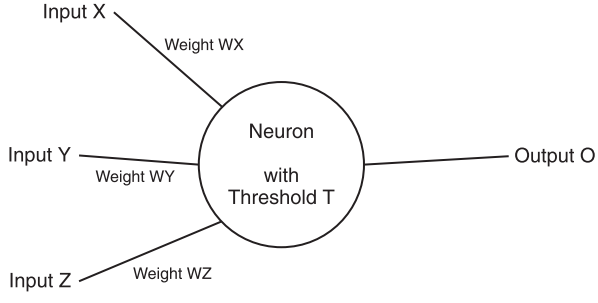

FIGURE 5.2 The neuron.

If we assume that the neuron in Figure 5.2 works entirely with binary inputs and outputs, then the input values will be equal to the weights when the input is at maximum and zero when it is at a minimum. If the weights are evenly split across the three inputs, giving each a weight of 0.33, it is clear that a threshold of 0.99 will make the neuron fire when it receives input from all three. When those inputs are connected to a series of sensors, then we have created a trainable decision-making unit with a single neuron. In other words, the outcome can be changed by adjusting the weights.

We can assign meaning to the inputs and expose the neuron to patterns of inputs on the one hand and the desired output on the other. Training then just becomes a case of adjusting the weights according to the desired result.

The result is known as *classification*, and neurons are very good at making binary decisions based on multiple complex inputs. For example, we can train a neuron to tell the difference between a "1" and "0," based on a grid of cells. Each cell is then an input to the neuron. When the neuron is shown a "1" pattern, the weights to specific cells are strengthened. When a "0" is shown, nothing happens. These grids can be seen in Figure 5.3.

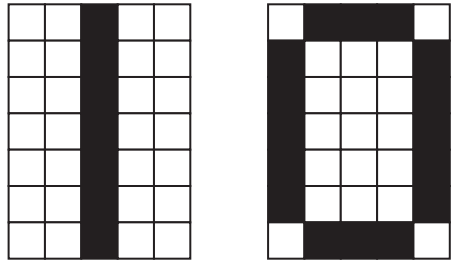

FIGURE 5.3 Character grids.

The strengthening approach described here is only half of the story, because the same "0" response will be given for an unrecognized arrangement of cells as will be for a zero. The certainty inside the neuron might not be the same, but we cannot see that. All we see is the output. Mathematically, the equation that changes the weights is currently as follows:

change_factor = output_level * weight * limiting_factor

The *limiting_factor* ensures that we only make small changes to each weight in response to the desired *output_level*. The *change_factor* is then applied to the weight strength of the input neuron.

If you work through these multiplications without limiting the change by building in the *limiting_factor*, you will quickly see that if the *output_level* varies by only a small amount, the result can be wildly different values in the *change_factor*. This is a problem because if the *change_factor* is allowed to vary too much, the result will be massive swings in behavior.

In fact, it may get so bad that the system oscillates between two extremes of behavior as it attempts to stabilize on one weight value by over-compensating each time. We shall see a little later on how a sophisticated system can be built up whereby the *limiting_factor* can be made to be dynamic, which improves the capability of the system without producing these wild behavioral differences.

We therefore maintain two sets of neurons—inputs and outputs—and use the desired final outcome ("0" or "1"), coupled with the pattern that leads it (in the output pattern) to generate a weight change that will be applied to the strength of connections to the input neuron. Then, when signals are applied to these connections, the neuron will fire if the result is above a given threshold—or as close to "1" as is deemed appropriate. We are only strengthening the weights.

There will be occasions, however, when it is desirous to weaken the weight associated with an input in order to enable more accuracy in classification. This requires that our equation change in one of two ways: We can choose to reduce the weight based on either the input neuron being active or the output neuron being active. In each case, the opposite neuron is inactive. But in both cases, a negative weight change must be generated when there is a mismatch.

Reducing the weight based on the input neuron being active, (pre-synaptic, input neuron awake) implies that we want to provide a negative weight that is proportional to the margin by which the output neuron is "asleep" with respect to the input neuron. Mathematically, this is as follows:

```
change_factor = (2 * output_level-1) * weight * limiting_factor
```

The post-synaptic equation then decreases the weight when the input neuron is asleep, and the output neuron is awake:

```
change_factor = output_level * (2 * weight-1) * limiting_factor
```

So far, we have learned that we can set up neurons, show them patterns, and make links to the inputs that enable us to detect, by adding the relative weights together, whether a grid of cells is equivalent to a "1" or not. In video game terms, this equates to being able to take a vast number of inputs and reduce them to a single output. With appropriate training, this can be used to make a simple yes/no decision from complex and changing input data.

If we need to make several such decisions, another neuron can be added that might even share some of the inputs, but with its own weights attached to them. In this way, we move from binary decision-making to something more complex.

For example, assume we have a robot that can move in one of eight directions and fire. These are represented by nine neurons. Each neuron has information that comes to it from eight different angles, representing the distance to the nearest object (assuming that all objects are threatening and there are no walls). At each moment in time, the information is gathered from sensors (inputs) and fed into the neurons, and the neuron that fires the strongest is picked—the neuron connected to the gun, the one making the robot fire, the neuron connected to the "go north" command, and so on.

The robot can now be trained by accepting or rejecting given decisions. If a decision is accepted, we strengthen the inputs that led to it; otherwise the inputs are weakened. Over time, our robot can be trained to advance toward objects, fire at them, or just run away. However, if the robot is presented with a completely new experience (in terms of the known inputs), it will have to make a decision. That decision, after training, should be the best, given the training that we have offered.

We do not need to know every possible input in advance. In other words, we do not need to prepare for every situation or provide a path through the system that correlates to every possibility. This differs dramatically from FSM and FuFSM building blocks, where possible routes through the various options need to be modeled in advance. The module can be trained to make decisions in a different sequence, but we cannot present a new situation and expect it make a right decision.

Feedback and Training

The previous discussion is a high-level look at something called *feedback* in the neuron. It is a form of training where the actual result is compared to the expected result, and the neuron's input weights are adjusted accordingly.

The various input patterns that are used are known as the *training data*, and for each set of training data, the expected output must be supplied along with the inputs. In this way, we can train the neuron by adjusting the weights associated with the input data with respect to the deviation from the expected output.

So if the output is slightly different from what's expected, we can say that the response of the neuron is "almost" correct, and we need only change the importance of the input data slightly. On the other hand, if the deviation is large, we must change the weights associated with the individual input signals more drastically, to allow the neuron to adapt itself according to the training data.

All we need to know is what the neuron's inputs are and what the possible output should be from a discrete collection. Whereas a single neuron can be multiplied to emulate different decision points from a collection (as was seen), or a system trained to take the actual output (ranging, say, from zero to one) and use that to perform a weighted action, it is still just a collection of neurons and not a neural network.

Neurons on their own could be enough for decent AI/A-Life system building blocks. However, we should look at putting these neurons into clusters and see how much more they can do without drastically increasing this method's complexity.

The Network

Typically, we want to make a decision that is more than just the probability that a single outcome is correct. So an output neuron is needed for each of the possible positive decisions. In the previously discussed number grid, we stated that the single neuron with 35 inputs (seven multiplied by five cells) could be trained to classify between one and zero.

That was not strictly true. Actually, it can only classify between one and everything else, because everything else would resolve to a value below the firing threshold. Instead, we need two neurons; one should fire if the result is one, and the other should fire if the result is zero.

The way to implement this is by setting up three layers: an input layer (in this case with 35 neurons), with the output of each connected to a hidden layer, which is in turn connected to an output layer (in this case with two neurons). This is known as a "multilayer perceptron" (MLP) and is fairly complex mathematically. Added to the complexity is the type of function used to calculate the output of the neuron—a threshold and usually a sigmoid (or S-curve) that produces a value between zero and one for inputs that represent large negatives or large positives. The reason for using this is because it closely mimics the way that the brain functions when firing neurons. Even though it digresses from our building-block discussion somewhat and into topics much more complex (but highly useful), the construction of an MLP is worth considering for a moment in terms of the neuron model presented in the previous section.

Previously we only had inputs, weights, and outputs to worry about. A real neuron, as depicted in Figure 5.2, also has a threshold value. And we no longer have a collection of neurons in the same layer; three layers are connected together to transform 35 inputs into two outputs via a hidden layer comprised of an undisclosed number of neurons.

When inputs are applied to the input layer, the neurons fire as before, based on the sum of their weights, but this time they are passed through the sigmoid function before being used as the output value. This output value is then used as the input to the hidden layer, in which neurons perform the same summation of their weights before triggering the output layer.

The output neurons (we hope) are significantly different in their activation output, based on the input weight summation from their inputs (the output of the hidden layer), and give a clear answer. One or the other must fire with some certainty—that is, one near 0 and one near 1:0.02 versus 0.98 would be nicely accurate.

Clearly, the influencing factors in all of this are the weights that connect the neurons at each layer. In order to achieve the right output, the weights must be set to match the input patterns. To do that, we need to train the MLP.

Training the MLP

This is where the complex math starts. An MLP is usually trained using back-propagation, of which the historical weighted reasoning algorithm we discussed earlier is an example. Back-propagation can be seen as using historical information to update the network to reflect experience. In the case of an MLP, we want to use the error between actual and desired results to recalculate the weights on the hidden and input layers' neuron input/output connections.

So the network is run and the results examined. Assume that it is 0.4 and 0.6—that is, tending toward the zero neuron. This might not be a high enough confidence level for us, so we calculate the error based on the desired level of confidence. If the

input value was, in fact, a one, this would be 0.9 and 0.1, respectively (allowing for a 10 percent margin of error). The output error is thus:

```
0.9: 0.4 = 0.5, and
0.1: 0.6 = –0.5.
```

These values need to be used to change the weights that were fed into the output-layer neurons from the hidden layer. Since there is no direct link between the hidden and output layers, the previously calculated errors must be summed and used to determine the likely input errors. Knowing the error factor for each neuron in the hidden layer is only half of the story, because we also need to change the weights that convert the activation values from the hidden layer into the inputs for the output-layer neurons, which is a simple multiplication.

Then we do the same for the input values to the hidden layer, which are actually the outputs from the input-layer neurons. The threshold values must also be trained, but a static bias of –1 is used as the input, along with a limiting factor and the error for the neuron in question.

This back-propagation needs to be done for each training run. The training run needs to be done as many times as necessary in order to have the result tend toward the desired result.

A final point: The hidden layer can include as many neurons as needed—as long as there is a mapping between the output layer, hidden layer, and input layer, that is. However, remember that resources might be stretched if there are too many neurons in the system. Each one has a collection of weights that is proportional to the number of neurons connected to it. So if there are 35 input neurons and 10 hidden neurons with 2 output neurons, then there will be 35×10 weights between the input and hidden layers and 20 between the hidden and output layers. This might not seem like very many, but they tend to increase when more complex decision-making networks are required.

It is quite clear, if we have a three-layer MLP, that the input layer represents the sensors (or other input information), and the output layer represents the action (or output information). The hidden layer then provides a logical mapping between the two. But how do we decide how these layers interconnect?

For that matter, how do we decide how many neurons ought to be in each layer?

Deciding how many input layer neurons should be used is easy; there needs to be one for each piece of information to be evaluated. Similarly, the output layer, representing the actions that are taken having evaluated the input layer, can be mapped to possible actions in the game environment, which is a known quantity.

The hidden layer is slightly more complex. The first thing to note is that, in an MLP, there should be at least one and no more than two hidden layers. One hidden layer allows us to represent decisions that are nonlinear.

Two hidden layers would allow us to represent functions that go one step beyond this and can represent mappings between entirely arbitrary decision boundaries with any accuracy. These are rare, and as noted earlier, for performance reasons, we

should try to limit the complexity, so it is assumed that one hidden layer will be adequate for any game-related problem space that you are likely to encounter.

The number of neurons in the hidden layer should be enough that the complexity of the decisions in the problem space can be represented, but not so many that the range of input values cannot train all the neurons in the hidden layer. Underfitting will occur if the network cannot represent the complexity of the mapping from input to output, and overfitting will occur if there are too many neurons in the hidden layer.

Not to mention the fact that the network will take too long to train if there are more neurons than absolutely necessary. If the network is to be pretrained, this may not matter, but if it is to be provided as a real-time learning and recall mechanism, this will be an issue.

So how do we decide? Firstly, according to a paper (http://www.heatonresearch.com/articles/5/page2.html) by Jeff Heaton of Heaton Research, there should be a number of neurons between the sizes of the input and output layers. This number should not exceed twice the neurons in the input layer and, as a guideline, should be 2/3 the size of the input layer, plus the size of the output layer.

Jeff Heaton also proposes that the fitness of the network can be evaluated and the number of neurons in the hidden layer adjusted according to the success of the network in training and recall of specific patterns. This, for those who have the time and inclination to experiment, seems like a good approach.

As we have already seen, the MLPs proposed in this book are n-connected. That is, each neuron in the hidden layer is connected to each neuron in the input layer. Likewise, each neuron in the hidden layer connects to each neuron in the output layer. This need not be the case, and a similar trial and error process can be used to remove connections and test the effectiveness of the network.

Again, this is only important if resources are at a premium, and behavior is not affected by removing connections. The testing that would be required to implement this kind of selective connection algorithm might well outweigh the benefit of being able to have a network with exactly the right number of neurons in the hidden layer and exactly the right connections between neurons.

It also might not be practically achievable, at which point the approximations that we have mentioned here would be the best approach.

Quite a lot of time has been spent discussing neurons and neural networks, so how are they used in video game programming? As mentioned, classic MLP can be used at a high level to decide the extent to which a series of actions should be taken, given a set of related inputs. This makes it a little like an FuFSM, except that we can train it to approximate behavior in new situations. For example, the neural network for a driving game can be trained to steer around a track, given input from various sensors that compute distances from turns, the depth of the turns, the current speed, and so on. This network would be trained by example; if the vehicle tends to crash for a given set of weights, the weights are updated with knowledge of this experience to prevent it happening again. Theoretically, the driver gets ever better at steering around this track in particular—and hopefully tracks in general. However, there are much easier to deploy neural networks that can be used in less fuzzy situations—recall and prediction—and these are true building blocks for AI and A-Life.

Recall

Neural network theory can be used to build simple associative memories. In an associative memory, we try to arrive at a network of weights that link an input pattern to an output pattern. These weights are then used to regenerate the output pattern from the input pattern.

So we have two layers—an input layer and an output layer—which are linked by a single network of weights in a one-to-one relation. In other words, there is a single weight from one input node to an output node. Training the network is very easy. All we do is create the weights by multiplying together the node values, and the result is stored in the linking weight. There is a slight caveat: The binary values are not 1 and 0, but 1 and –1. To calculate these from 0 and 1 binary values, we use:

value = old_value * 2: 1

Assuming that we have an array of weights, represented here in C as follows:

```
int weight_array[num_input_nodes][num_output_nodes];
```

we can then calculate the weights based on a very simple double loop:

```
for ( int input_node = 0; input_node < num_input_nodes; input_node++)
    for ( int output_node = 0; output_node < num_output_nodes;
output_node++)
        weight_array[input_node][output_node] +=
            (input_layer[input_node] * 2–1) *
(output_layer[output_node] * 2–1);
```

The interesting part is that we can train this array of weights (our network) on various patterns (hence the additive nature of the calculation), and it will still be able to more or less regenerate one set of states from the other. The regeneration part is equally easy. All we do is reset the input layer and then fill out the information that we know; it might be all the nodes, and it might not. We can then run the weights over the input nodes one by one to calculate the output nodes.

The array of weights is, however, bidirectional. Because an associative network has been created, we can feed the generated output pattern back through the array of weights that will be added to the input pattern. If this process is repeated until the patterns are stable, then we can be reasonably sure that the information within the output layer reasonably approximates the memory of the originally trained, input layer pattern.

Experimentation will reveal several useful behaviors for video games. One such behavior is that, given a partial input pattern, the associative memory will be able to reproduce a partial output pattern that is sometimes more complete in terms of the information originally provided. This means that we can use it to make decision maps that might tend toward a specific behavior, since partial inputs that trigger that behavior have been seen. As with neuron mapping, a relationship must be established

between the input and output patterns. However, in the case of associative memory, these tend to be one-to-one relationships, as can be see from this discussion. An alternative is a self-associative algorithm that can reconstruct a pattern from partial information, piece by piece, given only the single input pattern. This is also known as a Hopfield net.

Prediction and the Hopfield Net

The Hopfield net is the simplest of all neural networks. It has a single layer of neurons, each of which receives an input from outside the network and an input from each of the other neurons within the network, taken from the output of each node. The final output is only harvested from the neurons when the network has reached stability—that is, the neuron outputs are no longer changing. Each of the input/output links has a weight associated with it; the inputs for the neurons are a product of the output value of a given neuron and the weight associated with it.

Training the network only requires exposure to a library of patterns. Each exposure will affect the weights, which are recalculated based on the relationships between the input values (applied once) and the output values being fed back. If we then give the network a partial pattern, it will feed the information around the network until the final output contains the reconstituted pattern, which should be the same as a pattern from the initial library.

Experimentation will show that sometimes the network oscillates between two pattern states, and testing will be needed to limit this behavior. These networks are still useful in predicting output datasets from incomplete input datasets and can be very useful in video games where second-guessing the player is part of the desired AI. They are also useful in simulations when we might want to arrive at a given situation and only know what a few of the variables might be, and we need settings for the rest. As in the other building blocks, however, the system needs to be broken down into fairly discrete units, tending toward binary values.

Expert Systems

Guided decision-making is the process of creating a system that starts from a given premise and repeatedly asks questions until it settles on an answer. An expert system is one in which questions can be asked that are designed to enable the system to narrow down possible options until a decision is reached. The behavior emulates a real-life expert, such as a medical practitioner, who progressively asks about symptoms, narrowing down the possible causes for illness, in an attempt to diagnose the patient. In the video game problem domain, expert systems can be used in a similar fashion—to narrow down the solutions toward the next best move or strategy.

This concept is fairly abstract, and it will be briefly discussed without concrete examples. The reason is because implementations will be specific to the projects in which expert systems are deployed. A generic framework for expert systems might be feasible, but each deployment will be different.

The core of the theory is rather like the game 20 questions. We can train a system to ask the right questions and get to a response that is probably good and within a margin of error, based on the answers. In fact, there are 20-question implementations that use an expert system to do exactly that, with the results fed back into the system for future use.

So an expert system can function on a purely question-and-answer level, but it can also learn. In order to do so, the system will likely use a historical weighting algorithm or possibly back-propagation (as was discussed earlier in this chapter) to approve the network of choices leading to a specific answer.

In effect, when an expert system is created, either we need to give it all the data it needs to make choices, or we must train it. Giving it the data could be as simple as creating a text file with a list of common questions, their answers, and the resulting questions or final outcome. This data could also be expressed in code or as a script (see Chapter 4) that is applied during the play session. Or we could choose to train the expert system from the ground up.

Training the system from the ground up necessitates a network of choices, with weights indicating what the relationship between the nodes will be. This is analogous to a neural network consisting of many layers, with a slight adjustment possible. We might choose to have discrete paths or provide paths from every neuron (node) to every other neuron (node), with a weighted chance of taking each one. Another way to look at this was previously discussed—a Hopfield net.

A neural network does tend toward combinational behavior in which effects combined to produce triggers that can be used as inputs to other nodes. A neural network expert system is also possible in which new nodes are created as we train it with new situations. This takes the otherwise linear expert system to a new level—possibly a step too far for many video game implementations, but perhaps useful for combining some of the building blocks mentioned in this chapter.

In the run-time part of the system, a neural network can also be used to create a path through a given situation, based on trail and error. The expert system then becomes, in effect, self-trained opposition, or even a self-trained helper for the player's character.

Building Blocks Review

The beginning of this chapter covered the basic AI building blocks that are within the resource constraints of a video game. Even though we have tried to work within those constraints, some of the more complex blocks will fall outside of them. The most immediately accessible are the finite state machines (and fuzzy variants), Hopfield net, associative memories, and various types of expert systems.

For now, we will discuss AI and A-Life as they pertain to 99 percent of the video game market. The previously mentioned building blocks are purely AI mechanisms that are well understood and researched and upon which A-Life can be created.

There are also blocks that can model lifelike systems. Combining these AI blocks with scripting mechanisms and A-Life blocks can provide an improved playing experience. This is the main thrust of the rest of this book. The details of these techniques will flesh out as we go through chapters covering emergent behavior, A-Life testing, and some fairly concrete examples, before learning how to deploy them in video games.

First, however, we need to look at how the blocks covered so far can be deployed so that a better understanding of their combined strength is achieved. Therefore, the rest of this chapter is dedicated to mapping the blocks onto the gaming environment. Notice that there are certain constraints that restrict us from just picking an algorithm and using it directly. Quite a lot of groundwork must be done in the design phase in order to leverage AI and A-Life techniques in the implementation.

Deploying the Building Blocks

This chapter discusses the major aspects of designing an A-Life system that starts with the right kind of blocks. For example, if you are creating a first-person-shooter and want to use A-Life enemies, then evolving them with low-grade biochemical building blocks is unrealistic.

What follows is a discussion of the theoretical building blocks that are needed and that can be deployed using the building blocks discussed in the previous sections. First, the various constraints and qualifications should be restated. These (for a large part) restrict our implementation of AI and A-Life paradigms.

For example, chemical and biological modeling do not find their way into many games, but they can be modeled using some of the building blocks from the previous section, such as finite state machines and neurons. In combination, they can model the interaction of chemicals in a biological system.

We can extend this idea to model processes such as hunger and fear. An enemy's approach might cause fear in an in-game monster. However, that same monster might be hungry, and these sensations can be connected together in a primitive neural network that enables the response to fear, weighted by a need to satisfy hunger.

The tipping point at which hunger overpowers fear will lead the monster to approach the enemy rather than run away, and this can be modeled as a motor response to the network. The output from the synapses firing in the primitive biochemical neural network, combined with various FSMs to model the behavior, is a ground-up style A-Life approach. However, normal computing power does not allow a completely ground-up approach. We should find another way to emulate the behavior of our monster, unless that is the actual core idea of the game—as in *Black&White* or any of the *Creatures* series.

If it is not, then the right building blocks must be chosen. Even a complex set of reactions (such as those described) can be modeled using the more basic building blocks mentioned in the previous section, rather than a complete (albeit primitive) neural network solution.

A simple fuzzy FSM might suffice, or a form of Hopfield net that manages the behavioral models in a way that allows feedback. We could even use an expert system with appropriate weights, which can be modified to mimic the effects of approaching starvation that overpowers basic fear.

Of course, we can go in the other direction, if need be, and emulate a brain that uses an entire MLP neural network from the ground up. Coupled with associative memories and other mechanisms, a reasonably sophisticated A-Life being can be created.

Or, a metabolism can be added that comprises a collection of neurons to simulate breaking down food into energy and feed information into the rest of the system. Part of this system would include using a neural network, coupled with the internal and external stimuli in order to differentiate pain from pleasure, in an attempt to help the entity learn when it needs to eat and what to do about it.

The previous section covered the basic elements available to AI programmers; these are the templates that can be built into more specialized building blocks for each individual project. The A-Life part of the equation helps us link these units together, enabling a lifelike and not just random result.

In-Game Representation

Part of connecting the building blocks together correctly involves creating the right interaction between in-game elements. This starts with being able to choose the right in-game representation; then we can decide how the various elements interact and use the AI building blocks to create A-Life by modeling the elements and interaction to create behavior.

During gameplay, the interaction between elements describes how actions affect nearby entities. The game design will show what those interactions are, and they must be mirrored in code that exists purely of virtual entities—possibly scripted, possibly not. The finer the system's granularity, the more complex the interactions will be, and communication between the elements will be mirrored in these interactions. This action and reaction approach works well for interactions where there is no explicit communication.

However, in-game object-to-object communication is also possible and is the way that elements make their intentions known. This communication might trigger an action by another element in the system. Again, we can model this using the AI building blocks, but unlike direct interaction, there is no direct link between the entities.

The source code is also virtual, so communication can be modeled or directly implemented within it. Although the communication is explicit, the collection of elements needs to witness that communication (hear or see it, in a virtual sense) in order to react. Therefore, stimuli to the AI implementation come from the elements' own simulated senses and *not* as a direct connection between two elements, as before.

Before moving on to how the building blocks can be leveraged to emulate behavior, we should address granularity and put these issues into the context of the two extremes of video game design—games in which the A-Life is the game, and

games in which the A-Life becomes the same as other in-game elements. Where A-Life is simply used for behavior and behavioral modeling, the two are not the same. In other words, the elements of the A-Life system are there to guide the in-game elements. These elements call on routines to manage their discrete behaviors, and these routines may be part of the individual entity's makeup or not.

Again, this depends on the granularity of the system. The finer grained the system, the more we have to place in each entity. The coarser the grain, the less AI and A-Life matter in the grand scheme; we can pull more of the various implementations of the building blocks away from the entities themselves and put more under the control of the system.

This, then, has an effect on the way that communication and interaction are managed. The less each entity is responsible for these tasks, the more we can model into the system. Some of this will rely on using a paradigm called "templated behavior."

Templated Behavior

Part of the templated behavior paradigm involves creating AI and A-Life entities based on predefined templates, as opposed to the potential chaos that can result from a completely unprepared or random population. Many of the building blocks define a way to model the behavior and interaction, but very little has been said about how we can prime these for use.

For example, in a game session, we need a starting population. It is very rare that we can begin with a truly blank canvas. Even a neural network needs to be populated with some input values in order to create the initial system of weights that enable it to make decisions.

An FSM needs to be set up with an initial set of states and ways to move between them, as does a fuzzy FSM. Any scripted AI also needs to be primed with correct first-order intelligence. Neurons need to be given initial thresholds, memories must be created—either empty or with some default patterns. All this requires that we create a template from which the evolving game can derive new behavior (if required). Some AI routines are best left static if they have no real impact on the game or are at a low enough level (fine enough granularity) that they will not change substantially over time.

There are cases where the game includes some principles to marshal the creation of problem and solution spaces during the game session. This "guiding hand" of the AI system uses a top-down approach to arrange the bottom-up approach used by A-Life. The template in this case is the model that the controlling AI uses to arrange the A-Life underneath it. (This will become clearer when we look at emergent behavior and some A-Life examples in coming chapters.)

Another reason that template-style AI and A-Life are good approaches is because it lets us create whole populations of modeled entities with very little resource use. For example, it enables us to employ shared routines. The individual objects will have states and some limited memory allocated to them; the more complex the AI model, the more resources each entity will need.

It is sometimes better to start with a template, coupled with some shared routines that can be applied to any entity derived from that template, and then store the behavioral offset from the template, rather than try to model each object in the system discretely. This lets us define accurate, basic behavior and then change it slightly to yield variations. Above all, we can then breed those variations. In doing so, each variation can grow and allow us to benefit from newly emerged behavioral patterns.

By creating a template (either one or a collection of the mentioned building blocks) that represents the default behavior, we save resources. We are, after all, just storing the offset from that behavior. On the one hand, this means we have a good starting point from which lifelike behavior can be derived; on the other hand, we have a system that can extend itself with new and emergent behavioral patterns.

It is the template's job to provide some semblance of first-order intelligence, even if it emerges into something new based on the application of A-Life and emerging behavior paradigms. That first-order intelligence represents the default behavior. This is important because chaos is often the result of a truly random population. It is better to deal with that chaos in the design and development stages, rather than in the game session itself.

However, building blocks can also be deployed in a way that observes and records the actions of the player. This means that we can start with a blank canvas and gradually build up a behavioral model, based on those observations. The player then becomes our first-order intelligence, and this starting point can be used to breed variations. Each variation will start off as being good or bad and evolve as the player plays the game. The ensuing reinforcement will serve to temper that behavior and, hopefully, provide a new depth of gameplay in the bargain.

Behavior Management and Modeling

As will be seen when A-Life mechanisms are covered in more detail, AI management must be used for behaviors implemented in the game. This was previously referred to as the "guiding hand" that enables us to keep the worst excesses of a reactive and adaptive system in check.

Behavior Management

Neural networks have built-in limiting factors, and the individual components can be artificially restricted to try and make sure that they do not behave in undesirable fashions. However, when combined them using A-Life or purely emergent techniques, there is often no real way to predict exactly what the outcomes will be.

If the AI routines that are monitoring the system detect that a routine is reaching a critical point, it needs to be able to intervene. That intervention can take many forms—from directly interfering with the behavioral propagation itself to providing new stimuli to change the behavioral pattern, using the object model created as part of the game design.

Behavior Modeling

Another way to use the guiding hand is in directly modeling behavior that we want deployed. The modeling AI decides what is the desired outcome and makes the correct decisions to achieve that outcome. For example, the step-by-step move analysis in chess can comprise a combination of methods: the AI routine that plans the moves, having looked at a representation of recent moves and an extension of the possible move space, using one of the AI building blocks, such as a neural network; and an analysis routine.

Other examples include behavioral pathfinding with backtracking. Here, we can use an expert system to provide a number of paths through a behavioral system and couple it with an AI process capable of negotiating those paths.

We can also apply the same idea to learning through trial and error. The individual building blocks are used to test the boundaries of templated behavior, with a guiding AI routine on hand to decide the difference between good and bad behavioral traits and feed that information back into the system. This also applies to learning by example and routines that can sort a completely random collection of data to yield the desired behavior. This is important when we apply A-Life routines that might return combinations of digital genes that have unwanted behavioral traits.

Our final example illustrates this last point. Given enough resources, we can train a program to solve a maze, using pure trial and error with a genetic string of choices. Part of the solution space will be chaotic, and part will be derived from trial and error. The result will be a behavioral model capable of completely solving one particular maze, but the technique can be used to solve any maze. However, the algorithm may need to visit every point in the maze before arriving at a solution. This is an example of ground-up deployment.

Ground-Up Deployment

In the previous section, we looked at an example of a critical compromise between AI and A-Life paradigms. We often need to use that compromise solution because, as often noted, resources do not usually allow us any other approach.

In a perfect world, we might want to create everything from the bottom up. This is not usually an option. There is just too much going on in the average game, so we create models (as was seen) that approximate the desired behavior. So rather than create an entity that has a purely biochemical background and expend all of our resources on that part of the system, we build a model using an FSM or similar mechanism. The alternative is to model the system right down to the individual chemical reactions.

On the other hand, when trying to create a game like *Creatures*, ignoring these low-grade biochemical building blocks and injecting A-Life at a purely logical level will not yield the same depth of gameplay. In these cases, the models become the game. Some areas where this can be used to great effect are as follows:

- First-order generation
- Environmental modeling
- A-Life simulations

First-order generation means that A-Life is used to create the initial set of entities and models, which are then implemented using a less resource-intensive mechanism. We have covered this before and will revisit it, so only the highlights are presented here.

Environmental modeling uses A-Life and AI techniques to help model the background environment and the framework into which we can drop our in-game entities. A-Life simulations are games that are nothing more than simulations.

First-Order Generation

We have a certain luxury in time when preparing the first-order intelligence (the default behavioral sets) that is not necessarily present in-game. During the design and development stage, ground-up deployment of the building blocks can be used to help create that first-order intelligence.

Providing that we can adequately represent the problem space using low-level building blocks, they can be used to create a system that will allow behavior to emerge. For example, a driving simulation might benefit from a behavioral model that takes into account all the various inputs in terms of the driver's virtual senses. This could be modeled as a collection of neurons in a certain formation, which can then be trained to drive a car around a racetrack. This may sound far fetched, but it is quite amazing just how much of an entity can be modeled in this way.

In fighting games, developers have already looked at using in-game moves and sequences to allow the game to observe and copy (or at least try to predict) the player's moves. Taken in the opposite direction, it is feasible that, through round after round of seemingly random bouts, a first order of fighters could be bred. To do this, we again need to deploy the building blocks appropriately from the ground up. This will probably include FSMs to define the discrete behaviors and some form of neural network to choose appropriate modes from the collection. Even flexible FSMs are possible that can be created, extended, and adapted according to a guiding AI routine. This can also be applied to RPG nonplayer characters, in which we might need to populate a large game with some reasonably intelligent life forms.

If the ability to generate scripts is included, for example, as part of the ground-up approach, we can attempt to breed the first order, which is then used as the default behavior. If we do not have time to pick the best trainers from endless iterations of scripts, we would again need a guiding AI to pick them for us. How to do this will be covered in Chapter 6, "The Power of Emergent Behavior," and Chapter 7, "Testing with Artificial Life."

Environmental Modeling

Environmental modeling is a somewhat experimental area in which the basic building blocks are deployed, and complex, lifelike behavior is derived from them. In

essence, A-Life techniques are applied to produce an environment that reacts as if it were "alive." This can be anything from flocks of birds to flowing water. It can also provide us with the mechanisms to marshal some of the excessive effects of emergence by providing an adaptive environment that can help steer the evolution of behavioral models via additional stimuli. In this way, it can be used to provide inputs for all entities' decision-making processes. The environment in which they exist becomes active—helping them learn from their mistakes and tempering some of the behavior deviation by providing more active feedback, if necessary.

For example, if we really would prefer that an NPC does not try to eat gold, we could vary the negative response that eating gold produces. If the NPC is persistent in eating gold, then we need to recognize this as undesirable behavior and increase the negative feedback in order to stamp out the unwanted trait.

On the other hand, we can apply these techniques to the environment in which the game takes place—terrain, rivers, planetary resource distribution, and so forth—essentially using the building blocks to model and then extrapolate from existing examples. One particularly interesting example is in the modeling of a river. Noting that a river always takes the path of least resistance, we can combine building blocks to create a pathfinding algorithm to seek such a path for a given piece of terrain. Whichever it chooses to be the easiest path will likely be the path that a river would naturally take.

A-Life Simulations

Finally, there are cases in which the A-Life is the game, rather than just a mechanism that is in the background. These games use the basic building blocks more or less as they are; each one will have its place in creating the game itself. The point of the game is to allow life to evolve, and so we are not overly concerned with creating first-order intelligence or tempering the system in any way. Instead, we just want to provide the basic starting point and see where the interaction with the player takes the system.

However, we still have to choose what parts to deploy and where to deploy them. Trying to use an FSM to create associative memory will not work or will require so much implementation and in-game resources that it becomes impractical. The resources will not be available to build beings that are, from the ground up, as complex as real biochemical beings. So we have some hard decisions to make as to where the various building blocks should be deployed for maximum effect, and some shortcuts might be necessary to compensate for limited resources.

In the end, observed models sometimes are as good as allowing truly representative models to emerge from smaller parts. When we can discover where these observed models are more efficient than building a truly evolving system, the deployment of A-Life becomes both achievable and accessible for providing a richer gaming experience.

SUMMARY

This has been a somewhat heavy chapter. There has been a lot of theory, some fairly detailed implementation details, and examples of how the various building blocks might be deployed. The rest of this book deals more or less exclusively with A-Life in video games and how AI can be used to help manage it.

The basic building blocks presented thus far will help you cope with the more rudimentary aspects of A-Life. These include the basic algorithms that are needed to model individual parts of A-Life systems. We have also covered how to simulate some of the A-Life processes, using those building blocks, as well as (briefly) the ways in which behavioral and biochemical processes can be defined in terms of the AI building blocks, and how external AI routines can be brought in to manage the resulting systems.

Now you know how to decide what level of A-Life will most likely be useful in specific parts of the video game project. How this choice affects the outcome of the game was described in some detail, as well as how to make an intelligent decision that hinges on the needs of the game.

However, how to efficiently deploy A-Life and AI has not been covered, and this is key to emergent behavior. In effect, we have been building up to the next chapter, which delves into how emergent behavior can shape the resulting game.

Now that you understand the AI behind A-Life and some of the basic principles, it is time to move on from local applications in single entities and modeled behavior and see what happens when all the various elements are put together. The result is emergence.

CHAPTER

6

THE POWER OF EMERGENT BEHAVIOR

In This Chapter

- Defining Emergent Behavior
- The "Sum of Parts" Emergence Design
- The Individual Dynamic Emergence Design
- The Group Dynamic Emergence Design
- The Reproductive Mechanism in Emergence

One point should be made before we embark on this chapter: Emergent behavior already exists in *the vast majority* of modern video games. Emergence is not a new idea; it has existed in theory since the classical times of Plato and Socrates, and many other great minds have written on the topic.

This book is not intended to provide an entire scientific discourse on emergence. Instead, our discussion will be kept relatively simple. Emergence occurs whenever a set of entities is required to follow a similar set of rules, part of which involves interaction with each other. The result is that the observed behavior of the system (a group of rule-following entities) is more sophisticated than the individual elements and the rules they follow. In the real world, ants are a good example of this. Their behavior is governed by instinct, and a single ant will not appear to be particularly sophisticated.

Put the ants together, however, and they are capable of working to keep an entire population fed and housed, while transporting and protecting the queen and her eggs. As a collection of individuals, their behavior is far more sophisticated than each individual by itself.

However, we are not concerned with transient emergence (which comes and goes as entities come together and then go their separate ways). Rather, our concern here is with the kind of emergent behavior that provides a lasting and useful mechanism within a video game—the kind of emergence that permanently changes the environment, rules, entities, and interactions that might even extend beyond a single game session. We have to start somewhere, and some simple emergence can be quite powerful if the effect is appropriately processed by AI and A-Life systems.

The purpose of this chapter is to demonstrate how AI techniques can be put together to generate *useful* emergent behavior. The beauty of planned emergent behavior is that it can be a very efficient way to create artificial life—that is, entities and environments that react in a lifelike way—and this lifelike behavior exhibits properties that would not ordinarily be achievable using pure rule-based systems.

A distinction is made here between unplanned emergence and planned emergence; after all, surprising emergence can happen in situations where there is absolutely no AI at work and there is a solitary rule or set of rules (Conway's Life, for example). What we would like is planned emergence with a useful outcome that can be trusted.

Conway's Life is a good example of emergent behavior and one that we shall look at later on. For now, all the reader needs to be aware of is that it is a system in which each cell has rules that govern whether it lives or dies, but the resulting behavior of the system is far more complex than the simple rules would suggest.

Wouldn't it be better to stick with regular, dependable expert systems, or weighted rule-based systems coupled with neural networks? There are circumstances, however, when even the combination of rule-based systems (and other AI paradigms discussed Chapter 5, "Building Blocks") produces unexpectedly sophisticated and unpredictable behavior. Here is where the power lies and where the trouble starts.

If the behavior is unexpected, then will it be impossible to control? Ordinarily, yes; however, there are ways to mitigate some of the excesses of a runaway emergent system so that the behavior is kept to within certain limits. The key will be the application of game rules.

However, emergence is not only about the individual entity. Group emergence is also a powerful paradigm in video game design. Here the reliance is on flocking, mob culture, and other pseudo-social mechanisms where the behavior of the group emerges over time in a way that such individuals exhibit behavior that is different *because* they are part of the group dynamics. Then there is "evolved emergence"—behavior that emerges over time due to an unpredictable recombination of genetic material. Evolution and genetic algorithms/programming, coupled with virtual propagation of derived populations, are common themes in A-Life, but since they can be tied in with emergent behavior, this seemed a good place to bring it all together.

Chapter 4, "The A-Life Programming Paradigm," covered (at a high level) some of the driving principles, but here we dig deeper into the supporting algorithms and their applications in video games. Emergence via evolution is not a novel idea. Some designers argue that the purpose of digital genetics is to take advantage of this emergence.

Evolution, in this case, is the result of a set of simple rules being applied in a system, which leads to behavior being created that is more complex than that rule set. While evolution deals with the individual and emergence with the system, there is a strong link in the abstraction of evolution as a system that can be manipulated to give rise to emergent behavior and produce individuals who could not otherwise have been predicted, despite the absence of random genetic mutations.

Of course, add those mutations, and the system becomes even less predictable.

However, it is also possible to have evolution without emergence, since emergence is defined as a change in behavior. And while evolution might *hone* behavioral patterns to make them more or less successful, the behavioral pattern itself might not change. This is the key difference between the two.

It is also not possible to have emergence without AI, while it is perfectly possible to deploy digital genetics in a system where there is no AI whatsoever. The results might be limited, but they will also be generally predictable, which absolves them from being examples of emergence.

Evolved emergence can also be called "reproductive emergence," but this suggests that it is only achieved through the act of reproduction. Some programmers might maintain that there is an insistence on evolution, as well. Therefore, we have three forms of emergence:

- Individual emergence,
- Group emergence, and
- Reproductive/evolved emergence.

Before looking at these in more detail, the different kinds of emergence that exist in A-Life theory must be defined. We will also discuss what each one means in terms of video game design and development.

DEFINING EMERGENT BEHAVIOR

Prior to deciding how to deploy its power in a project, we must first understand what constitutes emergence. Then we can work out how to arrange for emergence in a controlled environment as well as discover the mechanisms available to help in this effort.

As with other aspects of AI and A-Life, if emergent behavior is not part of the game design from an early stage, it will be almost impossible to integrate later on in the development cycle. Intelligence is not an optional "extra." If the game is designed in such a way that AI/A-Life could be added as an improvement "later on," the effort needed would negate any benefits that it might bring. In short, it is better to plan AI's inclusion from the outset or leave it out entirely.

Note, though, that if the advice in Chapter 5 has been heeded, and the system is based on reasonably flexible rule systems and scripting (even if it is balanced by some digital genetics), emergent behavior might occur unintentionally during the scripting stages of the game's final balancing. This is the equivalent of an MMORPG, where strict rules are in force, but where the inventiveness of the players and their scripted bots can lead to behavior patterns that the developers did not expect. In this case, it is the players who create the emergence by adjusting the code of their bots and then sharing the results.

So emergent behavior, like other aspects of AI and A-Life in video games, is a sum of mechanic parts. That is, the end result is more than just the expected sum of its parts. In the design phase, game and system rules have been put in place that govern what actions can take place, how the interactions work, and what the roles of various entities are. However, when we add these functions together and introduce some variations (either from within or without), the result is more than just the application of rules and the interaction between a set of entities—yet the mechanics that drive it are only according to those rules and interactions.

Take a racing game, for example. From *Wipeout Pure* to *S.C.A.R.*, arcade racing games and any other games that involve competitors moving around an obstacle-based circuit inevitably employ some emergent behavior. The rules are simply stated (see Chapters 2 and 3). The vehicles move around the track and avoid contact with each other and the walls/off-track areas, and crucially, they slow down for bends. What happens when a track designer decides to create a combination of track and obstacle—say, a narrow gap between two buildings, like in *Need for Speed Underground*, followed by a bend? Following the rules, each entity (the player included) will slow for the gap and remain slow for the bend, before accelerating out the other side. When several vehicles arrive at the gap within a discrete time frame, they will be forced to either be reckless (and perhaps crash) or line up single file. The resulting traffic queue was not planned and is an example of *temporary* emergence; that is, the effect does not last, and the behavior of the entities involved does not necessarily change as a result of the temporary queuing behavior. It can be argued that this is emergence, but not emergent *behavior*, per se.

As an individual dynamic, learning by example is an example of emergent behavior. In isolation, an entity's behavior can be altered by the way that it interacts

with its environment. For video games, this usually means that the behavior emerges as a result of something that the player has done.

In the previous racing-game example, a human player will over time learn to adapt his behavior to the unique conditions at the narrow gap part of the track. The emergence of queues will teach the player to change his behavior in order to find the best way of maneuvering the gap and bottleneck of cars. In order for this to be different than adaptive behavior, the pattern that results must be more than just a slight adjustment of the algorithm. It must be an entirely new behavioral pattern. In the case of our example, the new pattern might even include an internal representation of the course, providing simulated learning.

The result, however, might lead to further emergent behavior; the effect that the player has on the other vehicles might also be factored into their behavior patterns. If the other vehicles are imbued with the ability to change their own behaviors (via the building blocks mentioned in Chapter 5), we might arguably force even more emergence to take place. In fact, even though we used the racing example to support individual emergent behavior (queue avoidance), this is also an example of emergent behavior as a group dynamic—that is, the interactions between individuals. Most transient emergence of this kind is a result of group dynamics and is the easiest kind of emergent behavior to explain.

It might be obvious that the confluence of track design and rule-based behavioral models will lead to some emergent behavior when a collection of racing entities is placed on the track. What is less obvious is that this interaction can be used to spur changes in individual behaviors, which result in emergent behavior of a permanent nature on an individual basis.

These will then feed into new applications of the rules and result in yet more emergent behavior as a group dynamic, which will then cause permanent behavioral changes and lead to emergent behavior on an individual level, and which can be referred to as "behavioral feedback."

This brings us to emergent behavior as a reproductive mechanism, and evolution is an example. Whether it be positive or negative (for better or worse), evolution is the result of the application of different rules, which inevitably leads to behavior that is richer than the rules themselves might create. If we assume for a moment that a video game is designed that needs to take advantage of the gradual ramping up of intelligent behavior, this can be achieved through evolution—reproduction from within a selectively successful population. Only those solutions (entities) that are gauged as successful are selected.

Over time, behavioral patterns will emerge as a result of the application of digital genetics that satisfy our criteria for success (or failure if we choose to "dumb down" the population), but the exact way that this occurs will be emergent. Put another way, we have determined the nature of the behavior that we want, but the actual behavioral patterns are left largely to processes outside our control. We might not even know, upon observation, which elements have caused the behavior—which brings us to a discussion of the different kinds of emergence that might prove advantageous.

Types of Emergent Behavior

As has been suggested throughout this chapter, not all emergent behavior algorithms are created equal. In fact, it should be clear at this point that several of our examples (based on racing games) can be broken down into two categories: emergence that is predictable and emergence that is unpredictable. However, these two perspectives have differences that run deeper than simply being a question of behavioral feedback. Their names are in direct opposition:

- Weak emergence, and
- Strong emergence.

Most of the emergence that occurs in video games falls into the "weak" category. Weak emergence results from the interaction between game rules, the game environment, and the entities within it, which gives rise to new behavioral patterns that stem from the application of a flexible (adaptive) rule-based system. More important, we can derive the effect from the cause. The behavior can be examined and its cause discerned just by looking at the elements of the system that have come together to produce it.

A traffic jam, for example, is an example of emergent behavior. The cars have not been programmed to line up behind each other, but they do so nonetheless. If we look at the state of the road and the number of cars and know that they will try to avoid hitting each other or driving off the road, we can conclude that the narrow path in the road has contributed to the traffic jam, as has the number of cars and the rules that govern them.

Strong emergence is defined by emergent behavior that results from the same interactions, but in which the cause cannot be discerned. That is, the system's effects cannot be traced to the individual components that must surely be causing them; nor can we predict the behavior or artifacts that might stem from the combination of these rules and the environment. Therefore, the behavior is more than the sum of its parts.

And, these rules need not be especially complex. "Conway's Game of Life," for example, is based around four very simple rules:

1. A live cell with fewer than two live neighbors dies.
2. A live cell with more than three live neighbors dies.
3. A live cell with two or three live neighbors lives.
4. A dead cell with three live neighbors comes to life.

These are applied in all eight directions at a given moment in time, and the new set of cells is calculated. We can work back from specimens such as the glider and derive what has caused them, but we could not predict what might happen if we changed one or more of the rules.

A glider was a discovery in the Life universe that proved Conway's point: that the rules would give rise to things that would emerge from the complex interactions between the cells but, crucially, *could not be predicted*. That is, knowledge of the

initial state of the system and the rules in place to govern transitions in that system is not enough for us to be able to predict which will happen in that system. The simulation has to be done for us to find out what will emerge.

The glider was a configuration of cells that was capable of moving. In other words, the glider would rearrange its configuration through several intermediate configurations and arrive back at the initial configuration but shift one unit along horizontally and vertically.

Furthermore, glider guns were discovered, which were capable of firing gliders through the Life universe, creating them out of nothing except the strict adherence to the rules quoted earlier. Astonishingly, logic gates could also be represented by streams of gliders colliding at a specific point with other specific arrangements of cells, producing a glider (or not), depending on the laws of the universe.

Thus, despite being able to derive cause from effect, Conway's Life might belong in the strong emergence category based on the simple fact that the behavior of the system cannot be predicted without running the simulation. The stock market is an often-quoted example of a system that exhibits strong emergence. We cannot predict what will happen with any degree of certainty, nor can we know the exact causes of specific stock behavior.

Which brings us to another interesting way to look at emergence. If the cause of emergent behavior can be discovered, then we can probably make it occur again; it has become predictable. In a weak emergent system, this would be possible. On the other hand, a strong emergent system precludes this possibility. Even if we knew the cause of the effect, we cannot be sure that these causes will have the same effects if the circumstances are repeated.

These are the two perspectives of emergence distilled into cause and effect. They are not exclusive; that is, there is substantial interplay between the two, and since behavioral modeling is based on layers of interaction, the two phenomena can coexist in any system, including video games.

It might be argued that the player is the ultimate example of a strong emergent entity within the system. It is nearly impossible to predict how he will react, and it is very difficult to make the player react in a desired way. There is also some disagreement over the definitions of strong versus weak emergence. Here, we will mainly be dealing with weak emergence.

Even weak emergent behavior sometimes looks like magic. For example, Steve Grand [GRAND01] points out that:

> *"Nobody who was given the rules for Conway's Game of Life would immediately infer the existence of the glider. But is that perhaps because we are just not clever enough to see it?"*

If, as we saw previously, the race track example of a traffic jam is an example of weak emergence because we can tell exactly what has cause it from the elements that we fed into the system, then what would constitute strong emergence? That is, what kind of behavior (within this example) can possibly be so complex that its cause cannot be determined?

The answer is simple. If we apply an algorithm for adjusting the behavior of the cars such that there is a lasting behavioral effect, then the combination of experiences can lead to strong emergence. When looking at the behavior of a single vehicle (that is, its driver), we might not be able to determine what has caused a specific pattern from the input elements. The question is, however, do we actually want strong emergence at all? A system that is unpredictable and in which causes cannot be determined from effects sounds like it is too chaotic to be useful.

Is it not better to restrict ourselves to weak emergence so that there is a fighting chance of being able to apply some form of control? After all, the only thing that we want to gain by using emergence in the first place is the ability to exploit existing game rules and put them together in a low-impact way in order to allow behavioral patterns to emerge, which will offer value-added challenge to the player.

This is an alternative to actually prescribing all the behaviors in such a way that the player is inherently challenged by the game environment. The flexibility of emergence allows us to steer the entities within the game to better reflect the style, capability, and personality of the player(s).

With this as our goal, surely weak emergence is enough to leverage AI techniques such as learning, finite state machines, feedback systems, neural networks, and so on, in conjunction with programming techniques such as dynamic script rewriting and A-Life techniques like genetic algorithms and genetic programming. The arguments for weak emergence are simple enough: We can test and debug the algorithms better, since the causes of "bad" behavior can be determined, and we can take steps to control the *elements* that cause it (rather than the behavior itself). Emergence can spiral out of control if we are not careful—especially if behavioral feedback has been used to augment the adaptability curve. (Note: The *adaptability curve* has been used here to describe the rate at which a given system tends to change its behavioral patterns to arrive at a desired (good or bad) behavior.)

There is also a case against limiting ourselves to weak emergence. By designing a system in which causes and effects can always be known, a layer of complexity has been introduced that needs to be thoroughly tested. It might be easier to limit the excesses of strong emergence, rather than attempt to make certain that strong emergence cannot actually take place.

Herein lies the answer—or at least *an* answer. A better goal would be to provide a mix of emergences—some weak and some strong. As noted at the beginning of this section, the two can coexist in the same system. They are not mutually exclusive and can even be combined. Our smart racecar driver is an example: Every little piece of group behavior observed that results from the application of driving rules to the game environment (track, other drivers, and so on) is an example of weak emergence. If the effect causes changes to the player's behavior, then strong emergence will take place.

If we take this a step further and "breed" new drivers from the best (or worst) contenders by using GA or GP techniques, then the system will tend even further toward strong emergence of a kind that can be controlled by the same applications of GA and GP, selectively. Our "gene pool" of behaviors is the mitigating control

factor. This brings us to how systems can be designed to take advantage of emergent behavior in an active way, rather than allowing it to take place without constructive application.

Emergent Behavior Design

As will now be obvious, all emergence starts with the application of a series of small rules or *elements*. Equally obvious is that emergent behavior is a feature of any video game in which such rules force an interaction with the environment. Arguably, there is no sign of emergence in *Space Invaders*, since there is no direct interaction between the environment and the entities.

On the other hand, all games that are based on entity-environment interactions will display varying levels of emergent behavior. For example, the monsters in *Half-Life* follow rules that, when put together, force patterns of behavior to emerge that would not be possible on their own or if confined to a certain level design.

In certain circumstances, the emergent behavior (behavior that has not been explicitly written into the code) can be predicted and channeled. We know that certain rules must be followed and what the outcome ought to be. Therefore, given suitable actions by the player, the behavior is predictable. It follows that the player can also use this emergent behavior to solve in-game problems—from forcing cooperative entities into situations that make them act as cannon fodder, a distraction, or handy allies in a fight, to persuading the enemy units to destroy each other.

The small rules that interact to build complex emergent behavior are all part of the building blocks mentioned in Chapter 5—in a sense, tying it to Chapters 4. The A-Life paradigm and building blocks will inevitably lead to *some* emergent behavior, and it is up to us to try and work out how to make the most of it.

Emergent behavior is the key to using A-Life techniques in video games. It would be both impractical and inefficient to try and program a system in a prescribed rather than emergent way to exhibit this kind of behavior. When designing emergence, two distinct areas are considered:

- Emergence over time, and
- Emergence through combinations.

Emergence over time deals with the behavioral emergence of a single entity (or group), where the behavior emerges gradually. As the entities gain experience through interactions, the rules that govern the way they interact with the system can be updated (see Chapter 5), which will lead to different behavioral patterns.

For example, assume that a soldier has a set of movement rules that enables him to move in any one of eight directions at a variety of speeds, as well as duck, stand, and fire. The action scripts must be triggered by something—so we choose an FSM coupled with behavioral scripts, such as Advance, RunAway, and Hide.

A neural network is then applied to respond to input stimuli—seeing an enemy, being shot, changes in terrain, and so forth—which can be trained over time to choose between behavioral pattern scripts and ad hoc actions. Over time, we expect the

behavior of the soldier to become more lifelike. In the beginning he might just duck to the floor (crouch) when being shot at, but over time, the behavior might evolve to include running for cover toward a building. The soldier was never instructed to do this, but the combination of behavioral patterns, scripts, and neural network offers the ability to learn and choose between different courses of pre-programmed and ad hoc actions, which might well lead to emergent behavior of this kind.

By using a combination of strong and weak emergence (with the behavioral pattern FSM, possibly augmented to weight the outcome, providing the weak emergence), emergence that has been designed into the system can be controlled a little. However, once other entities are added, as well as a reactive environment, we transition from emergence over time to emergence via the combination of entities and algorithms. Combination emergence leans strongly towards strong, rather than weak behavioral emergence. That is, the more interactions and algorithms that are added to the mix, the more difficult it becomes to predict the effects or discover their causes.

Take, for example, the previous soldier example; suppose that we now want to implement squad-based interaction. Each entity is following its own set (instance) of rules, and its behavior is being tempered by a neural network. This may be shared, or each may have its own sub-network—or there might be an overriding "squad network" that directs their actions (like a field commander).

However, they still need to be able to act on their own. Otherwise, once the squad is broken up by the player, they will just run around, unable to function. It is the combination of all the various inputs and managing algorithms that leads to strong emergence. This combination, when linked with the passage of time, will make it hard to predict the outcome.

It is tempting at this point to just use adaptive rather than emergent behavior-modification algorithms. Emergence is not the same as adaptive behavior, although the effects might be similar to an outside observer. Adaptive behavior just means that the system can change its reactions for the better (or worse), based on a changing (evolving) situation. These reactions are limited to behavioral patterns that the system is already predisposed to exhibiting. All we are changing is the frequency with which actions are triggered or the extent to which they are carried out.

We can plan for adaptive behavior because the behavior is within the guidelines laid out. It is hoped that the system will balance itself toward a good approach and use only our prescripted behavior, which should keep to a minimum the danger of the system doing something unforeseen.

Emergent behavior, on the other hand, is rarely planned in a detailed sense. In broad terms, the system is designed to be capable of providing emergent behavior, and the result should improve the behavioral models—create effects that are more lifelike, more successful, and so forth; but it is not be possible to state exactly how the improved behavior will manifest itself. This behavior will probably lead to the system adapting to new circumstances, but this is different than providing explicitly adaptive algorithms. On the one hand, the system is allowed to adapt (through emergence); on the other hand, the system is forced to adapt by using algorithms specifically designed to improve or degrade performance.

So in a sense, adaptive behavior just means making transient adjustments along the same lines, whereas emergence is more powerful, permanent, and (generally) more impressive. A system that exhibits emergence cannot go back to a previous behavioral pattern simply by readjusting itself, especially if it is strongly emergent.

There is a parallel in the realms of genetic algorithms and genetic programming—GA provides us with very adaptive algorithms. The data that is fed into the algorithms will be changed to create new behavior. This is adaptability, with the adaptability curve dependent on the rate at which new datasets are bred and the amount of change allowed to creep in (through mutation and recombination).

On the other hand, in GP, where the algorithms are themselves modified, emergent behavior is provided for. Optimal solutions are not specified ahead of time. Instead, we allow the system to create its own solutions through emergence. This requires that the code driving the behavior be modified (which has been part of our informal definition of GP). So the design has to allow for this modification using our building blocks (see Chapter 5) and respecting the A-Life paradigms (see Chapter 4).

When all of this is pulled together, we quickly realize that the design is going to be layered. A mixture of emergent and adaptive behaviors is needed, coupled with the appropriate techniques to manage it at a higher level. For example, adaptive behavior can be used for individuals, emergence used at the group level (a smart daemon controller), and a mixture at the highest level. Or we might allow emergence at the very lowest level but enable it at a higher level by using AI to spot, correct, and adapt behavioral patterns according to some predefined criteria. It is at the design stage that we set the parameters for the emergence.

Layering the different aspects of A-Life therefore centers on emergence with different ways to allow the emergence to manifest itself and different control methods to make sure the system behaves in an appropriate manner. As it stands, this approach is fairly abstract, so we should flesh it out with some specific examples of how emergence can be designed into various types of gaming systems.

THE "SUM OF PARTS" EMERGENCE DESIGN

The "sum of parts" mechanic means that only by building the system and letting it go can we see the emergent effect of the interaction of its parts. The sum of the parts is greater than the parts by themselves.

Steve Grand, author of *Creation: Life and How to Make It*, offers a rare insight into *Creatures*, a system design that deploys nearly all of the techniques described in this book. [GRAND02] *Creatures* is a game that embodies A-Life principles in a fun environment. Other games have since traveled the same road with varying degrees of success. But *Creatures* gives us a good starting point for analyzing how emergence works and how a system can be built from components in which nonprescribed behavior emerges.

Central to our effort is the theory that the system is modeled on actual life. In other words, we create an analogy to an organism's (roughly) distinct stimuli and responses:

- Stimuli (pleasure/pain/vision)
- Movement (motor control)
- Biochemistry (feeding/poisoning/immune system)
- Thinking (reacting/learning/instinct and so on)

The game environment in which these organisms live is full of objects. Each object communicates with the organisms through a messaging protocol that is part of the underlying system design. The actions and reactions are modeled via scripts that are directly linked to sensory or motor control points.

Now, the actual implementation of *Creatures* is far too complex to analyze in detail here (it was, after all, the subject of an entire book), so we are going to concentrate on a less ambitious example. An emergent system will be described that is based on very simple organisms that can eat, move around, and sense their environment—and of course, they can also reproduce and die.

Stimuli

Our organisms need to be able to sense the immediate environment, and this can be implemented in a number of ways. We will assume that the organism is:

- Passive,
- Active, or
- Both passive and active.

If it is passive, then it will be bombarded with constant messages from the environment, and it will need to organize those messages appropriately. Steve Grand discusses this at length—referring to what he calls the "cocktail party effect," which involves filtering these messages by importance.

Our building blocks can be leveraged to help us out here. If we choose to implement a filtering system as suggested by Grand, which is based on a collection of neurons all firing at a certain level with a "head neuron" choosing between them, then a single-layer neural network can be used to achieve this. In fact, it is the first example of emergent behavior by design (albeit weak emergence) that we are going to take advantage of. It turns out that if we feed each neuron's negative output into the input of each other neuron, this will have a mitigating effect on the output of those neurons.

Subsequently, the level at which they fire will diminish and have increasingly less effect on surrounding neurons. This means that over time, one of the neurons will tend to fire more strongly than the others. If the head daemon then samples these outputs at appropriate times, it can decide which is the most important incoming message by simply pinpointing the sensory input that is firing strongest.

This is emergent behavior. We haven't told the daemon to look at the most important sensory input, but we have arranged for it to emerge. We can also temper this arrangement by layering an adaptive system over the emergent one. This can be achieved by adjusting the feedback weights to favor one of the senses (stimuli), even if it is not producing the most attention-grabbing signal. An observer network, for

example, might note that our organism is frequently being attacked, because it pays more attention to the pleasure of feeding (exhibiting gluttony) than it does to approaching danger.

So we might allow the network to adapt itself accordingly and remove the emergent gluttonous behavior in favor of self-preservation. Here is an important note, however: Because this is an emergent system, there is no guarantee that the effect will be the one originally desired. This is because the balance of the system might change inadvertently and cause other behaviors to emerge. For example, the poor organism might spend so much time running away from things and so little time replenishing its energy that it expires prematurely.

This is where the uppermost layer comes in to play—tempering the system so that undesirable behavior does not occur. Of course, digital genetics will also play a part, but we shall come to that aspect in due course.

Our example treats the stimulus from a passive (or reactive) point of view. The active (or proactive) organism seeks out feedback from the stimulus. In other words, it chooses which one to evaluate ahead of time, thus removing the necessity to gauge which one is most important. But this also removes the ability to spur the organism to action without constantly reminding it to pay attention to the game world.

Again, adaptive algorithms can be used to solve specific sets of circumstances. This means that at the lowest level, the organism is purely adaptive. Emergence only comes from the constant rebalancing of the adaptive network as it shares its attention between action and sensing. We need two adaptive networks—one to choose the stimulus that needs attention (as a matter of priority), and a feeding supervisor network that decides when is the right time to interrupt other activities to check for incoming stimuli. This can be augmented with the ability to choose between stimuli. Thereby, a behavioral pattern would emerge based on the interaction of the three basic rules.

The final option is to have emergence that is both passive and active—proactive when there is nothing better to do and reactive when need be. This is the equivalent of setting a very high threshold on the input neurons, which would cause an interrupt to trigger when that level was breached. An adaptive neural network can also be set up to choose an appropriate response between the two, create an FSM or even a network of FSMs to change between active and passive states. This can be associated with other systems to choose the appropriate state based on internal or external triggers.

The relationship between feedback (positive and negative) and pleasure/pain sensors is an interesting area. The simplistic way to look at these is that pleasurable acts should be continued, and painful acts should be stopped.

To design this in a system, the things that cause pleasure are connected to the pleasure sensor, which boosts the inputs of those events that were active at the time the entity experienced the pleasure. Conversely, the inverse of the pain sensor's output can be connected through another collection of sensors to the inputs of those neurons that are active at the time the pain is experienced.

Note that the two sets of sensors (pleasure and pain, and their causes) are not necessarily connected, except through the system as a whole. This means that if the entity is eating, the act of eating in itself might not cause pleasure, but the suffusion of energy through the entity's body as a result of digesting that food might well be a source of pleasure. Likewise, movement might not cause pain, but hitting a wall might. Movement itself is a complex issue, but it can be summarized fairly succinctly at this stage.

Movement

So far, the organism (entity/object) can sense. We haven't been very precise as to exactly what it can sense, but it is assumed that these are external and internal stimuli in equal measures. It needs some way to move toward or away from the things that it senses. There is, however, more to movement than that—especially emergent movement, which can take the form of following a trail, avoiding danger by running away, moving out of sight, or approaching food or a mate.

Of course, we do not want this behavior offered to our entity on a plate. The behavior should emerge by itself, based on rules with their own little spheres of influence. Again, this is weak emergence. If the organism is following a trail, it is known that this behavior is the result of interaction between stimuli and a controlling network. In the same way that stimuli can be set up and read from (along with the "attention" system to choose between them), there are also several movement algorithms to work with:

- Forward
- Turn
- Backward
- Eat
- Idle (do nothing)
- Procreate

Again, we are going to rely on building blocks, and again, the appropriate blocks at the base level are FSMs. The states are likely to include one from the previous list, and the triggers can be any one of many possible stimuli.

Of course, this is reasonably straightforward, but for emergent behavior to take place, what we really need is some way to create artificial triggers that can fire state changes, along with some additional information, such as speed and direction. An adaptive neural network might suffice, provided the inputs and outputs are connected in a fashion that allows the network to be aware of the results of its actions.

This feedback is important, and part of the underlying rules that lead to emergence requires it, since without it, the organism cannot connect the two. If that connection does not exist, then emergence will probably not take place, except in a vaguely random fashion.

For example, if the organism senses food and moves toward it, then the feedback should provide a boost to that movement function. On the other hand, if the organism

hits a wall, then the resulting pain should make it increasingly less likely to repeat the action, with the end result being complete avoidance of the wall in question.

Without the connections between the sensory network (however implemented) and the motor network (providing movement), there will be no emergence. It is the interaction of the small rules that creates the emergent behavior. Also, just connecting them together without any controlling logic will not result in emergence, except in a very loose, reactive manner. Here we've used neural networks, single neurons, and adaptive FSMs.

Our organism design is becoming quite robust. It can move and sense, and it can probably (with the appropriate sensors) determine when to stop doing something and when to continue. However, it has no reason for these acts; walking into a wall brings no consequences beyond the acknowledgement of pain, and eating has no advantages beyond some slight pleasure. What needs to be added is some kind of biochemistry that instills consequences—for example, that our organism lives, gets hungry, depletes energy, and eventually expires if no food is found.

Biochemistry

The biochemistry in a game like *Creatures* is extremely complex. This book is not intended to explore the details of creating a game that is based on digital life forms. Our organism needs only a simple set of rules for triggering a neural network based on:

- Hunger,
- Tiredness,
- Pain, and
- Pleasure.

Hunger can increase and decrease—increase as a result of tiredness/expending of energy and decrease after ingesting food—and may have trigger levels that cause pain or pleasure to be triggered. The emergent behavior could take the form of the organism moving and getting increasingly more hungry, until further movement causes it pain. At this point, thanks to the pain sensor, the active neurons (in reality, movement actuators) at the time are fed a negative signal, which should cause them to wind down over time and stop. Now, the organism is not going to last long in this case because it can no longer move to find food. So the pain must be made to subside (although the hunger does not) so that the organism can continue to be active. This is akin to sleeping.

Now, this scenario is very linear and not terribly emergent. What is needed is a collection of design rules that can yield similar behavior, be linked into the other parts of the system (sensors and movement), and lead to emergent behavior that is unpredictable in its details, yet which fulfills the design goal: The organism must learn to look for food at the first sign of hunger. Hunger, then, becomes a different kind of pain—at least, the emergent behavior that it is coupled with must be different than just cessation of movement.

Until now, we have created a system that revolves around reactive, directly connected subsystems. To take our organism further, it is clear that this kind of digital instinct will not serve us terribly well. We need decision-making capability and the ability to differentiate between inputs; in short, we need an organism that can *think*.

Thinking

The definition of "think" here is rather a loose one. Ants think—at least, their emergent behavior, built up from a crossover of instinct and sensory information, would suggest that they do. Our organism should also be able to think or at least present the illusion that it is doing so. Emergent systems in general should create the same illusion. After all, that is the root of this approach: the ability to allow behavior to emerge that is unplanned and yet manages to succeed at a predetermined goal. For our organism, that goal is survival and possibly procreation.

Different games will have different goals. *Creatures* exhibits more sophistication in this area (as well as in others) than other games are likely to require. For example, a first-person-shooter or similar action game will not usually have enough processing resources to implement a full neural network-based brain in the same way as a game in which A-Life actually is the game.

The design pattern for an artificial brain (in the context of this book) revolves around instinct and emergent reasoning. Here, the term "emergent reasoning" means the appearance of reasoning ability over time, based on simple AI building blocks. Those building blocks have to contain a certain amount of prescribed behavior, otherwise we need to embark on the aforementioned quest for real A-Life, such as in games like *Creatures* and *Black&White*. So what does all of this mean for our poor little organism?

First, the direct connections between stimuli and movement are removed. Instead, the design calls for a "central processing" center where the connections between sensor and motor functions can be controlled, as well as feedback from the biochemical system, based on a collection of simple rules.

Second, we need to define behavioral units (instinctive patterns) that are designed to help fulfill the basic goal of the AI. So if attacking while remaining unscathed is the goal, then the instinctive patterns should reflect this. If survival (as in the example here) is the goal, then the patterns should reflect looking for food, consuming it, and conserving energy.

Obviously, the building blocks of choice for many of these functions will be FuFSMs that provide adaptability within well-defined sets of rules. However, these need to be linked together so that the weight with which they are employed in-game is altered, depending on the circumstances. The managing rules will form the *thinking design*, and the techniques from the previously covered stimuli and movement sections can be pulled together.

Here is the third and final point: Depending on the required complexity of the design, we can choose to augment the instinctive patterns with other bits of AI. Memory can be added, using any of the building blocks that enable a connected

adaptable network. Memory would be helpful for remembering where food tends to be, for example. Otherwise, using purely instinctive patterns, the organism might have to wander around randomly for quite some time before finding food. At least the emergent/adaptive system defined will allow it to locate food if that food moves, but it might expire through lack of energy in the process. On the other hand, expect different behavioral patterns to emerge if the food source is (predominantly) moving itself, again based purely on the instinctive patterns.

Memory would interfere with the balance. It would also give the organism an advantage, but this is something we have to design for, bearing in mind that if every entity in the game needs a rudimentary memory, this is going to cut in on our in-game processing and storage facilities.

Emergence through the rule-based application of adaptive systems is usually going to be more efficient, especially if we have a way to store the offset from normal behavior (thus only storing the behavioral patterns themselves once) for each entity in a memory-reducing way. We will see more tricks like this in Chapters 8 and 10, which cover implementing these designs. Another way to save some of our processing resources might be through the use of emergence through reproduction.

Reproduction

Granted, some strict A-Life (and possibly AI) scientists might take issue with some of the classifications in this book. However, within the confines of video games and the application of AI and A-Life within them, the definitions presented here are deemed justified. Emergence through reproduction simply means that behavioral patterns are expected that have not been explicitly designed to emerge, based on the way that the in-game entities are allowed to reproduce. Reproduction may occur at:

- The data level (genetic programming),
- The rule/algorithm level (genetic algorithms), or
- The entity level (A-Life and digital genetics).

Here, the term "reproduction" loosely means that the system will take two (or possibly more) entities (datasets, algorithms) and combine them in such a way that a child is produced that can be considered a product of the two parents. It might contain some aspects that are the result of mutation, but otherwise, it should contain enough of each of the parents to be representative.

At the data level, this means that we can abstract a behavioral pattern in such a way that the dataset can be run through a fixed algorithm and have a different effect than another dataset run through the same algorithm. Refer back to our organism example: We might choose to supply a dataset to the attention algorithm (that chooses between stimuli, all competing for attention), which dictates its behavior. Assuming that different organisms exhibit different behavioral modes related to the same instinctive pattern based on that dataset, we can design a system that takes the two encodings and produces offspring that reflect a combination of these two parents.

Over time, the emergent behavior should tend toward a single pattern more ideally matched to the environment, since only the survivors were picked to reproduce

from. In addition, the entire system needs to be designed based on the same dataset and algorithm combination, so that a digital DNA strand can be created that represents the entire set of the organism's instinctive patterns and the way it interacts.

From these small rules, we have emergent behavior that can then be used to create more organisms that will, in turn, become a population that exhibits emergent behavior of its own. This is also likely to be strong emergence, too, as it will be nearly impossible to tell which input element has caused the behavior to evolve.

At the rule/algorithm level, the approach is similar, except that the system must be designed to abstract the steps taken in creating the behavioral pattern, rather than use a set of data that defines a specialization of that pattern. (Ways to do this are also covered in Chapters 8 and 10.) At the entity level, a mix of the two can be employed—instinctive behavior on the one hand (the data) and emergent pseudo-reasoning on the other (the algorithms). All of this requires that we model (or design) the system at a very low level—such a low level, in fact, that the end result is likely to be computationally expensive. Nonetheless, the general principles still hold, even if we choose to abstract toward an individual, rather than build it from the ground up, piece by piece. This abstraction is called the "individual dynamic" of emergence, because it takes us a step away from the "sum of parts" into a part of A-Life that can be much easier to realize both at the design level and in implementation.

The Individual Dynamic Emergence Design

Much time has been spent discussing the principles of the "sum of parts" design and how many of the techniques can be reused, so this section will serve as an abstraction of that technique. In fact, the design differences are also abstractions; rather than modeling the biochemistry, we just create a set of rules that represent its effect on the individual. This is, however, a worthwhile addition to the discussion on emergence because it provides a more accessible method for including this kind of emergence in video games and is easier to appreciate and implement, which also makes for easier design decisions.

The first assumption is that, when designing the individual, we are only concerned with higher-order control. In other words, only the reasoning systems of the entity will be modeled. These reasoning systems will choose between more primitive behavioral patterns that are prescribed or possibly adaptive. This adds behavioral variations to a collection of entities in a way that means it is not necessary to model every possible aspect of their behaviors. This is commonly called for when a large number of entities is anticipated, and it would be impractical to extend the behavioral modeling down to individual elements that make up each "character."

We are aiming for emergence that is equivalent to learning by example, but without the need for complex and resource-intensive mechanisms. Of course, much of what we need has already been discussed under different guises. The design pattern will use neural networks in conjunction with FSMs to produce emergent behavior. This is not a new concept—only another way to manipulate the techniques to achieve the appropriate result.

The FSM Network

The first step in the design process is to isolate the behavioral sequences that need to be modeled in terms of the number of states and transitions (with their triggers) between those states. These can be discrete action scripts—such as duck, run, fire, reload, or eat—or they can be more detailed scripts that define behavior—such as hide, hunt, patrol, and so forth. The next step is to take the collection of states and transitions and link them to triggers that are based on things that the entity can perceive. These could be location based (if the player is nearby, then …), or they could be based on the internal state of the entity (if shot and wounded, then …).

At this point, we can create finite state machines or fuzzy FSMs that represent the various ways in which this behavior can be leveraged. For example, crisp state machines leave no room for further customization, while fuzzy state machines have the ability to be (at the very least) adaptable. (For more detail, see Chapter 5.) We need to make sure that the triggers are connected to system events that make sense. Unlike the sum-of-parts mechanic, there is no sensor apparatus (unless it's mixed in), so the information passed to the entity, which keeps it informed, comes from the system itself.

Now that we have designed the possible state changes and determined what can cause them, a network of FSMs can be created, all linked together, which will determine behavioral patterns and sequences of our entity. If the expert system-style decision-chain building blocks are used, as described in Chapter 5, then we can only hope that the adaptable FuFSMs used will lead to variations in behavioral patterns.

However, we want more than that. So neurons are used at the key decision points, and their inputs and outputs are arranged so that FSMs are conceptually linked together. The idea is that the neurons act as ways to keep track of the state changes and influence these state changes, based on the overall activity of the network. Essentially, we have built a neural network with FSMs at the end points, rather than neurons. Compared to the previously used layered models, the FSMs take the place of the outer layers of neurons, with a layer of neurons in between.

As shown in Figure 6.1, on the left are the triggers, which enter the neural network as inputs to the neurons, and on the right are the FSMs. The FSMs would then be fed by the outputs of the neurons, as well as fed back to the inputs of those same neurons—positive feedback to reinforce behavior and negative feedback to cancel it.

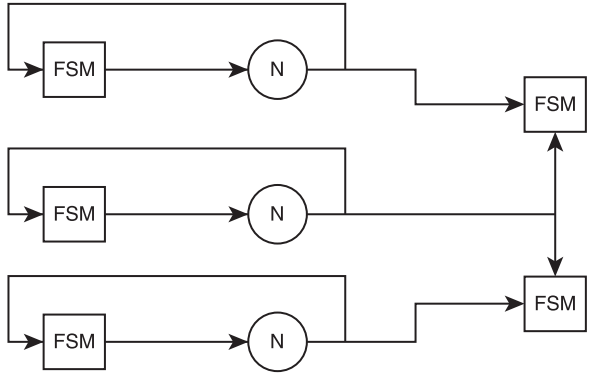

FIGURE 6.1 The FSM network.

Since the FSMs are also free to change states based on external inputs, the network is more active than a simple neural network that is based on input/output processing. Given in-game environmental situations, however, the network can be trained in much the same way. This means that the entity can be put in new situations, with good behavior ensured. The FSMs make sure that the behavior is within certain bounds, and the neural network behind it provides a way to allow all the necessary inputs, sensors, triggers, and actions to be balanced out so that the required behavioral patterns will emerge. But in order for the entity to function correctly, the network will have to be trained.

The Neural Network

Recall that neural networks operate on the principle of patterns. In Figure 6.1, it is clear that the input patterns consist of state change information and sensor information. These pieces of information are transient; they are always changing.

Therefore, we need to be sure that during the training process, the desired behavior is being correctly reinforced, and negative feedback is provided for the behavior that we do not want. This tendency toward good behavior should also be emergent; we do not want to have to actually prescribe any of it. To do this, some concept of pleasure and pain (or at least reward and punishment, respectively) must be provided, otherwise training will not take place. One way to do this is to walk through the situation and provide the appropriate triggers for the FSM to "encourage" the behavior to take place.

Having trained the system on behavioral patterns (sequences) A and B, we can be sure that, given evolving situation A, the network will respond with an appropriate path. Any deviancies from the path might lead to slight variations, and this is where the benefit of emergence becomes apparent.

It is weak emergence, because a logical connection can be made between the input and output, but the result is emergence none the less. We are enabling the system to live and giving it the occasional pat on the head for good behavior, while

its own folly will likely lead to pain, which should keep it in check. Of course, other representations will occur—such as encoding situations and acceptable responses as tableaus of data (the sequential becoming a 2D representation, rather like a training pattern seen in Chapter 5). In the end, it will be a design choice as to which representation to use. The combination of a neural network and these other systems will produce the emergent behavior, so long as every small part is contributing to the network as a whole.

The Individual as a System

Much of this discussion has centered on entities, but some games need a system model (such as in *Civilization*) that makes decisions that have direct effects on the individuals within the game. Here, emergence is at the topmost level—the guiding AI for the whole game. The game itself adapts to the player, and the system is one big collection of rules, each of which has an effect on the guiding network.

Individual A-Life routines will only be there to offer some "color"—for example, modeling armies as they move around, dictated by the general in command. This does not make it in any way a less worthy application of A-Life, just that it is less deep than creating autonomous creatures.

Other games, such as soccer-management simulations, have similar requirements; the players may seem autonomous when they make decisions or play in a match, but it would be difficult to give them too much freedom. It is usually better to farm out much of the actual lifelike gameplay to a higher power—the system AI. However, the same approach is needed as in entity modeling. There will be state machines, there will be neural networks that can be taught, and there will be adaptive algorithms with which we attempt to enable certain behavioral patterns to emerge. What changes is the abstraction or mapping of the system and its representation onto the building blocks. With each small mapping, we add to the possibility of emergence, so long as the links are maintained (preferably via neurons), which continually keeps the system in a state of near-flux.

The Group Dynamic Emergence Design

Having looked at two ways in which individuals can be modeled to exhibit emergent behavior and therefore add realism, challenge, or life to the video game, we need to see what happens when a collection of these entities is pulled together. This will be called the "group dynamic."

There are two kinds of group dynamics—that which happens naturally simply by virtue of being thrown together by events and/or by the environment, and that which we engineer. In other words, we can let them interact with each other, or they can be given rules that govern their interaction. Rule-based dynamics is common in gaming algorithms, for example in flocking and squad play.

Our strategy is a balance between shortcuts and emergent mechanisms. If we give a collection of birds certain behavioral pattern models that dictate the way they

react to their peers and environment, then flocking might emerge. (Chapter 8 has details on how to do this.) On the other hand, a slight shortcut would be to create a set of rules that tell the birds how they should be moving and use a general algorithm that allows the flocking to emerge. The difference between the two approaches is that, on the one hand, something untoward might cause the flocking to break down, or we might have individuals that exhibit incorrect behavior; and on the other hand, individual choice is precluded by controlling the entire flock.

So the group dynamic relies on interactions between individuals within the game environment. This is yet another step back in abstraction; first we looked at each entity as a sum of parts, then we looked at individuals as units with modeled behavioral modes—in this case by state machines.

Now, let us consider each entity as it exists in one of a collection of states, but with the possibility to control the extent or characteristics of that state. A typical group dynamic such as flocking is easier to work with. In a flock, an individual bird can be in one of several states:

- Roosting
- Level flight
- Rising
- Falling
- Turning

Depending on how much precision is needed, there might be more or fewer of these states. Each entity, however, needs to have more information than just "the bird is turning." We need to know the extent of the turn, in which direction, and so forth.

At a certain point, we will probably want to dispense with the notion of states altogether and just concentrate on modeling the flight path directly. This enables us to substantially reduce the number of states, as long as the parameters that they represent can be controlled.

Then, the interactions need to be designed—that is, those things that through communication or observation cause the state changes. Each of these interactions has to be modeled as a flexible rule that causes a change in the internal representation of the current state. At a certain point, a threshold inside the state will be reached, and a state change might occur—from flying to not flying (or falling), as a bird realizes that there is a wall in front of it and changes state to avoid hitting the wall. The realization is the embodiment of a rule that is based on the proximity of the wall.

Now, if similar rules are in place that determine the bird's actions with respect to its peers, it might only take one or two observed crashes for the bird to realize that the wall can be avoided if it pays attention. The chances are that the bird behind ours has already turned away, having been given some advance warning.

The emerging behavior of the group is what keeps the flock together. It is also just another collection of intercommunicating rules that can be designed and implemented in a variety of ways—from state machines (probably not ideal), to algorithms connected to adaptive rules. A single entity might display certain behavioral qualities that are either chaotic or entirely predictable, depending on the algorithms used.

Putting one or more of the same kind of entity together, coupled with some rules to manage their interactions, can change that behavior and make it more chaotic (within certain limits) or more ordered. This change in behavior is emergent and is caused by the group dynamic. Human crowds exhibit similar behavior. A crowd acts differently than individuals might on their own; the behavior emerges from the interaction between the individuals. This interaction happens in one of two ways:

- Communication, or
- Observation.

Communication can be direct or indirect. Direct communication involves one entity sending a message to another, such as status information or a direct command. Indirect communication takes place through other senses—like the reading of body language that ripples through a crowd as it turns aggressive.

This is tightly coupled to the observation part of the interaction; only, in this case, we treat observation as an active rather than a passive sense. The entity is actively observing the surrounding peer space in order to take its behavioral cue from them.

For example, in our video game, we might model a squad of soldiers that are moving through a village, looking for the enemy (which could be the player, or the squad could be cooperating with the player). In order to enable the modification of their behaviors according to the situation, they might need to communicate, especially if there is a "leader," which might be the player or an NPC.

Each soldier will also be actively observing the rest of the squad, and a set of rules will allow the squad's behavior to emerge naturally as they encounter situations during their patrol. Even if they have no knowledge of each other's characters before gameplay begins, an emergence algorithm can be used to enable them to adapt their own behaviors so that the squad benefits.

A simpler example is to examine what might happen if a collection of our organisms is put together in a confined space. Without adding any additional rules to what we already have, what kind of behavior will take place? We can, to a certain extent, predict the group behavior that will emerge. First, since the entities are just searching around for something to eat, they will probably avoid contact. After all, it doesn't benefit them to move toward something that cannot be consumed.

Second (unless they've been instructed that dead organisms are edible), they will avoid places where expired organisms lie; by the same token, they are not food. So we end up with a collection of organisms, all moving around seemingly at random—much as we would expect.

Now, if we add some rules that deal explicitly with how they interact with each other, the group dynamic will produce emergent behavior. For example, if the organisms have rudimentary memory and find that the corpses of expired organisms make for good eating, they might put two and two together and hang around in groups, waiting for each other to expire.

Make the rules cover poison, however, and different behavioral patterns will emerge. If the reasoning systems of the organisms are made sufficiently open, the rules for interaction could even write themselves. After all, if an organism eats

something that looks like food, and it turns out to be otherwise, we can train it not to repeat this bad experience, using a neural network.

So the group dynamic occurs at two levels. The first is the behavior of the group as a whole, based on rules that dictate the way that the entities should act with respect to their peers. The second is the way in which the individual's behavior changes as a result of its exposure to the group. Both of these mechanisms are very useful.

To implement them, however, we also need to deploy ways for the entities to communicate with and observe each other. This can be accomplished by using messages that cause state changes, or any of the observation or information-gathering techniques from the sum-of-parts discussion at the beginning of this chapter.

The Reproductive Mechanism in Emergence

The final mechanism that we shall look at is the emergence of behavior through reproduction—the creation of new behavioral patterns from existing ones through genetic recombination or mutation. This technique can be applied to all kinds of emergence. It can be used to change rules related to the individual and its interactions with the environment.

Over time, successive generations will (hopefully) evolve and improve their behavioral traits, making themselves more successful. Given our preceding discussions, this ought not to be terribly difficult to arrange, and it isn't.

What we are about to describe is a kind of "digital Petri-dish" in which we can "breed" A-Life constructs to a given level of sophistication during the design and development cycle and then import them into the game, where they will continue to extend themselves as long as there is a nonvolatile storage facility. While reproductive emergence can have many applications in-game, it is potentially most useful outside of the game session—and for practical reasons; using it inside the game consumes a lot of resources in cross-checking behavior (as we shall see at the end of this chapter). Used outside the game session—either in design and development or as preparation for the game session—there is no concern over unwanted side effects. On the one hand, during design and development, we can cut out any unwanted emergent behavioral strands; on the other hand, in preparation for a play session, there will be more resources available for the kind of checking needed to ensure that the resulting algorithms will not embody rogue behavior.

This technique can be used to breed full organisms or just algorithms that are used to control behavior. For example, if we need a fleet of spacecraft, each one can be modeled separately (using resource-saving techniques such as storing only the behavioral offset of each ship), or we can create and apply an algorithm to each one, with slightly different parameters.

To breed new behaviors for full organisms, we need to combine the individual organisms (or their behavioral offsets), but to breed only the algorithms, the governing algorithms can be recombined to produce new behaviors for the whole fleet. Bear in mind that some of these rules will likely govern the interaction between units (see our previous discussion on "The Group Dynamic Emergence Design").

Reproductive Models

Emergent behavior can consume a lot of resources due to the combination of reasonably sophisticated, individual parts. Modeled from the ground up—with each part of the virtual organism being modeled piece by piece—the game can become so complex that only a very limited number of in-game entities would be possible. We can cut corners (as seen in the past two chapters) by modeling state changes and so forth at a different level. Game design almost never calls for modeling A-Life entities from the ground up because it would be needlessly complex.

For example, rather than actually modeling the individual synapses that are connected to pain or pleasure and forcing messages through a system of simulated nerve endings, an FSM can be created that mimics the behavior. While a specific behavioral trait will also require many synapses all working together, we generally just abstract the resulting behavior into a single FSM.

Neural networks are a similar abstraction. They might be based on what we know about animal brain activity, but the resulting models necessarily have to achieve more with less. This is done by abstracting to a level where some of the intricacy of a biological system is lost but is replaced by information-processing techniques available to a digital system.

This means that in the end, we can save quite a lot of resources. The entire system does not need to be modeled in order to represent the desired behavior; but in doing so, we run the risk of losing some of the flexibility and "flair" when the living system is abstracted to a finite state machine.

However, to arrive at the FSM, we can feasibly create an offline system and observe the state changes within. In other words, the entire system *could* be built from the ground up and the behavior allowed to emerge as the rules that we encode interact with each other. It would have to be offline (not part of the game), because far too many resources might be required for it to be a viable part of the game.

By observing the emergent behavior, a set of rules can be abstracted that describe the behavioral state changes, which would parallel those actually taking place. Of course, the resulting FSM will suffer from the same drawback as every other FSM that we have described this far—it is not flexible enough to force new behavior to emerge. It can only deal with the states and changes that were present in the offline environment. If something happens in the game session that is not present in the offline environment, then the FSM will not be able to cope. This is, therefore, a simple and not particularly realistic scenario. FSMs can possibly be created for winning strategies, but there are better ways to create the FSM (or network of FSMs) based on simple deduction, rather than modeling the system.

The basic principle remains the same: During the design and development phase of video game creation, we can derive algorithms and strategies by letting the game rules play out with a population of entities that are governed by rule-based AI. The driving AI could be based on FSMs, networked FSMs, FuFSMs, or any of the building blocks from Chapter 5.

Using emergent behavior as a reproductive mechanism, we can breed state tables and so on for use in play sessions. Essentially, algorithms will be played off

against each other, and we will keep those facets of behavior that seem to advance the causes of specific algorithm-relevant entities.

To do this, it is useful to restrict the virtual play sessions to one aspect of a particular entity's AI at a time. The principle is, if we have too many variables, then the strong emergence that occurs will make it difficult (if not impossible) to discern which change in which variable (behavioral model) caused the behavior to increase in effectiveness. These behavioral models can be created using GA or GP principles—being either specializations of a single algorithm (GA) or changing algorithms with the same basic set of commands (GP). For example, soldiers in the army are taught how to scan the landscape for trouble.

It is a simple principle. Near, middle, and far distances are scanned, working from left to right across the field of vision. This sequence helps soldiers ensure that they are covering as much of the available territory as possible. To generate a more successful variant using GA, we can define the basic variations on the algorithm in terms of scan arcs and attention distances, resulting in the three variations shown in Figure 6.2.

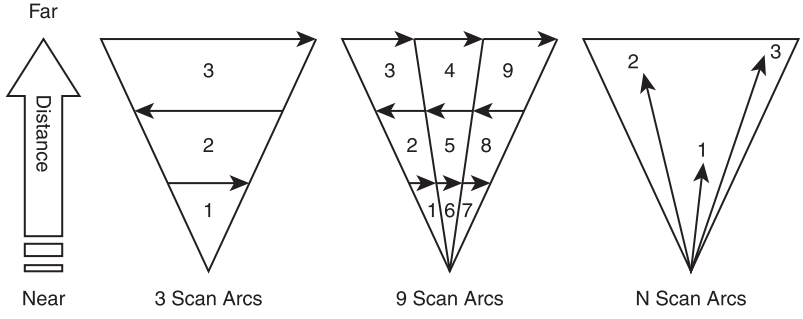

FIGURE 6.2 Scan arc variations.

The first is a basic three-arc scan from front to back. The second variation divides the scan area into three separate cones and then analyzes each one in turn, for a total of nine scan arcs. The third variation just analyzes very small scan arcs at random—the equivalent of eyes darting over a landscape, with no real plan or strategy.

(It is called 'N' scan arcs for this reason—there could be 1–N scans performed, where N is as arbitrarily chosen as the direction in which the scan is performed.)

Our basic algorithm remains the same: scan from left to right. The data that it operates on may tell it to scan from *x* degrees to *y* degrees, near, middle, or far. This is the scan arc, which can vary between 1 degree to the whole (in this case) 45 degrees that make up the available field of vision.

The GA approach might put two entities together, one facing the other (at random positions), and test to see which one consistently spotted the other before being spotted itself. By consistently breeding together the datasets that embody the scanning behavior of the winner each time, the best approach might be achieved. However, it will still be just a variation on the chosen algorithm. The GP approach would

redefine the process in terms of the underlying algorithm. So if the basic operations were to choose a bearing (in degrees) and a distance (near, middle, or far) and then look along it, these could be put together in a variety of ways that would result in many different algorithms for (hopefully) achieving the same thing.

By using the same emergence principles, the best algorithms can be combined in order to search for one that is consistently able to beat the others. If we are feeling particularly experimental, we can select this as the basic algorithm for a GA search of the best parameters to apply to it.

The same sequence could then be applied for every aspect of the AI required to run the game. The result would be a good algorithm and/or dataset for each aspect of the game AI, which would also likely be a variation on the finite state machine (strict or fuzzy). These can then be coupled together using decision networks or possibly neural networks, and the weight of all the FSMs working together may well lead to emergent behavior. In fact, it is almost guaranteed. In order to get the best effect, we also need to follow up and find the best ways in which the AI components can be combined to make a working system. The best scanning algorithm can be combined with a good movement model and in turn coupled with one of the various weapon-selection algorithms, and we might find that this works better than trying to use the best of each available. In theory, we would like to try every conceivable combination of approaches and let the system itself choose the most successful. The important point to note is that we have several ways to do this, bearing in mind that these are all done during development of the AI systems and are alternatives to handcrafting the AI manually.

One way is to just build the model and let it run through simulation after simulation, breeding out the most successful algorithms from the endless possibilities. This is quite time consuming and does not always result in the kind of evolution hoped for. The other way is to try and actively breed new FSMs by reorganizing the system of simulated individual components until we get the behavior that we want. In other words, we start with basic behavioral units and then connect them together logically using FSMs (or FuFSMs) to manage the transition between different behavioral modes. It is the coupling of the systems that must be correct, not the whole behavioral system. This approach is equivalent to accepting that certain algorithms are correct (or the best approach) at the low level and then managing the connections between them. In this way, behavior will emerge through the generations, rather than as behavioral traits in an adaptive system that allows for emergence to take place—that is, concrete recombination as opposed to the application of flexible rules.

To do this we do need to be able to break down the behavior algorithms into units that can be "bred from." Some recommendations need to be made as to how the representations of the behavior (using GA or GP) can be genetically recombined when the reproduction takes place. It is useless to just take the list of commands and then randomly combine them together to create new, possibly nonsensical sequences. We might stumble onto a good combination, only to then break it by recombining it in a way that cuts a particular algorithm in two.

Breeding these behavior algorithms might be easier said than done. Examples of how they might be achieved for some common gaming applications are covered in Chapters 8 and 10.

SUMMARY

Emergence, then, is everywhere in video games. Most of the time we let it happen, and it is only benign. Sometimes, however, we want to encourage a more active kind of emergence to take place, which we hope will enhance the gaming experience. This behavioral emergence should lead to permanent in-game changes to the entities' behavioral models. It can also be used to train algorithms outside the game environment for introduction into it.

Emergence is present in single entities, groups of interacting entities, and when using genetic algorithms or genetic programming. These illustrate three distinct kinds of emergent behavior:

- As an application of sets of rules.
- As an application of rules triggered by others' behaviors.
- As an alteration of rules inherent to an entity over time.

The chapter has shown how the system can be designed with emergence in mind, taking advantage of the building blocks from Chapter 5, along with advanced techniques, such as GA and GP. Two kinds of emergence were also examined: strong and weak. In the interest of consistency, it was shown how weak emergence, where we can backtrack from effect to cause, is probably the most useful technique most of the time. Using weak emergence allows us some semblance of control.

Controlling Emergence

Until now, we have been dealing with emergence as if it is something that happens outside of the play session. We have used it as a way to build an organism and as a way to describe the constrained behavior of groups, as well as a useful approach to breeding algorithms and entities for the gaming environment.

As was pointed out, there is a caveat for this approach: Very bad things can happen when using emergent behavior that is not tightly controlled. We have to ensure that the process of restricting the emergence does not negate our reason for using it in the first place. After all, if it is restricted too much, we might just as well create a system of static rules and ignore emergence completely.

The trouble with emergent behavior is that the very aspect of the approach that gives it power can also be its downfall. Steps must be taken to ensure that combinations of independent behavioral patterns do not feed into each other to produce unwanted, dangerous (within the confines of the game rules), or erroneous behavior. To do so we need to be able to:

- Spot evidence of runaway behavior,
- Identify the principle cause (element), and
- Restrict its effect on the population.

Runaway behavior can take many forms. For example, the endless streams of vehicles colliding with each other in *Transformers: The Game* (not to be confused with the earlier game, *Transformers*) or the slightly mindless AI associated with some FPSs where soldiers run around in circles or up and down ladders when cornered are both examples of rules leading to behaviors that just didn't quite work.

Unwanted behavior can be benign, but it can also be less benign, leading, for example, to a game that cannot be beaten or one in which everything seems to turn against the player. Reviewers and players might be able to shrug off some of the more benign behavioral loops, but they will find it less easy to forgive AI that proves unbeatable in certain situations. To recognize unwanted behavior, it needs to be identified in terms of the game rules. The only way we can do this is by measuring its impact on the game environment—either that, or build an AI system to oversee the other AI systems and use neural networks to trigger appropriate responses.

Developers are often quite wary of emergent behavior—or at least they worry about the way it will be implemented and the complexity that it will add. They also worry about how they are going to debug it. After all, if the code goes wrong, there are many places in which there could be an error, such as in:

- The entity script (or dataset),
- The entity-processing engine (script engine),
- The game environment (or engine), or
- Any supervisory AI (which may itself be built on emergent behavior).

Emergent behavior also tends to feed back into itself. The mechanisms that we put in place to encourage the system to alter the application of the rules (and in some cases, the rules themselves) open up the possibility that small adjustments become large ones, which causes the effects to be further amplified. Thus, the behavior can spiral out of control and result in strange behavior. Unless we can identify it and the cause, this behavior will become an issue. Play-testing will catch most of the problems (and Chapter 7 describes some clever ways of doing that), and we can always provide for a general "reset" condition if things progress too quickly in the wrong direction.

Nonetheless, publishers can be wary of emergent behavior; they recognize that there might not be the expected level of control over the system. In addition, since more often than not they are the ones paying for the development, there is also concern over the additional resources required to make sure that any in-game emergent behavior works correctly.

However, using emergence outside of gameplay, during design and development, can help to reduce development time, because we can allow the system to balance itself by creating entities that are fit for use. So long as we make sure that they are not overcapable, or that we use some kind of mitigating algorithm to reduce their effectiveness, it is a powerful technique.

References

[GRAND01] Steve Grand, *Creation: Life and How to Make It*. Wiedenfeld & Nicholson, p. 62, 2000.

[GRAND02] Steve Grand, *Creation: Life and How to Make It*. Wiedenfeld & Nicholson, 2000.

CHAPTER

7

TESTING WITH ARTIFICIAL LIFE

In This Chapter

- Testing A-Life
- Bottom-Up Testing
- Testing AI with A-Life
- Testing A-Life with A-Life

The purpose of this chapter is to situate A-Life within one of the most important parts of the video game development lifecycle—testing. When deploying AI and A-Life, and especially adaptive and emergent systems, the testing paradigm has to change. The lessons learned will apply to design as well as development, and of course, testing; so testing is a wise inclusion. You will learn a lot about the implementation of AI and A-Life from actually testing it, so this chapter represents an important part of our general education in artificial life.

Results might vary, depending on an incredible number of different possible input values. Testing is traditionally about checking that one particular part of a system works consistently in as many situations as necessary to be confident that the system as a whole will function to a given level of reliability. The trouble with adaptive and emergent systems is that it might be impossible to guarantee that all combinations of input data and all combinations of system components have been tested. Furthermore, emergent behavior leads to unpredictable results. In a system that is as interconnected as a game, this means also that the inputs to various components (being the outputs of others exhibiting emergent behavior) are similarly unpredictable. The resulting feed-forward effect leads to something described by chaos theory—a small variation in the inputs producing radical changes in system behavior.

The result is that (in common with other real-time systems) the nonlinearity of the system requires much wider test coverage. In other words, the total breadth of test coverage and the number of test cases included in the testing process must increase in proportion. This is simply not practical, so we have to set a limit that reflects our confidence in the system and strive to test up to that limit.

This is not news to game developers, of course. Every system put together suffers from these same problems, which is why play-testing was invented; employ enough play-testers and we can be fairly sure that there will be ample coverage to ensure the system's viability.

However, testing the A-Life paradigm is no longer about just making sure the known behavior is working. It is also about checking for the things that might happen (adaptive)—systems that lead to behavior that wasn't expressly provided for (emergent behavior) and that might have a bad effect on the game.

The cornerstones of many of our AI and A-Life discussions have centered on adaptive and emergent behaviors. When the system is built, we can take into account the expected adaptive behavioral traits—in other words, we can more or less guess what the system will adapt to and how it will evolve. We can also cause situations to occur that will force the system to adapt. This can be as simple as pretending to be a less-capable player in order to check whether the adaptive systems downgrade the AI accordingly, or as complex as actually loading datasets that have been manually created to populate the system with specific parameters, thereby forcing certain situations for test purposes.

So we can test adaptive algorithms through extensive play-testing and manipulation of in-game data. This is less true for emergent behavior. Remember, most (if not all) video games exhibit emergent behavior, but we are concerned more

with the kind of emergence that serves a particular purpose. For example, a traffic queue in a racing game represents emergent behavior; we didn't expressly plan for it, but it can serve a purpose. It can slow down the pace for a moment, letting the player catch up and show off his skills at overtaking the competitions. On the one hand, it is an irritating side effect of the driving AI and course layout, while on the other hand, this is a handy solution to a perennial problem: how to give poorer players a helping hand.

This is a simple example that still needs a lot of testing to ensure that the emergent behavior is as desired. We need to make sure that a good player is not unduly held up by irritating, contrived traffic jams. At the same time, the racing must be improved so that good players do not get an easy ride and less capable players are not intimidated. A lot of this can be designed in, but there will still always be doubts that something might happen that will break the illusion and make the game less successful than it might otherwise be.

This level of complexity can make designers and developers shy away from using A-Life and emergent techniques extensively to create behavioral models. After all, it is assumed that the entities' behaviors will alter in an emergent fashion and not be just a transient fix for an immediate situation.

However, there is a third way that counteracts many of the issues relating to the sheer volume of tests needed: Do them automatically. Rather than play-testing or trying to contrive situations that may or may not be representative enough to validate the AI and A-Life under construction, we can put A-Life to use in the testing process.

This chapter concentrates on using A-Life to test A-Life—or at least uses A-Life–style algorithms to help the testing process. Using A-Life in the testing process involves two areas:

- Interaction and simulation, and
- In-game data generation, such as behavior and dynamic levels.

The first area relates to everything that provides the interactive experience, including any adaptive rules, reactive algorithms, and general play effects, such as animated fight sequences. The second area relates to the various settings that describe the game environment and the initial state of play, or input parameters to any flexible AI or A-Life rule-based algorithms.

We also have two testing-related issues to address. On the one hand, we need to test the game, and on the other hand, the processes that use AI and A-Life to test the game also need to be tested. Fortunately, the two are related, and there is (as we shall see) substantial overlap between the two.

This chapter will look at how to test the A-Life that we create as an artifact in its own right, as well as how to use those same techniques to test the rest of the game. First, however, we need to look at how to test the A-Life techniques so that they can be used in testing games that may or may not have an A-Life component.

TESTING A-LIFE

Whether the video game design and development team chooses to use A-Life techniques in the game or not, if A-Life will be used during the testing process then the developers must be able to test it. This is vital. It is useless to proceed without understanding that the test tools' quality must be at least as good (if not better) than the game itself.

A game that uses A-Life as part of the gaming experience will overlap the A-Life testing tool in many areas. A nonexhaustive list of these areas includes the following:

- The game environment
- The script engine
- The entity scripts
- The higher-order AI functions
- System/management AI functions
- Adaptive and emergent behavior constraints

The guiding principles behind A-Life as a testing tool, as seen in Chapters 5 and 6, relate mainly to the adaptiveness of the algorithms, as well as the digital genetics that are at work behind the scenes to create the in-game A-Life that will form part of the test cases. This will become clearer, but for now, we need to look at the basic theory behind using A-Life as a test tool in conjunction with standard AI and manual testing practices.

In a simple game, there is only the game environment versus the player to test. The player is pitched against an opponent whose actions are contrived to make it difficult for the player to win. Some industry professionals maintain that even the most static set of rules can be called "AI," but for the purposes of this book, these kinds of games are not put in the AI category.

Take, for example, the simple game of Battleships (which is included in the A-Life examples in Chapter 8). Battleships is played on two grids of an arbitrary size, upon which the players place boats of varying sizes (say two to six units long). They then take turns calling out grid references where they think that the opponent's vessels might be placed. Each time they uncover a vessel, a peg is placed in his board to indicate that a hit is scored. The winner is the player who manages to discover all of his opponent's battleships. There is very little AI involved in this game. The perfect opponent can be created with only a scratch-pad memory (noting misses as well as hits) and a collection of very simple algorithms for making sure that no turn is wasted unnecessarily.

In Battleships the player plays against the computer, so we only need to check that the computer never chooses outside the grid and that the perfect memory we have endowed it with does not fail. We should also check that the algorithms in place to help/hinder less/more capable players work by pitching the computer against itself.

Here, we start to move away from what could be considered a "simple" game and into more interesting A-Life possibilities. For example, an algorithm can be written that chooses between several possible search-and-destroy algorithms of varying success rates.

Experiments with Battleships have shown that a simple algorithm firing at random into the grid takes, on average, around 1,000 shots (into a 20×20 grid, with five vessels varying from two to five units in length). If we upgrade the algorithm to provide for a perfect memory, never firing into a space that has already been "tested," experiments show an average of around 300–325 shots. As it turns out, the best algorithm is one that, upon scoring a hit, proceeds to find an adjacent square that has not been "tested" and then fires into it. If no hit is scored, then it defaults to the memory-enhanced random-shot algorithm. If it is unable to find an adjacent grid location that has not been tested, then it again has two possibilities—resort to memory-enhanced random firing or expand the search.

The expanded-search algorithm also has two variations. The choice is between a simple expansion of the search radius and a clever algorithm that takes note of the suspected orientation of the vessel that has been hit, based on locating adjacent grid locations that contain direct-hit data.

So, how do we test the AI-enhanced version of our simple game Battleships? By pitting the computer against itself and checking the outcome, of course—an elegant solution, and one that (if run for long enough) will test every possible situation.

In an AI-enhanced game based on scripted behavior, we also need to check the AI and scripts, along with script engine and higher-order control functions. Our Battleships game could well have been implemented with an underlying script facility. All that's needed are operations for changing the grid location to test and some very basic condition processing.

If the algorithms are changed from pure code to scripts, then an extra layer of complexity and flexibility is suddenly added. The complexity that we have added is in processing these scripts and making sure that they are correctly interpreted. In addition, the interpretation mechanics must be tested—things like maintaining the correct pointer to the current instruction (since we need to allow test and branch constructs), possibly a stack of values to act as local memory, and other mechanisms. Some of these will require interaction with status information from the game, as well.

If some of the A-Life paradigm is desired to generate scripts that are "bred" from existing scripts, we need to test the genetic programming algorithms that give rise to new populations of scripted behavior. To do this, it must be approached in stages. Fortunately, this is one facet of the testing method that will not be lost; after all, if we are going to use A-Life principles to test the game, then we need to be sure that the genetic programming or genetic algorithm techniques are correct.

The first step is to make sure that the basic combination algorithms work correctly. These range from simple split-and-merges through to crossover and mutation techniques. The best way to do this is to start out with simple arrays of numbers, which makes it easy to see where the algorithms might be going wrong. For example, suppose we have two sets of numbers representing two scripts as shown in Table 7.1.

TABLE 7.1 GENERATING A CHILD SCRIPT

Script A	Script B
1	10
2	3
3	11
4	12
5	13
6	14
7	3
8	
9	

From Script A and Script B, we want to generate a child script. The two scripts are intentionally of different lengths, because we want to make sure that the recombination algorithm is robust enough to deal with it. Next, we might isolate certain numbers to denote the end of a logical block. Just recombining lines arbitrarily is no good, as this will likely violate the logical flow of the scripting language. There are two options: Either recombine entire blocks of behavior or make sure that we never cut across a logical block within the script. (In Chapter 8, the implementation of a simple scripting language for the Battleships game is shown; for now we are just concerned with how it will be tested.)

Assuming that we just want to perform a simple genetic crossover (without mutation at this point), then block number "3" would be chosen as the logical block delimiter and two children created, each sharing parts of Script A and Script B, as shown in Table 7.2.

TABLE 7.2 CHOOSING A GENETIC CROSSOVER

Script A	Script B	Child A	Child B
1	10	1	10
2	3	2	3
3	11	3	1
4	12	10	2
5	13	3	3
6	14	4	11
7	3	5	12
8		6	13
9		7	14
		8	3
		9	

Since numbers were used, it is very easy to spot an error in the child chains, should one occur. Note also that we have used sequential numbers, and that it is easy to tell which numbers have come from which parent (except the logical delimiter, "3"). Once these have been tested, and we know that the A-Life genetics work correctly, the results can be mapped to behavioral scripting (which will be covered in the "Bottom-Up Testing" section, later on in this chapter).

The next stage to test is the adaptive side of the game. In our Battleships example, the system is able to select an appropriate algorithm, depending on the outcome of the last shot. It adapts to the circumstances, but could also be made to adapt to the opposition. In an adaptive game, any AI that is designed to make the whole system adapt to the player must be cross checked. We may have some control over the adeptness of the system as a player, depending on whether we have chosen rule-based AI (the basic Battleships game) or some kind of scripted A-Life that can provide varied play patterns using genetic algorithms or genetic programming.

Part of the testing cycle, then, is to differentiate between errors in the programming and intentional mistakes that are designed to reflect real errors of judgment or memory.

In the global system, there are two separate testing issues to address. First, there is the testing of the AI systems that drive the A-Life. They need to be tested such that the developer is confident that they function correctly with respect to their design. In most of this discussion, we assume that this takes place as part of the normal software development cycle—that is, as early as possible.

The testing of the behavior, in the A-Life context of this discussion, happens at a later stage, when we are confident that the *components* that make up the system

function correctly. Then, the balance of the interconnected components that make up the system can be addressed.

This is why components must be tested separately.

An adaptive driving game, for example, might include an A-Life component that alters the various driver personalities according to algorithms. The adaptive nature might well be simple manipulations of the bounds within which these facets can vary. For example, we might wish to have an aggressive driver and, on a scale of 1 to 10, allow his aggression to vary between 7 and 8 (fairly aggressive). A less aggressive driver might have an aggression factor that varies between 1 and 4 on the same scale.

During the game, the aggression of the two drivers can be varied with respect to their other defining characteristics, but only within these bounds. However, if the player is seen to be driving in an aggressive fashion, we might choose to raise the upper limit (and possibly the lower limit) of the aggression "gene." This might lead to emergent behavior that is detrimental to the success of individual drivers and make it far too easy for the player to beat them—something that has been experienced in the game *Squadra Corse Alfa Romeo*, which uses a similar system. Therefore, in an emergent game, whole populations of generated and player-controlled entities must be checked to make sure that there are no issues related to observed or not-yet observed circumstances/events.

Games that are open ended, A-Life–oriented simulations (like the *Sim* series—*SimCity*, *The Sims*, and *SimEarth*) suffer from these unanticipated transients. The game developer can only guess where the player might try to take the game; after all, we only provide him with the gaming environment and the means to have fun. We cannot control, beyond the rules themselves, what the player might do with the game. To be completely sure that everything works as it should, *years* of play-testing would be needed, or the game environment would have to be restricted so much that any sense of freedom would be lost. Of course, once we know how to spot errant behavior, as described in the next section, we can invest that time automatically, using A-Life.

Testing with A-Life

In the previous section, we took a detailed looked at how A-Life can be tested. Turning that around, the exact same techniques can be used to deploy A-Life as a testing tool. In other words, A-Life principles (culled from the A-Life paradigm in Chapter 4 and the building blocks from Chapter 5) can be employed to test for us.

As a side note, any game can be adapted for testing with A-Life as long as there is a hook into the system. In fact, with appropriate automation software that can direct input (via keyboard or mouse hooks) to the game and recognize certain visual cues, any game can, to a certain extent, be tested using A-Life.

All that is needed is an appropriate use of behavioral models designed to stretch the system, a way to control them, and a way to record the success or failure of the system to react appropriately to the A-Life test constructs. The better the framework and the better the support for hooking into the game, the easier this will be.

If we have A-Life constructs in the game, AI can be used to help test their behavioral patterns and spot any problems by manipulating rules and data environments.

For example, the Battleships game uses adaptive AI to change the algorithms that it selects to choose targeted grid locations. These algorithms might well be implemented as scripts, or at least, the selection mechanism might be implemented as a high-level script. There will not be very many variations in the behavioral patterns, because not many different actions are possible. We only have four shot-selection algorithms—random, memory-enhanced random, radius search, Bessel calculation search—and one action whose result can be tested—hit or miss—so the scripts that select between these possibilities will likely be very simple.

Dig down a level and the actual algorithms can also be encoded, using a scripting language. In addition, we have various "fudge factor" mechanisms that can be used to dumb down or smarten up the simulated player (computer) to make the game feel more natural and interesting.

Once all of this is done, we then have all the mechanisms in place to turn the tables. If we think of a player as a set of data that dictates how he will play the game, then a dataset can be populated in advance to simulate that player. Using the Battleships example, this will be a fairly simple exercise. However, with games that are more complex, the dataset that defines how the player acts/reacts will be much more complex. Many of the AI mechanisms that are used to control the in-game entities, once tested, can then be used to play back the dataset, *simulating the player*. And we do not need to have encoded an actual game environment; the game can play out virtually and yield results—and much faster than a human play-tester.

Then the dataset can be populated using A-Life and some clever mechanisms, such as digital genetics to create child datasets; in this way we can hope to get wide test coverage, probably wider than with other kinds of testing. The same mechanisms used in the game to generate behavior can also be used to test that generated behavior.

The datasets can be used to populate player emulation models that will exercise the engine, as well as any flexible/adaptive/emergent game rules. All that is needed is a good routine to generate the datasets, which can be done at random (see "Pseudorandom Testing," later in this chapter), but it might prove to be better to apply some logic, as well.

Referring back to the Battleships example, if random scripts of random commands were generated, we might eventually stumble across behavior that is reasonably believable. However, it might make more sense to create logical blocks, rather than just random commands, and recombine them to get better results.

If a reasonable mapping has been defined from the abstract datasets to the in-game data, this can be taken a step further. We can apply datasets in-game to generate dynamic resources that are required—from flexible rules and adaptive systems to dynamic levels and so on. Again, A-Life techniques will be used to test the game and make sure that the results are as expected within the confines of the game rules. In other words, these generated datasets will test the way that the system responds and creates the environment for the player to play in. There is one small catch: For this to work, errors must be detected as automatically as possible. Otherwise, we would

have to either spend human resources looking over reams of abstract test results or wait for the interface to be created and play back the results, and then keep an eye out for errors.

Either of these will reduce the efficiency of the approach, but these are not the only caveats that deserve consideration when deploying A-Life and AI as test tools.

Automatic error identification is an AI discussion in itself. It is worth taking a time-out here to think a little further about what it entails. First, we have to know what an error "looks like." Clearly, visual errors are going to be nearly impossible to detect. Clipping errors, such as when a wall suddenly disappears and reveals what's on the other side, cannot easily be detected except by extensive screen-reading.

On the other hand, behavioral failure cases might be much easier to detect. Fuzzy systems that compare behavioral patterns with design patterns (or known good patterns) can be a great help, as they can assign a level of confidence to the output of a series of input values and alert human testers to possible problems.

If the results are then fed back into the system, it will begin to learn what the limits of the system are and produce fewer false alerts. Still, it requires a human tester to analyze a cross-section—some good and some bad—to build confidence in the testing process.

Of course, it is much easier when crisp pass/fail criteria can be established. This is helped by choosing an appropriate level of granularity to test at. The finer grained the testing is, the more likely it is that a series of pass/fail (almost binary) test cases can be built up.

Indeed, we might say that the ease of analysis is linked to the test design. The more constrictive the test cases are in the expected behavioral output, the easier it will be to install automated error detection. However, the power of A-Life is in being able to test on a much wider basis, so we should not restrict the test cases unnecessarily.

As is so often the case with A-Life, we must strike a balance between the power of the A-Life system and its emergent potential, and the reliability and predictability of the result. This goes as much for testing the game as is it does for the game itself.

Caveats

Primarily, the caveats to using AI and A-Life as test tools involve making sure that the ends justify the means. If more resources are expended in creating the test tools than in creating the game itself, then clearly, we need to start asking ourselves whether the end result is worth the extra effort.

As long as the test designers can build in reusability, the extra effort becomes much easier to justify. It is also easy to justify effort that yields a better overall confidence in the quality of the behavioral systems (or indeed the game). Satisfying both of these in addition to any other criteria for the usefulness of A-Life constructs as a test suite will ensure that the extra effort can be repaid in full.

It can take a lot of effort to create programs that produce scripts and datasets. Many programmers enjoy writing code that writes code, but it can be a time-consuming job. A certain logic also needs to be applied, so design time will be used

up, as well. To minimize this, we need to reuse as much of the code that will be the actual game as possible. (The Battleships example has pointed out where this might be occur.) Not only is this a good design choice in terms of resources that need to be expended in order to achieve the end result, each piece of reused code (or design pattern) is tested twice. In other words, we test it once (possibly manually) as a piece of code, then use it to test itself, thereby giving it another pretty thorough workout.

The testing principle must be extended to include the monitoring systems designed to keep the adaptive and/or emergent AI units in check. This includes the reuse and testing-twice functions. Otherwise, the cost of development will outweigh that of failure, and we might as well go back to rule-based AI systems that are more reliable. Since these kinds of algorithms are less capable with less varied behavior than their emergent cousins, they will be easier to test.

Even if we choose the rule-based path, however, A-Life will still have an important role to play in the testing paradigm (see "Testing AI with A-Life"). So even if only AI and rule-based reasoning systems are used in the game, our A-Life education comes into play as a test tool, even if it is not deployed in the game.

Since all games that are based on independent rules are intrinsically emergent, we need to keep an eye out for emergent errors. The caveat here is that we need to keep one variable static each time we test in order to be sure which element causes the (emergent) behavior. This can make it more expensive to test using A-Life; many iterations might be needed to test all the behavioral aspects. The way to make sure that resources are not wasted in the test cycle is to deploy "bottom-up" testing.

Bottom-Up Testing

This part of the chapter deals with one of the major problems encountered when using AI and A-Life in video games—how to test in such a way that the results of the emergent behavior are those that were originally intended. This issue is not limited to video game development; it can apply to any software-engineering project. Testing AI and A-Life in video games has its own specific set of problems, many linked to the misconception that everything but the AI must be developed before it can be tested—in other words, that it is not possible to test the AI until the entire game environment is in place.

By now you should realize that this book takes the approach that AI and A-Life can be implemented in the game as we go along; there is no compunction for anything beyond an abstract representation to be present at the time that the AI and A-Life are implemented. Some parts of it can even be implemented and tested in the absence of the rest of the system, such as script engines or digital genetics. More than one of the commercial projects mentioned in the opening chapters of this book have suffered from lack of resources, resulting in a more modest application of AI than the original design called for.

This is why the emphasis has been on bottom-up development for AI/A-Life, as well as for testing. After all, if we can develop from the bottom up within an incomplete game environment, then we can also test in the same fashion.

The AI and A-Life algorithms can be tested as we go along, as they are developed, and independently of each other. This will help to ensure that the correct behavior emerges and allow us to be confident that if inappropriate behavior emerges, it is not the algorithms themselves that have caused the problems, but a combination of them. This is important, because the system will become large—too large to cope with in any other way than bottom-up; there will simply be too many variables. The art of manipulating the test cycle is in knowing what has caused an error. Hold as many things static as possible while varying a specific function (or minimal functions) to try and reproduce the error.

Clearly, we would like to be as certain as possible that the algorithms themselves are not at fault, as well as the underlying mechanisms, before we start testing them in combination. That is the principle of bottom-up testing.

When testing does take place, graphics subsystems, menus, or other GUI widgets are not needed. In other words, the actual game does not need to be in place. All that is needed is the internal representation of the game environment. Much of the actual data that will eventually represent the game universe can also be generated as abstract numbers.

Given all the things that need testing, it is best to start as early in the development cycle as possible. Otherwise, there will likely be issues that lead to parts of the AI being shelved, as happens when features become classified as "nice to have"—and the resulting game will likely suffer.

There is an order to the testing process, though, if the bottom-up paradigm has been followed for the AI and A-Life being used. There are three principle areas to be tested, usually at different stages in the development process.

First, we need to test the underlying mechanisms for the AI, as they will be implemented in the video game engine. These will include mechanisms that facilitate the development of AI and rule-based systems, such as those from the building blocks, and might include:

- Neural networks, and
- Abstract structures like FSMs and FuFSMs.

If abstractions of these can be tested early in development, and they work, then we can forget about them and concentrate on the mechanisms that are designed to utilize them. Even if only stub code is placed where normally there would be processing of data or data structures not yet defined (code that returns a fixed value and does no real processing), the whole mechanism can be tested very early on.

Developed properly and with time taken to design abstract, reusable structures and processes, the code can be reused in future projects. After all, if we are going to make a substantial investment at this stage, it might as well pay out over the long term.

Next, we need to bottom-up test the processing mechanisms. These are the parts of the AI and A-Life that provide the flexibility to create an environment that is easy to tailor to specific needs. They are one level above the previous elements and will

need to work in conjunction with those elements to provide a good platform for the AI/A-Life. Examples of these processing mechanisms might include the following:

- Script engines
- Script management, such as script recombination and automated generation

Ideally, we want to validate these in the mid-stages of development or perhaps earlier. In the case of script engines, for example, they might provide the basis for balancing the gaming system or even serve as part of the level design. In addition, we need to make sure they work because they are also the foundation for the adaptive and emergent systems. Again, if testing them is left to the end of the cycle, it will be difficult to separate out any errors in the programming itself from errors in the way that the different pieces fit together. For example, the scripted side might just be an internal collection of processing rules or even a full-fledged scripting environment, complete with decision processing. In either case, it must be tested to ensure that the mechanism works before being deployed for further game development.

Those parts that have been designed as accessible to level designers or others outside of the development team must also be tested before they are deployed—and tested early on in the development cycle. This is particularly true of any routines that are designed to modify the scripts automatically, perhaps following some AI rules and direction. Again, if they are part of the general testing for that specific function, it will prove that much more difficult to find errors that may exist due to the emergent behavior and relative opacity of the system.

Finally, we need to bottom-up test the management routines—all the routines that provide the game rules and prevent in-game catastrophes. Remember that the video game AI and A-Life is designed to create the illusion that the game is thinking for itself. Therefore, there is a high risk of error, so we need to be sure that the checks and balances that are put in place work well enough to catch errant emergent behavior. For example, we might test components such as:

- AI systems, and
- Adaptive AI systems.

Testing these areas requires separate approaches. Three paradigms have be outlined that can help with systems that employ rules (FSMs), engines (script processing engines and the like), and what will be called "pseudorandom testing," which works with large datasets that contain unpredictable but repeatable sequences of test data.

Rule Testing

Rule testing is based on the guiding principle of putting test data into rules and monitoring the output. It should be easily automated and output conditions broken down into empirical values, which makes cross-checking and debugging easier. Each rule should perform one operation that yields a known result.

These are relatively simple principles, but combinations of rules can create emergent systems (see Chapter 6) that can be hard to control. Even if the output of each rule can be verified as correct, it is probably being fed into other rules as input data, such as with neural networks. In these cases, the emergent behavior is difficult, if not impossible, to predict. Examples include Conway's Game of Life; which is based on simple rules that are easy to validate in isolation, but combinations of the rules make determining the outcomes in advance quite difficult. Therefore, we need to make sure that enough test conditions have been created to enable the trapping of all possible problems for a given rule. Only then can we be sure that the rule is working correctly and that any erroneous behavior is being caused by something else, such as simple emergence, an error elsewhere, or bad balancing of the game rules. A rule can be processing correctly, but it can still yield results that are outside of a certain expected acceptable range.

When following the caveats mentioned here, remember that we must be able to spot both good and bad behavioral traits. The bad behavior might be corrected by tweaking the underlying rules (manual or automatic) as they are built up into the combinations of rules that yield the actual in-game behavior.

Tweaks that can be done automatically include using adaptive algorithms to change game rules in response to behavior that seems to be out of bounds, given the current game state. (This is covered more extensively in the section, "Testing A-Life with A-Life.") Manual tweaks include manipulating the specific settings of data that is designed to make sure that the behavior never exceeds controlled boundaries.

In order to create the test data, models must be built to test the rules, using the bottom-up approach, and each component is tested before adding it to the mix. It is necessary to build up logical models that represent different game states and inputs and use these to test the components.

It might work to try and test them by generating datasets purely at random, hoping to cover all possibilities at least, but this is very time consuming. Conversely, we could populate the in-game rules using datasets that are arrived at using specific techniques (such as pseudorandom generation) in a controlled fashion. The benefit of testing in such a controlled way is that we can keep the known good settings static. If only a single element is being varied in each iteration, it will be much easier to find the cause of a problem. By building up known sets of good (and bad) test data—which have positive and negative outcomes—the overall testing strategy can cover a very wide set of cases for little additional effort.

The key here is in being able to validate the results automatically, test in small increments, and keep certain rules static—thereby avoiding the emergent behavior problem. It is always necessary to be able to backtrack to the cause of an error; otherwise, testing is useless if there is no way to single out the piece of code to change! This sounds obvious, but it is worth pointing out. As with other complex systems, there is the temptation to cut corners and simply hope that all comes out well in the end.

This is where a combination of bottom-up and top-down testing can be useful. For many systems, a combination of top-down and bottom-up testing will ensure that the effects of emergent behavior can be adequately combated. However, a system-level test will not usually give more than an indication that something is wrong.

So the testing needs to continue at ever-deeper levels (hence the name top-down) to try to find the exact cause of the error. For rule testing, this means creating some system-wide use cases that can then be broken down into smaller coverage units, the deeper down in the system they occur.

Some of these units will likely coexist with, if not be directly equivalent to, existing test cases used in bottom-up testing. The only difference is that, beyond a certain point, the bottom-up testing will be replaced by test cases that combine several aspects of the system and become system-wide tests that can only be executed once the whole system is in place.

Bottom-up rule based testing, therefore, might not be capable of testing the entire system, while top-down testing leaves too much to the end. A combination ensures a healthy mix of timeliness and system coverage.

Engine Testing

Since much of the AI and A-Life will be based on processing chains of commands, it is necessary to test the script engine as thoroughly as possible. This book has been geared toward making the AI and A-Life as extensible, flexible, and adaptive as possible, even to the point of being self-modifying.

The core of all of this is to be able to selectively execute rules or even create rules from selective (or conditional) action execution statements. To do this, a gaming system needs some kind of mechanism that permits the game to run through the available options and decide what to do—a script.

Even if the scripts are internal—that is, "opcodes" representing basic sequences—it is still necessary to make sure that the engine cannot crash and performs as it is supposed to under all conditions. Of course, the more complex and rich the scripting language, and the larger the range of possibilities, the more difficult it will be to test.

On the one hand, a script can be a list of numbers, with each one referencing a piece of behavior or a condition; on the other hand, it can be an actual language that is implemented in such a way as to make it almost infinitely flexible. Where scripts are self-modifying, there are additional, specific problems to contend with relating to the kinds of situations the engine will get itself into. These relate as much to being able to create an AI system that respects the syntax of the scripting language as they do to problems concerning the end result.

Basic, flexible, adaptive systems rarely have the chance to create the chaos that self-modifying sequences of commands can create. An engine that processes a language consisting of commands and variable values, for example, can possibly alter the variables and produce variations in the behavior. This might not create anything particularly unstable, except through the emergent behavior resulting from the addition of other rules and scripts. But these kinds of conditions can be tested for, as long as ranges of possible values are defined. However, as soon as the script AI can generate new sequences of commands, either in a controlled fashion (manipulating blocks of commands representing known algorithms, but just changing the order,

for example) or a more open fashion (creating new scripts with previously undefined behavior), there is much more potential for gameplay errors to creep in.

Once again, if the underlying mechanism (script engine) has proved to be correct, it is one less variable to contend with. That means that any errant behavior must be the result of script management, generation, or manipulation. By the same token, it is also necessary to test the game engine that handles all the graphics, physics, sound, and so on—which can only really be done by a human. It is possible to reduce the workload a little, however, by following a similar bottom-up approach.

If testing is performed in a bottom-up fashion, a first step can be to test the internal representation, early in the development cycle, by throwing all kinds of scripted player behavior at it. This behavior can be generated as a result of some of the techniques that appear in the following sections, or it can be prescripted.

When the internals are verified as working (that the game environment bounds are respected) by checking path data, for example, and manually spotting out-of-bounds movement or other anomalies, the same tests can be performed visually. In other words, because the engine has been tested, if there are any on-screen anomalies, the cause must be somewhere between the engine's internal workings and the screen.

This is a simple example, but it demonstrates, once again, why bottom-up testing is beneficial. In the long run, if AI techniques are used to spot anomalies in the internal data, and humans are employed to spot anomalies in the visual data, the testing process becomes quite efficient. Furthermore, when bottom-up testing is used to verify various parts of the system, it becomes easier to debug (remove errors) in the actual code. For example, if A-Life techniques are used to generate behavioral patterns for an army, a human can watch as the action unfolds and attempt to spot errors.

This is part of using AI and A-Life to test interactive software, including games, and is covered in more detail later on in this chapter under "Testing AI with A-Life." First, though, there are some important issues to discuss on the subject of test data generation.

Pseudorandom Testing

An article that appeared in *Dr. Dobbs*, entitled "Pseudorandom Testing," was written with test coverage in mind. [LECKY01] The theory behind the article is worth recounting here, because it provides the basis for creating test datasets when using AI and A-Life techniques. The article proposes a way to generate large sets of test data and hope to achieve wide test coverage without resorting to manual generation. In other words, it can be possible to test both the *bounds* and *sequence* of values within the test dataset. By covering both of these aspects, the test coverage is wider than just trying to sequentially plug every value into an algorithm and check what comes out on the other side. Pseudorandom testing recognizes that the sequence of events is also important, as is repetition and the repeatability of test results.

So how does this relate to video game testing? Testing fight sequences is a simple example, in which a list of commands can be put in a variety of different orders. The tester would like to ensure that the transitions are correct. Not only that, he also

needs to know that any mechanisms that have been put in place to respond to those sequences/transitions are also correct. For example, there might be defensive triggers that can be trained against sequences of fighting events. The keywords here are "triggers" and "train." Not only can pseudorandom testing be used to validate behaviors and their triggers, but it can also train a system to respond in a certain way, based on AI and neural network principles.

Put into context, this means that for every sequence of test data, there might be a set of responses that, in testing, prove to be more or less successful. Assuming that the AI can tell the difference between a good (successful) response and a bad (unsuccessful) one, it is possible to train the neural network to attach more weight to the good responses. This is yet another application of AI and A-Life building blocks; they yield powerful results when combined with other test data-generation mechanisms.

Take, for example, the fight sequence example; one entity might be playing the attacker and the other the defender. The test dataset is populated with individual moves and sequences of moves, which are replayed through the game rules. If a neural network that consists of free neurons is attached to the game engine, with connections made depending on sequences of moves—such as one for each kind of attack—it can be trained by making links of different weights between attacking moves (combos) and defensive sequences. The link strengths must be weighted according to the relative success of various strategies.

Playing back the attacks, the combos can be preempted with the right defensive moves, according to the strengths of association in the neural network. Not only has the test dataset tested the interaction between the two fighters, but it has also trained one to adequately defend against a set of test combos.

Whenever the game encounters a player that is likely to string together the same set of moves, it will have an adequate defense. The important thing to remember is that the neural network must be constructed based on:

- The likelihood of the chains of moves, and
- The relative success of any defensive measures.

If both components are not present, then either the prediction will not work or the wrong defensive moves will be chosen. The AI-training algorithm can then be reused in the main engine code. No effort is wasted.

However, to make sure that everything is working correctly, the test team will need a lot of data. Ordinarily, this would mean a great deal of play-testing and observation, but instead, the principle of pseudorandom testing can be brought into play.

The trick with pseudorandom testing is that the same set of unpredictable (patternless) numbers can be generated over and over again, thus satisfying the principle that the tests should be repeatable. The actual datasets need to be mapped to the gaming system being tested, but once the parameters are known and mapped to, say, a range of numbers, we can use standard, seeded, random number-generation techniques to create a large test dataset.

This test dataset is then used to trigger the game engine by mapping each value to something inside the game environment itself—such as fight moves, simulated

player movement in a shoot-em-up, or a random landscape used to test squad play, terrain handling, or pathfinding. When an error is found and repaired, the outcome is known *for the entire set*. This is much better than asking a play-tester to "try and do the same again," which sometimes is just not possible. It might give the impression that the error has been fixed, when in fact, it has not.

There is one final point. All of this is much easier to achieve if moves (such as terrain or actions) can be generated on the fly, perhaps with the help of a neural network or rule-based AI. This helps to reduce an unnecessarily large test dataset for active training. However, the initial tests should be done with as large a set of test data as possible.

TESTING AI WITH A-LIFE

This part of the chapter deals with using A-Life–enhanced software applications to test video games that contain rule-based AI or similar behavioral modeling implementations. The principles hold for systems that contain A-Life algorithms, but the next section will build upon these further, as there are some other concerns to address.

What will be discussed here is good groundwork for the rest of the chapter and on its own is perfectly capable of helping to test games that contain fairly predictable AI. In this sense, predictable behavior does not necessarily make the game any easier, but it does enable the developer to be fairly sure of what to expect and, therefore, test. Emergent behavior might still occur, of course; this is the nature of any system that is based on multiple rules. If the bottom-up testing approach has been followed, it can also be assumed that any observed errors in the behavior are the result of putting the system together, and not any coding errors in the various components.

The general principle of using A-Life to test AI is to exercise the AI by putting it into as many situations as possible. We saw this previously with pseudorandom testing, but the test data that this technique creates needs to be massaged somewhat before it can be used for validation.

Since AI has been used in the game, it makes sense to leverage it in testing, as well. The test dataset can be used to create "bots" that can be made to play the game as if they are players, thus exercising it. These bots can be as simple as an external application that generates in-game events, or they can literally take the place of players. Bots can exist within the virtual game environment (as abstract representations), or they can be used late in the production cycle when everything is in place and play-testing begins.

Like a human player, the bot must be able to adapt and rewrite its own behavior, hence the insistence on A-Life. If only adaptive AI or rule-based systems are used, then only a narrow corridor of test cases can be evaluated. Even using pseudorandom testing as input, the nature of rule-based AI makes it less flexible than combining A-Life with these techniques to extend the test dataset while keeping it within reasonable bounds.

Knowing the Boundaries

From the onset, we assume that the behavior boundaries that stem from AI and rule-based systems are well known. In other words, everything that represents the internal play state of the game has a range of values that are allowed, and that it is possible to be fairly certain what the expected behavior will be (except for emergent behavior). For example, in a shooting game, entities might be able to speed up or slow down or vary their movements and/or shooting rates, depending on the skill of the player. As the player progresses through the game levels, new, faster, or more challenging behavioral models might emerge.

The game AI might also bombard the player and decide to drop them (or start dropping them) next to the player, on top of him, or at random. This might also be linked to the AI, which chooses where to drop the bombs and how many, according to the skill level of the player. This is an adaptive system of the kind mentioned in the opening chapters of this book; taken even further, the AI might work out how fast the enemy craft(s) should move and to what extent and how many of them should exist at any one time, either in flight or in formation. All of this has boundaries beyond which the system should not go, and it all needs to be tested. For example, if it is a formation-based shooter, the AI is also responsible for generating messages that tell the enemy craft when to break formation. This can happen at different times and intervals—again, something that can be tested against known parameters or boundaries.

Furthermore, each enemy craft is probably endowed with some basic logic (and possibly some AI) that will enable it to make rudimentary decisions, such as when to drop a bomb, how to move, and so on. These behaviors also have boundaries and can be tested.

So all of these functions are based on rules that have parameters associated with them—parameters that have limits or behavioral boundaries. Every game will have similar sets of behavior, parameters, and boundaries, all of which will need to be tested.

Some emergent behavior will be the result of the level design (or, if generated, the algorithms that generate the levels), the behavioral models of the enemy craft, the AI, or the interaction with the player. These might be more difficult to test, but applying A-Life in the testing paradigm will make it easier. However, the game rules and the layout must be respected so that the boundaries are known in advance. Otherwise, it becomes very difficult to determine where and how they have been breached.

For example, if the game rules dictate that a bomb may not be dropped while a bomb is already in the air, tests can determine occurrences of multiple bombs from a single enemy craft. The input to this might be the test data along with some adaptive algorithms that represent the perfect player, taking the game to its limits. By a similar token, if the game rules state that only one group of enemy craft can break formation at a time, then tests can be made to see if this ever occurs. These are very simple examples of empirical tests that can be carried out.

There might be other aspects of the adaptive rules that must be tested for—usually hard tests, not soft tests. In the previous examples, if there are adaptive rules that increase the rate at which formations can break and drop bombs, there are hard tests that can be carried out that verify correct behavior. In other words, these tests have binary/scalar results, such as yes/no, true/false, 1 to 100, and so on. As long as the behavioral boundaries are set in this way, it is possible to take the next step—modeling test behavior by using A-Life to exercise the system. Behavioral anomalies can then be detected, based on the output from the game engine. This can be carried out early on, since there is no compunction to have the whole system in place.

The A-Life that will be used is based on the same building blocks that were discussed for the creation of the in-game AI and A-Life. There is, therefore, a good chance that the project will be able to reuse some of the algorithms that have been implemented for both testing and in-game modeling.

Systems used for testing in this way must be adaptive and based on rules that can be discovered by the system or generated by humans. The test data can be used as inputs to trigger specific actions and elicit responses, and anything from FSMs to neural networks can be deployed to bring variety to the input behavior, as we shall see. To do this, it is necessary to be able to map the behavior accordingly—from the video game system to the test system—in a way that makes it easy to abstract the behavioral models.

Mapping the Behavior

As was seen in previous examples, automated and semi-automated testing requires, at the onset, the creation of a dataset that represents behavior, which in turn exercises the system under test. Without this "seed data," it becomes inefficient to create large sets of test data. This test data can be created using pseudorandom techniques or with AI, or a mixture of the two. Any test data is therefore an abstract representation in the same way that a bot script is an abstract representation of a behavioral model. By itself, this abstract data has no significance; it is only when mapped to the underlying system that it begins to become useful.

All of the A-Life techniques seen thus far can be deployed to create that test data—from combining a kind of digital DNA to creating neural networks of adaptive systems that can learn from their own input data. For those games that use scripting, there is an additional built-in abstraction—the scripting language—so all that is needed is a way to create datasets that represent the implementation of a certain kind of behavior. Put another way, the input test data can be converted to a scripting language, which is then deployed within the video game environment. This is essentially an abstraction of an abstraction. We have already identified the additional complexity that this incurs and the testing that needs to be carried out (see "Bottom-Up Testing").

For games that are not based on scripting, such as in our shooter example, it is necessary to provide a hook into the various functions that govern the behavior of individual components of the system. Since there is no existing internal abstraction, one needs to be created, and this is part of what is called *mapping* the behavior.

For example, assume that in the shooter example we are using flocking algorithms to manage the swooping attacks of marauding enemy crafts. Each flocking algorithm has parameters that manage the distance that each craft should be from its peers, among other settings that govern the in-flight formation, while allowing some autonomy to each of the entities in the flock. If crafts are not swooping, then they are just following a regular pattern, rather like in the original *Space Invaders*. Craft selection for swooping could be done at random (another parameter), with the AI choosing both the number of swooping crafts in the flying formation and where in the static formation to take them from.

To make it more interesting, a flocking algorithm could also be used for the static formation; one craft is chosen as the leader, and the rest follow each other. Doing this allows us to reuse the AI controlling the movements of the crafts. All that needs to be changed is the nominal "leader" and the algorithm.

By the way, this can be achieved by using a variety of AI building blocks. The key is to have an abstract mechanism that can be applied with specific routines that govern the behavioral outcome. That way, to change the behavior, only the parameterized routines need to be changed. One step up from that is to *generate* rather than *prescribe* the behavior, and the whole game moves from rule-based AI to A-Life. The shooter design also stipulated that no more than one bomb per craft may be in the air at any one time, which gives us something else to test.

These behavioral triggers need to be mapped to a set of values, which in turn are used to test the system by calling the appropriate function in the game engine. The values that are mapped need to provide both input (triggers) and output (response/validation) possibilities. For example, assume that the standard behavioral pattern for a non-expired enemy craft is based on a flocking algorithm (see Chapter 8 for a discussion of flocking). Some functions in the game engine cause state changes. A state change in this case might be:

```
Static ➔ Attacking
```

Since the test developers have access to the source code, they have the advantage of being able to map both the behavioral triggers and the results to values that can be easily checked. They will probably need to enlist the help of the development team to provide easy entry points to the code, and this should be part of the design.

In other words, it would be much more difficult to test the behavior if an AI system had to be built that could read the screen. This is why human play-testers are needed, but what we are trying to achieve here is automated high-coverage testing. The benefits are that the test process is foreseen in the original design, and it can be conducted in a bottom-up fashion as the game takes shape.

In this case, it is not even necessary to have a GUI to look at. At the very least, rows of numbers that represent the play state can be analyzed for unexpected behavior; at the most, the mapping can allow for more descriptive or even automated analysis. To sum it all up, there are two sets of mappings that are required:

- The behavioral triggers
- The game state

Whether or not the game has implemented a scripting or other abstraction of the in-game behavioral models, this does not change. Both kinds will need mapping to take advantage of the wider test coverage provided by AI and A-Life testing methodologies as presented here. All that changes is the mapping itself—that is, the values to triggers—and the possible addition of a piece of AI software that writes a script that is then put into the play session. This is the advantage that scripted game engines (however rudimentary) have over nonscripted ones.

Even if the script is just a collection of numbers that select built-in algorithms (as in the Battleships example), there are advantages in design, development, tuning, and testing that cannot be ignored. The more flexible the processing of the game rules, the more open the system will be to using A-Life at each stage. Still, it is important to bear in mind the caveats regarding the time invested. The goal, as always, is to keep the dataset generation as simple as possible or build the sophistication up in layers.

The approach taken will naturally have an impact on the level at which the test data operates. In this case, the term "level" applies to the abstraction (distance from game engine code) that the test data is mapped to. The earlier stages of bottom-up testing will tend to abstract to a lower level. When the whole game is in place, the abstraction can be very high—to the extent that the test algorithm is a virtual player.

The lowest level of abstraction is where individual functions are called that perform a single operation, such as MoveLeft, MoveRight, or Fire. At this level, it is possible to test discrete and emergent behavior with respect to commands that carry out very low-level actions. Wide test datasets will yield the best results, as long as it is efficient to test the outcomes.

The medium level of abstraction calls functions that perform moderately complex maneuvers or that embody behavioral models that operate based on a combination of parameterized data and sequential function calls, such as FollowPath, SelectRandomEnemy, SelectTarget, and so on. Here, it is also necessary to test any adaptive systems and combinations of rules that are designed to yield A-Life–style behavior. This includes all the generating AI, any GA or GP implementations (mechanisms that should already have been bottom-up tested), and so on. The test datasets at the medium level of abstraction need only be as large as necessary to test behavior within the fairly rigid confines of the desired behavior. This can employ generated A-Life algorithms to derive new behavioral patterns (such as GA or GP); after all, the bottom-up testing has already ensured that the individual rules and behaviors work correctly.

Finally, at the highest level of abstraction, sets of parameter data are provided, and nothing more. This should allow the game to run its course without explicitly altering the calling sequence. At this stage in the development process, the test process is concerned less with actual behavior and more with balancing the system so that it plays correctly.

Of course, the system will be also capable of selecting the calling sequence itself, based on the current state of play. This is what makes it a game, rather than an elaborate, interactive movie. Games built around AI and A-Life building blocks will have even more flexibility in this area of self-determination, within which the parameters need to be fine-tuned during the high-level testing phase.

In all cases, the game state must be continually monitored to check for discrepancies, and it will be advantageous if the design allows for monitoring automatically. This "debug" functionality will necessarily be removed for the final build, but it is useful to have it defined with these AI and A-Life testing principles in mind from the outset.

Ideally, what is being mapped and what is being tested should be linked. This is not as easy as making a vague statement of intent. It is necessary to be able to identify what should be tested in the same way as it is necessary to define the mappings.

Identifying What to Test

So we have decided how to map the behavior, based on the kind of AI and/or A-Life that has been implemented. The next step is to identify exactly what we are testing with each set of test cases. The previous discussion detailed how to map the inputs and outputs, but did not explain how to determine what to test.

If it is not possible to measure what is being tested, then it is not possible to test effectively. Even the mapping from the game environment or engine to values that can be interpreted does not help if the measurements cannot be analyzed. By a similar token, if it is not possible to identify (and quantify) what is being tested, then it is impossible to validate the result. Without criteria for the identification of issues, the tests themselves are meaningless, beyond simply exercising the system and making sure it does not crash. This in itself is a useful exercise, but not the only kind of test that needs to be run.

Final stage beta testing, for example, will consist of tests that are designed to throw up issues. These tests might be nothing more than an instruction for the play tester to play the game and note anomalies. But, crucially, the play tester has to know what the game is supposed to look like, so that he can recognize those anomalies and note them. Automated play testing might not be able to do this, so it relies more heavily on system boundaries and strict pass/fail criteria.

The assumption is that the boundaries that represent the game environment and the limits of free movement are known in advance. This will be the mapping section restricted by game rules, the extent of the environment and/or behavior, and so on. Referring back to our shooter example, if a dataset is provided that represents the movement algorithms for the flocking groups of enemy craft that break from the static formation, we can check for a variety of positional data elements that will tell us if any of those rules have been violated. In this case, the empirical output items (possibly mapped) are obvious. They will be based on coordinates within the gaming environment, collision data, and so on.

It is also worth restating at this point that since often the on-screen display is simply a virtual reflection of the in-game state data, a lot of testing can be done without any of the screen display being in place. Position data is easy enough; after

all, the extent of allowable movement is confined by the boundaries of the game environment (or screen). There are, however, other areas that will require a little more ingenuity to test. These might even require the mapping of the game state to a suitable AI routine in order to check for problems. In the end, as long as a state can be identified in which the game environment is at fault, it can be tested for. Part of the solution is in the basic measure-and-verify approach described, and part comes from the next section: "Testing A-Life with A-Life."

TESTING A-LIFE WITH A-LIFE

The final part of this chapter part deals with using A-Life to test games that include A-Life as well as AI mechanisms. This is a dual-purpose exercise—a way both to validate the A-Life that will actually be used in the game and also as a shortcut to play-testing by using A-Life avatars in place of human alpha testers. However, this kind of testing is a rare, latter-stage testing methodology that can only take place after the other parts are tested. Unlike many of the preceding examples, this kind of A-Life testing really does have to be left to the end of the production cycle.

This also means that the tests will no longer be bottom up, but system wide (or even top down) in nature. The tests will cover the system as a whole and will be done once all the other parts, besides the A-Life, are in place.

There is a good reason for this. Unless every other component has been correctly tested, any errors (or less desirable behavior patterns) that are found may stem from somewhere other than the A-Life itself. Due to the strong emergent behavior of A-Life mechanisms, it might be impossible to know which component has caused the problem. Therefore, it is necessary to be 100 percent sure that the input behaviors of the units that make up the system are correct. That way, by a process of elimination, it is possible to be reasonably certain that it is the combination of these units, together with any AI or A-Life, that has caused errors. Thus, the game can be rebalanced with new information, safe in the knowledge that the underlying algorithms are sound.

One advantage of this methodology is that it can be mixed with other testing paradigms, such as using human play-testers. They can even be used in the same play session or game environment and at the same time (as we shall see later on in Chapter 9). Because A-Life techniques process data from the game engine (and environment), it makes them a good fit when mixed with other key testing paradigms.

Most of the previously mentioned key points also apply here:

- Knowing the boundaries.
- Mapping the behavior.
- Identifying what to test.

Since the A-Life algorithms process information from the game engine (in the same way that AI built into the game does), it is still necessary to make sure that high-level mapping, at the very least, has been done for the behavior and observed results. Whether these observations are used as triggers (or alarms), or whether it is possible to identify exact measurements that can be compared with expected results, it is still necessary to go through the same steps as with other forms of testing.

The caveats also hold true here, especially those relating to the relative increase in complexity. Using A-Life is necessarily a complex undertaking that involves all of the techniques covered so far, and designers/developers must take this into account when deciding to use A-Life testing techniques.

This is doubly true for those games that are not using A-Life techniques. Games in which a strict rule-based system is deployed and where units within the game have no autonomy (one of the key determiners for AI) will require resources to produce a A-Life result that has a single purpose: testing. The trick in using these techniques efficiently is to turn the A-Life around and use it to test itself. So if a driving game is being developed in which drivers have the autonomy to make their own decisions, evolve as drivers within certain parameters, and so on, these can all be reused in A-Life testing. The algorithms for driving, decision-making, crash avoidance, and everything that a human player would need to play the game have already been created for each autonomous driver. All that is needed is to follow the previous three steps to determine the boundaries, mapping, and tests and then connect the already existing AI to the player interface.

Then, using A-Life techniques, the testers can "breed" or "evolve" different kinds of drivers, all with different personalities, and see how they fare within the confines of the game. In doing so, the test team will also see how the other drivers (which will make up the competition for the real game) will fare against each other and a variety of different simulated players. The additional cost is minimal because the existing code is just being reused (provided the design has been approached with this use in mind). This is another reason why the deployment of A-Life (and AI) in video game development *and testing* is a design decision—not purely a developmental one.

Again, the bottom-up testing must be done to ensure that the actual mechanisms work. All that the A-Life algorithms need to test are the applications of the gameplay mechanisms and the ways that they interact with the static parts of the game environment.

Learning by Example

One of the most lifelike applications of AI is to give an in-game entity the ability to learn by example—through its own actions and/or by observing others. We saw this in previous chapters, from the ability of the player to teach a monster in *Black&White* through direct interaction with the beasts to the experiments of the Norns in *Creatures*. In the testing process, AI can be leveraged by combining it with A-Life techniques in interesting and productive ways. The first requirement is that the behavior be mapped in such a way that it can recognize patterns. Without this it becomes very difficult (if not impossible) to implement learning.

The second requirement is that there must be some form of reinforcement mechanism that can be manipulated. This is vital because it is necessary to selectively store patterns of behavior, depending on the last requirement, and the measured success or failure. Again, we have the three pillars: boundaries, mapping, and measuring (knowing what to test). When these three requirements are met, it should be possible to copy the in-game behavior of players (alpha play-testers) and

then alter it, using A-Life principles. This technique holds for action games as much as it does for strategy games and has as its goal the replacement of random behavior generation, combination, and evolution (previously suggested) with behavior based on actual in-game events.

If this approach is used during play-testing, AI can observe the player and use his behavioral patterns (at a variety of levels) against him and gauge its effectiveness. Then, A-Life techniques are deployed to alter the most or least successful approaches, based on the functionality provided for in the game design.

Scripted games and those based on GA or GP techniques will be able to make good use of the data, while those deploying digital genetics of other kinds will also benefit. For games that only allow the fine-tuning of parameters that serve as offsets to generic behavior, it is somewhat easier to deploy this A-Life testing technique because the input and output data are less complex, but it is unlikely that the resulting technology will be able to be reused within the game.

As before, it is necessary in the game design to allow for hooks that can be used to "observe" and record the actions of players and other in-game entities. Then, there must be a way to enact these actions (and similar ones) within the game environment. This takes us right back to mapping again, once more proving its necessity in the testing process.

This will be taken one step further in Chapter 9 when we look at multiplayer A-Life, where the ability to mimic players becomes more than a testing paradigm; it becomes an integral part of the game design. As we shall see, this leads to other interesting effects, some of which can change the balance of the actual game (and should be used with caution, as will be discussed soon under "Adaptive A-Life"). In single-player environments, however, this technique is still very valuable because it becomes possible to alter and replay behavior according to A-Life principles of digital genetics, which can be used as part of the testing process.

Video game testing requires that we test the *mechanics* and also the *balance* of the game—the way that it plays and the look and feel. The record-alter-replay sequence defined here is designed to help test this second aspect of testing: the balance of the game.

Other techniques concentrate on the mechanics, such as pseudorandom testing coupled with AI and brute-force algorithms designed to widen the test data to the point that each rule has been put into every conceivable state (during bottom-up testing). While A-Life can also be used to test the mechanics, it is a far more powerful way to make sure that the game remains playable under a variety of different strategies and that no one strategy gives an unfair advantage. This can be achieved with extensive play-testing, too, but if we combine the two, just as many test cases can be covered with less actual play-testing by (expensive) humans. In other words, by observing human play-testers and their interactions and then using that as a basis to create test cases using A-Life techniques, we can potentially reduce the amount of play-testing that will be needed by cloning testers, subtly changing them, instilling the evolved versions into virtual copies of the game environment, and then measuring the results.

Taken one step further, these datasets of recorded and altered patterns of behavior can then be coupled with the various rules that manage other A-Life aspects, such as flocking, squad play, and so on. The parameters that govern these "other" aspects can then be changed while keeping the parameters of the simulated player static, and the results recorded.

Again, altering the various parameters of the game environment can be done using techniques from the previous section, which are mixed with the A-Life techniques presented here. So if a game records a play-testing session in which a player successfully negotiates a given level by using behavioral patterns that have been recorded—not just the path through the level, but the way that the player dealt with obstacles along the way—and this recording, when replayed, was successful, that would provide a basis for altering other aspects of the game in an attempt to foil the recorded behavior. This way, the test team, as well as the designers and developers, will learn a lot about the various rules that have been implemented and the ways in which they can be adapted to provide an enhanced experience for the player.

In the previous driving game example, this becomes even easier, since it is easy to map the rather limited set of in-game actions to a neural network (or network of FSMs) that can be trained to execute the act of driving in certain ways. The drivers can be made to learn and adapt their techniques, based on their experiences. This can then be deployed in the game itself; but of course, we still need to test this kind of adaptive A-Life.

Adaptive A-Life

Adaptive A-Life is a slightly different spin on the previous learning-by-example technique. In fact, the focus is not so much on copying what testers, players, or other in-game entities are doing, but is on being able to detect and change (adapt) rules that govern the game environment, autonomously. The problem is that having changed these rules, there is a need to check that the end result is still appropriate within the boundaries of the game. Breeding or evolving rules in this way is potentially a dangerous extension of the techniques presented in this book.

However, we know what we want to test and have a scale by which to measure the results. Part of the requirements of the A-Life testing approach is to be able to map and test behavior against some nominal boundaries that represent an abstract allowable deviation from set behavior. In other words, the designers must know what sequence of events or conditions is considered to be successful and/or represent allowable behavior. Crashing into walls in a driving game might not be allowable behavior, for example. This is something that it is possible to test for, and assuming it is possible to discern which adjustment caused the crash, it can be counteracted.

Based on this kind of in-game knowledge, it is therefore possible to adapt the rules if failure is encountered. It should also be possible to define a system that is self-regulating in cases where AI and A-Life are extensively deployed. This can help counteract some of the problems that occur in games where the end result might be somewhat unpredictable.

This is all an extension of A-Life, where techniques usually reserved for creating artificial behavior patterns are deployed to adapt in-game rules as a result of play-testing. The play-testing can be automatic (using AI/A-Life, pseudorandom testing, or dataset creation) or use human play-testers, as before. If the game is using A-Life and adaptive reasoning systems as part of its design, then this is also a logical step in the testing process. All the rules and mechanisms have to be tested, and the best way to do this would appear to be with A-Life techniques.

By keeping one part (the player avatar) static and playing repeatedly in different modes against the adaptive system, it is possible to check that the adaptive rules are operating correctly. This was previously seen in the "Testing AI with A-Life" section, as well as in "Learning by Example," but is only one part of the adaptive approach. The other part is to change the actual in-game rules using A-Life techniques. Then, the way in which the system reacts to these changes can be measured. Using the same test data, it is possible to exercise all aspects of the system and check that the emergent behavior is still within the boundaries set by the game design.

This is an extension of the "Pseudorandom Testing" section. It creates a dataset that represents various (unpredictable) player behavioral modes and then replays the test dataset over differing rules and sees that the system does not break in the process.

SUMMARY

This chapter has concentrated on deploying AI and A-Life techniques as part of the unit testing process. Hopefully, you will now be aware that many of the principles also apply to other aspects of the design and development process. In fact, this is often true to the point that the very code that is deployed in the game can be used to test it. *However, this assumes that the bottom-up philosophy has been applied, and that the mechanisms have been proven to be correct*.

Care also needs to be taken to make sure that the process is designed to test the correct aspects of the system. Due to the way in which A-Life and AI work, this is not always obvious, especially in strongly emergent systems. Therefore, as in design/development, testing should be from the bottom up, rather than pulling the system together, deploying it, and hoping that it is possible to correct unwanted behavior during an extended testing cycle.

One of the key advantages of using A-Life is that it is possible to put the system in many more situations than is traditionally possible with straight testing that uses datasets created by humans or even generated by algorithms. Part of the natural deployment of A-Life is that it will produce unpredictable combinations of data; on top of this, it is possible to influence the frequency in which this happens by changing the way that the data is allowed to evolve.

In addition, A-Life can be used to breed bots (scripted entities). Some will be good, others will be bad, but all will exercise the engine and game environment in a multitude of ways. Again, we can have quite a lot of control over the kinds of scripts generated. The actual testing can be approached in a number of ways:

- Datasets representing in-game settings.
- Datasets representing scheduling of behavioral patterns.
- Datasets representing behavior.
- Adaptive rules governing boundaries of behavioral aspects (system).
- Adaptive rules governing behavior (probably based on FSMs).

Each of these needs to be represented in such a way that AI or A-Life principles can be applied to them. This makes it possible to create variations—preferably enough to catch every possible combination of in-game situations. If this is not enough, copying and adapting observed behavior can extend the dataset still further.

This is not quite the same as trying to breed good gameplay rules (an application of A-Life that we discussed in Chapters 4 and 5). It is a way to try and check that any in-game rules, adaptive systems, A-Life, or other parts of the system are working correctly. Some of the adaptive code might be reused, and any such reuse is to be welcomed. We might also leverage other system components as part of the testing process. Again, testing a game component twice is often better that testing it just once.

Not only is test coverage better when using these techniques, it can be more efficient, provided there is a tried-and-tested way of gauging success (or at least spotting failure/breakdowns). This efficiency is enhanced by the mixture of traditional (human) testing with A-Life principles. It might even be possible to replace some of the early alpha play-testing with A-Life avatars, which means that more can be tested quicker and earlier. This is coupled with the fact that virtual players do not need the graphical interface, since they can interface directly with the game engine.

In short, better testing makes better games in the same way that better AI makes better games. AI and A-Life make for better testing, which should help ensure that the game's overall quality is improved, along with its balance.

However, unless these mechanisms are anticipated in the design phase, it will be too late to deploy AI and A-Life in the testing process once development gets to a certain point. Therefore, as in the game proper, deploying A-Life for testing must be done in a bottom-up fashion.

There are actually three key areas in testing video games:

1. Unit (AI and A-Life) testing;
2. System behavior testing for A-Life;
3. A-Life in video game testing.

This book has dealt with the first of these and advocated a bottom-up approach almost exclusively. This rigid adherence to bottom-up testing can only go so far, however, and the system-wide testing and general use of A-Life in video game testing have to adopt a top-down approach as far as their deployment goes.

(Of course, the constructs used to effect the testing will themselves be tested in a bottom-up fashion.)

As we mentioned, there will be a certain overlap in the test cases used for testing the system behavior of an AI- or A-Life-based game. This will usually need to be performed in conjunction with human play testing, in a combination of bottom-up (automated) and top-down (human) testing approaches.

Throughout this chapter, we have also hinted at the wider use of A-Life and AI in general video game testing. That topic alone could fill an entire book and needs to be addressed in a top-down fashion. Individual A-Life constructs can be deployed to test virtually any video game and achieve tireless, repeatable test coverage of a system more effectively than human play testers.

AI can be used to analyze the results, and while the combination of A-Life and AI cannot replace good old-fashioned play testing, it can go a long way toward reducing the need for expensive and prolonged testing and replace part of the human test cycles.

References

[LECKY01] Guy W. Lecky-Thompson, "Pseudorandom Testing." *Dr. Dobbs Journal*, August 2004.

CHAPTER

8 SEVERAL A-LIFE EXAMPLES

In This Chapter

- Movement and Interaction
- A-Life Control Systems
- A-Life in Puzzles, Board Games, Simulations
- Video Game Personalities

The goal of this chapter is to provide examples for using A-Life enhancements at several levels—from simple A-Life add-ins to games in which the A-Life *is* the game, as in *Creatures*. It's also time for some sweeping definitions (and hopefully, you scientists out there will excuse any oversimplification of the difference between artificial intelligence and artificial life).

In essence, video game AI replicates human thought processes, using rules to dictate behavior. A-Life enables the AI system to exhibit lifelike behavioral patterns. Arguably, every game does this to some extent as a result of the weak/strong emergence that takes place when preset rules are followed en masse. You need AI (or at least AI mechanisms) to design A-Life, but the application of AI alone is possible, though it might not yield very lifelike results.

The example often cited thus far is the "perfect driver syndrome." AI allows the game designer/developer to theoretically imbue an in-game driver with 100 percent perfect skills. This driver will never crash and will usually win every race. It is imbued with the perfect algorithm for positioning, starting, and handling the corners, while keeping the perfect racing line, and so forth. That's not very realistic or lifelike. Even though some A-Life behavioral features will arise, thanks to the power of emergence (like traffic queues), when these drivers are pitted against each other, they are too clinical and perfect to be considered lifelike.

Our A-Life counterbalance is to add some unpredictability to the mix—emotions, perhaps, or simulated tiredness, frustration, mechanical failure, or inconsistencies and, if all else fails, outright errors of judgment. These are the A-Life add-ins: things that can be added to the AI to make it more lifelike. In this case, our add-ins dampen the perfection with which drivers handle the track. Using them implies that the model simulating the in-game avatar (a driver, in this case) is slightly richer than need be. A certain complexity is added, which turns a simple rule-following AI mechanism into something with personality—with life.

Each of these A-Life add-ins makes a difference to the game and the way that it is played. Take away the A-Life facets, and you probably still have a game, but it will be less so. Any simulation of real-world experiences or "assumed-world" experiences that have a root in human experience will benefit from A-Life add-ins. They become more than just add-ins is when they begin to affect the way that the game is played. When removing the A-Life add-ins from the game changes the very nature of that game, this promotes the add-ins to something more important. However, it is important that the add-ins do not become crutches for an otherwise less-than-inspiring title.

Despite the other problems that *S.C.A.R.* has as a driving game, the nod toward driver personality and its implementation applaud *S.C.A.R.* as the first "driving RPG" that makes for a slightly more lifelike experience; facets of the drivers' personalities make them prone to certain kinds of errors. Some can be easily pressured into making bad decisions, such as by tailgating them at high speed. Others are more aggressive and do the same to the player (whose simulated heartbeat rises until his vision blurs and the game becomes harder to control), which adds to the racing experience. Take these away and the game is playable but ordinary—a racing game with no special structure or features. The add-ins have enhanced the

game, and (arguably) their inclusion means that they walk that fine line between enhancing the game and propping it up.

The next level up is where the A-Life really makes the game. *Creatures* is the oft-cited example in this book, as is *Black&White*. *SimCity* can also be put in this category, although its rule-following, underlying structure might place it more in the AI camp. *The Sims*, however, falls squarely in the A-Life category, if only because of the finely tuned interactions between the Sims themselves. So here are the three main levels:

- Add-ins
- Enhancements
- Core features

These levels of A-Life inclusion obviously have varying consequence inside the game; A-Life add-ins will have less of an effect on the way that the game plays than core features. On the other hand, these levels of A-Life in the video game are in no way related to the complexity of the algorithm. A simple follow-the-leader flocking algorithm, for example, can be used for controlling the enemy flight patterns in a shoot-em-up. Without the flocking algorithm, it is perfectly possible to control the individual enemy craft, but using flocking offers options that exceed the rule-based control. Each unit has an autonomy beyond the central control system—autonomy that extends the flexibility of the underlying game without adding significant complexity.

In fact, to control large numbers of entities, the required processing power and control-system complexity are less than if a control system had to be built for each and every unit. The flocking algorithm, as we shall see, lends itself to object-oriented programming techniques that lend themselves toward autonomy, rather than central control.

The rest of this chapter is dedicated to demonstrating how (rule-based) AI can be combined with (natural) A-Life algorithms to create useful and interesting effects for video games. The algorithms are only the tip of the iceberg; it is, in the end, the game designer's imagination that will take these techniques further.

Where appropriate, these techniques will reference games that have appeared in issues of *EDGE* magazine (2007–2008), but there are many examples of where fairly advanced and safe AI is being deployed—and also where AI is being augmented and improved by the addition of A-Life principles like those covered in this book. Our examples illustrate a cross-section of the current state of the art in game technology and are representative of games scheduled for release between 2008 and 2010. Bearing this in mind, we open with a quote from Yoshinori Ono, producer of *Street Fighter IV*, which embodies our philosophy for balance in video game A-Life:

"From the creative point of view, it's kind of easier to make a game that allows you to win, but it's very difficult to make a game that would give you a reason for losing and make you want to challenge it again." [EDGE01]

It is the illusion of a victory that is just out of reach, coupled with a variety of different possible outcomes, that makes the player believe that next time he might

be successful, and this enhances the game. If the AI and A-Life do not provide this enhancement, then the game is better off without it.

Movement and Interaction

Creating lifelike movement is quite a difficult proposition that consists of two aspects—animation/appearance and the path of the entity (or its individual parts). They have to work together. Unlike Michael Jackson's moonwalk, artificial life will encounter conflicts between movement and path; they need to be completely congruous when interacting with the environment. And movement needs to give some visual clues as to what the entity is capable of and the way that it is likely to move. This allows the player to anticipate what is about to take place, such as in the following discussion about *Street Fighter IV*.

> *"Ryu bobs up and down slowly, methodically, in a definite and regular rhythm; Ken, on the other hand, has a more hyperactive style of movement, bounces on the balls of his feet when standing still…" [EDGE02]*

If either the animation or path is lacking or contradicting, or if either is out of sync with the entity, then the resulting movement will not seem lifelike at all. So movement is important. In addition, there is an emergent interaction between units that suggests life; for example, ants will work together to achieve a common goal, even if each one is working purely on instinct that dictates its action. This contrast has been previously noted, such as in Chapter 5 ("Building Blocks"). There is an aspect of A-Life that can be seen as being based on instinct, as opposed to reasoning, like AI is—and as such marks the difference between state engines and their application.

A state engine is like the instinct mechanism that drives an ant. The application of these state engines in A-Life is like building a colony of digital ants that produce an artificial abstraction of the real (naturally occurring) life. A-Life is the embodiment of natural life in artificial form, and like real life, the emergence of the system can be exploited to remove any reliance on reasoning (AI) systems that usually underpin such mechanisms.

This digital instinct can be modeled as state machines and deployed as an emergent system, devoid of reasoning, yet capable of lifelike behavior.

However, the line that separates the two is somewhat fuzzy. Pathfinding is a good example: Is it instinct or reasoning? Finding a path through a maze is clearly an example of reasoning (problem solving), while walking across a room and avoiding furniture is based more or less on instinct. For now, we will divide AI and digital instinct into two categories: non-reasoning (state engines) and reasoning (decision trees, neural networks and expert systems). In the video game world, at least, this separation ought to be sufficient in order to determine what constitutes artificial life and how it can augment straightforward AI.

Movement and interaction, for example, are two areas in which A-Life and instinct have more to do with implementation than AI and logic-based reasoning, at

least at the tactical level. At the strategic level, where there is no intended direct effect on the game universe (through the actual actions), clearly AI and reasoning systems will be used, even if they are implemented as state engines. However, learning and behavioral modification (part of the building blocks) can still be used. Indeed, to make the most of the artificial life paradigm, they must be used in order to reflect the entity's interaction with the environment. After all, walking through water and walking on sand will produce two very different variations in the "walking movement" behavioral model. And traversing a landscape littered with obstacles will yield very different paths for entities than if they attempt an open field. This shows two levels at which behavioral modification can be used to alter the decision process, and there are many more movement levels at which it can be applied.

The question for the designer is: where to stop? Where should the line be drawn between complex applications of artificial life and resorting to preprogrammed, concrete action/reaction algorithms and animations? The answer depends on the game, genre, and processing resources available. Hopefully, an analysis of a few different kinds of movement-behavior algorithms will help in making this decision. We will look at three types of behavior that are synonymous with movement/interaction models. (Animation is covered toward the end of this chapter.)

- Flocking (and swarming)
- Follow the leader (an extension of flocking/swarming)
- Squad play (extension/specialization of follow the leader)

Flocking is strictly an instinct-based algorithm—the ability of entities to arrange themselves with respect to their nearest peers while maintaining some form of forward movement. By itself (with no sense of direction), the resulting pattern will be either static (nothing moving) or fairly random (depending on the variations introduced). The entities can also be constrained (or contained) for a specific purpose, such as maneuvering them through an opening or containing a crowd in an arena. However, the results are usually best when mixed with the second behavioral algorithm.

Follow the leader is a mix of reasoning and instinct; instinct keeps the entities together (flocking), but they are following a leader that is imbued with directional reasoning. This could be received reasoning—that is, the system or player decides the direction—or some kind of innate AI that determines the course of action.

Finally, squad play takes follow the leader one stage further by implementing an explicit command-obey interaction link. This works across entities, but by itself would only offer rudimentary flexibility. It is also possible to bestow the individual entities in the squad with both responsibilities for certain tasks as well as some behavioral AI to help them carry these tasks out. In addition, squad members should have responsibilities toward each other. In a military setting, for example, the entities might be honor bound to cover each other when under fire. These responsibilities could also form part of their individual characters, with some less likely to stick their necks out than others, but the underlying mechanism is the same.

This is an example of behavioral modification at work. All of the entities might have the same basic premise for controlling their actions in a given situation, but if

they all reacted in the same way, then it would not be very lifelike. Adding behavioral modification to the mix allows the algorithm to be applied in a fashion that varies the input vectors and results in different behaviors among entities. But here is the important point: If, for example, there are 10 men in a squad with 10 different personalities, there are *not* 10 different behavioral models; the inner digital DNA of each man dictates how the single behavioral model that is programmed is applied.

Underlying all movement patterns is the ability to flock, be it among mobile or fixed entities. If we do not determine how an entity fits into the environment and reacts to its peers and enemies, lifelike movement will not be realistic or possible. Note that here we have left out pathfinding and reasoning-based algorithms. (They have been covered elsewhere, such as in Chapter 5.) Here we deal with the artificial life that manages interactions with the environment at the lower level, not the strategic niceties that are handled at a higher level.

Flocking

The basic flocking modeling algorithm is based on three key principles—separation (avoidance), alignment (average heading of peers), and cohesion (steering toward neighbors). Beyond this, there are the various attributes of an entity, such as acceleration/deceleration, turning circle, and minimum/maximum velocities. These are coupled with:

- The desire of the entity to travel in a given direction,
- Current information, such as speed, energy, and stamina, and
- Any AI that governs how it is to move with relation to these (for example, tired entities might slow down to conserve energy).

All of this information contributes to the algorithm's calculation of an entity's position in space at

```
t + 1
```

(where t is an arbitrary unit of game time). This calculation and the subsequent relocation of the entity in relation to its peers are the result of the flocking algorithm.

The most famous example, perhaps, is illustrated by Craig Reynold's boid behavior (1986), which was the result of research on flocking and is quite lifelike to watch as they flow around a virtual tank. The flocking movement is modeled on birds, but other creatures, such as fish, also exhibit this kind of behavior.

A modern implementation of the Boids algorithm can be found at http://www.dcs.shef.ac.uk/~paul/publications/boids/index.html.

The Java applet allows the user to change all the various aspects of flocking that are discussed in this book, along with a few other refinements that make for an interesting simulation.

Flocking can be used for a variety of applications in video game design. Some are fairly obvious—such as battling entities in virtual wars, waves of marauding aliens, or the attractive eye-candy of a properly modeled flock of in-game birds or aircraft. There are also applications in driving games, such as to keep computer-controlled

cars from crashing into each other, and any tactical games that require realistic computer-controlled animations of tight formations. This makes flocking useful, for example, in the control of crowds walking down a busy street (contained flocking), with the container being the narrow corridor that is the sidewalk, bound by a road on one side and buildings on the other. This is also an example of each entity's desire to move in a certain direction while being obliged to not bump into peers.

The same principle can be applied to modeling for direction-finding in racing games, for example. Each car will attempt to move in a given direction (perhaps itself controlled by flocking within a narrow corridor that represents the racetrack), but will also need to take account many other factors in play when other vehicles are encountered.

This puts flocking into two distinct areas: On the one hand, it can be applied with respect to multiple mobile entities (as in a flock of birds). On the other hand, flocking can be used for obstacle avoidance against static elements. The advantage is clear. The same algorithm can be deployed with multiple behavioral variations that change the way it is applied. The result for designers, developers, and the processor is less work—and better results. Before we look at some of the more esoteric, advanced, and experimental uses for flocking, we need to understand some of the basics.

Movement Calculations

First, it is necessary to look at the two sets of factors that affect the new location of an entity—the basic movement values and the three principle flocking criteria: separation, alignment, and cohesion. (The actual calculations will be dealt with later on.) Figure 8.1 shows a collection of entities, each with a little arrow that indicates its current heading. Separation is indicated by the straight double arrows that interconnect the entities and their immediate neighbors. The bent arrows indicate their cohesion (but are shown for only a few entities for clarity). In Figure 8.1, all of the entities are aligned; their arrows show that they are all trying to move in the same direction.

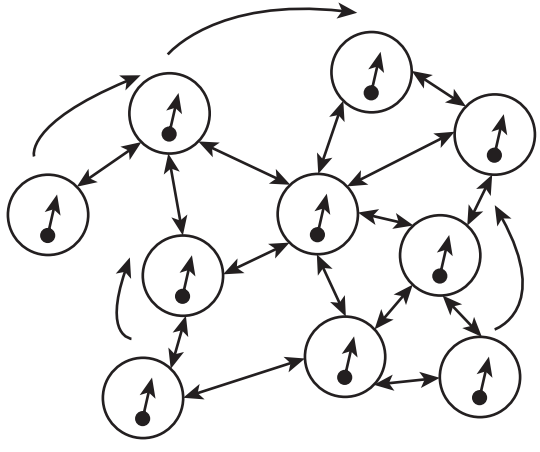

FIGURE 8.1 Separation, alignment, and cohesion.

For a given entity at time `t` and having the factors acceleration, deceleration, turning radius (representing maximum values, except where indicated), and minimum/maximum velocities, it is necessary to know at what pace the entity is moving, its heading, and if maximum or minimum velocity has been reached. Exactly what happens then will depend on the entity and the medium through which it is moving.

The entity's new position in space can be calculated by first applying the acceleration/deceleration to obtain the new velocity (taking into account the minimum/maximum velocities) and then calculating the position in terms of the angular velocity (at the current heading) using basic trigonometry. This will result in the desired position in space, everything else being equal.

This calculation does not take into account the position of the neighboring entities, however, so they need to be factored in through *influence*, rather than direct modification. In other words, each of the three flocking principles affects the desired velocity (through acceleration/deceleration) and heading, which have already been calculated.

For each entity that is within a certain sphere of influence, the new position has to be adjusted in relation to its peers within the same sphere, according to the separation, alignment, and cohesion factors. Each of calculations will alter the acceleration/deceleration and angular velocity (due to an alteration of the heading).

It is the emergent effect of all of these interacting values that gives rise to the flocking behavior. Without them, the entities would either move around at random or just plow through each other, following their own agendas. Note that this algorithm does not calculate the positions and then adjust them according to the separation, alignment, and cohesion factors. Instead, it modifies the movement algorithm through the application of the previously mentioned factors.

This approach can be refined a little with the inclusion of random variations so that each entity does not react entirely predictably. The resulting movement will be more lifelike and varied; after all, birds rarely just follow each other blindly. However, the main refinements will be in the way that the key separation, alignment, and cohesion factors are applied, and these can be subject to behavioral modification matrices that will change the resulting individual behaviors of the entities without overcomplicating the calculation algorithm.

Separation

Put simply, separation is the measure by which entities in the collection try to avoid touching each other. In fact, it is a factor that is used to determine whether or not an entity is correctly positioned with respect to its immediate peers and pulls the entity in several different directions at once in an effort to maintain constant spacing with other entities surrounding it. However, this is not a discrete measurement. It is more like a rubber band value in reverse. Imagine a rubber band of set length that holds two entities together. If they constantly try to pull away, then the rubber band would let them do so, but only up to the point at which it would pull them back together.

This is how separation in flocking ought to work. In other words, there is a smooth repelling action between entities. Otherwise, it would not result in very

lifelike behavior. Entities do not space themselves homogeneously within a flock; they move around inside a variable "envelope." Some entities wait until the last moment to move away from others, and other entities change position more gradually. To manage this artificially, there are some governing values that control the amount by which the entities are repelled, and these affect their movements in a given direction. It is these values that can be set, entity by entity, to provide the behavioral modification that leads to artificial life.

This is rather like giving the entity a personality. The various values are set in the digital "DNA" of the flocking entity and govern their behavior from when they are created until the time at which they are no longer needed or are killed off. Therefore, between these two times, their behavior remains the same and cannot be altered without some emulation of *training* or simply learning.

Two functions that can possibly be applied are the exponential/logarithmic functions that offer ways to develop a smooth-scaling repellent factor. Rather than using a straight-line relationship, these curved functions allow us to simulate the "panic versus foresight syndrome."

If x varies between 0 and 1, then the exponential function provides a higher repellent factor the closer the value is to 1 (between 1 and 2.5 times a given repellent factor). This can be called the "panic syndrome." For the same variance, the logarithmic function starts off with a repellent factor tending toward –1 (actually, –1.3 for an x value of 0.05), and the closer the value approaches 1, the closer the logarithmic value approaches 0. To use the logarithmic value, it can be multiplied by –1, which yields a practical range of values. This can be called the "foresight syndrome," because the assumption is that the sooner the entity takes action, the less action it will need to take later on. Of course, this does not preclude the possibility that something will move into its path and disrupt the model, at which point the governing state engine might choose to apply the "panic syndrome" factor, instead.

Both of these functions follow curves, rather than straight lines, which makes them more realistic for use in a separation algorithm, as noted previously. Various plotted values were used to construct Figure 8.2.

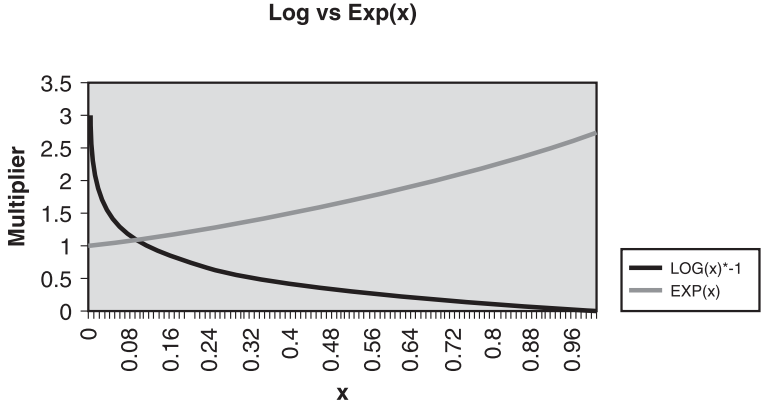

FIGURE 8.2 Log versus exponential (*x*) for 0➔ 1.

Figure 8.2 shows a plot of Log(x) * –1 and Exp(x) for values of x between 0 and 1. Clearly, the resulting values might not be usable in their raw forms, but it is the shape of the curve that is exploited, not necessarily the actual values.

If used as separation factors, the Log(x) * –1 function provides more repelling force the closer together the entities are, while the Exp(x) function provides a smoothly decreasing repelling force the closer together the two entities are. This follows the same "panic versus foresight syndrome."

Now, it is also possible to change the shape of the curve in order to bestow entities with different characters—a basic, but effective, form of behavioral modification. For example, if a section of a sine curve is used, it is possible to model a scaling factor that follows the typical sine wave; it starts off with a low value, rises to a peak, and then drops again. It might even go negative, following the curve, if that is the behavioral model that is required.

Any function that produces a curve from a scale of values (sine, cosine, and tangent might prove usable) can be deployed in place of Log or Exp to produce different variations of scaling factors. These can then be fed the separation factor between an entity and a nearby peer to derive the repulsion factor needed to satisfy the separation value that is part of the entity's "personality."

Clearly, the actual separation value is also important. Some entities might choose to travel closer to their peers than others, while other entities might tend toward constantly trying to get farther away. This all enhances the model that produces a passable imitation of life.

The separation factor can be applied to (subtracted from) the (desired) heading to turn the entity away from the neighboring entity and also applied to the velocity to slow the entity down or speed it up. Separation is the hardest to apply in terms of working out its effect and then applying that effect on the movement algorithm. Alignment, in contrast, is the easiest.

Alignment

Assuming that forward motion is directional—that is, the entity can steer itself in a given direction while continuing generally in a nominal forward direction—alignment is the ability to alter the direction, based on the entity's nearest neighbors' average headings. The exact definition of "nearest neighbors" will have a substantial effect on the result.

Admittedly, line of sight rules should produce the most lifelike effect. If an entity cannot see the other entity, how can it possibly know in which direction the other entity is heading? However, a more realistic algorithm, in terms of implementation, might be to take all entities within a given circle (sphere of influence) around the entity in question. This is the first factor that can comprise part of the alignment behavioral modification matrix. Entities take note of the directional headings of their peers within a wide/narrow sphere of influence.

While a response function could be applied to the alignment, it is questionable as to whether this is really worthwhile, given that the inverse square rule will dictate

that the further away an entity is from the source of the effect, the less the effect will be felt as it is passed on from one entity to another. This emergent behavior is due to the constant averaging out of directional data propagated throughout the flock.

Just like light emanating from a single point, as the alignment change ripples throughout the flock, it is absorbed by individuals that average it with the alignment of their peers, thus diluting the effect. This natural effect means that it is possible to directly apply the result of the averaging in each case without needing to alter it in any way.

The only adjustment that might make sense is to weight the effect on the average according to the distance between the entity and the referenced peer. Therefore, it would naturally align itself more with those immediately surrounding it than those that are farther away.

The alignment factor can be applied to the heading, altering it to bring it in line with the newly calculated average value. Logic dictates that this should probably be done before the separation or cohesion calculations. It is only the combination of alignment and cohesion that keeps the entire flock headed in the right direction, albeit with local variations.

Cohesion

Cohesion is the technique of steering the entity toward its nearest neighbors. It is not really the opposite of separation (which tries to keep a consistent spacing), but has an attracting force that can cancel out the repellant force of the separation function under certain circumstances. As long as the thresholds are different, separation and cohesion will not cancel each other out entirely, and the effect will be lifelike. If they were to cancel each other out, then this would not be possible. Similarly, if the separation function results in an adjustment that moves the entities closer together, and this is then compounded by cohesion, then the effect might be undesirable. One way to guard against this is to use an exponential function for the cohesion effect and an inverted (multiplied by –1) logarithmic function for the separation effect. This means that entities close together will steer toward each other gently but be repelled strongly, thus preventing collisions.

Again, the cohesion factor works on the heading and (possibly) the velocity of the entity in an effort to move the entity closer to those surrounding it. The emergent behavior that ensues is enough to keep the flock moving more or less as one unit, with (as before) some local variations (birds flying off, returning, darting left and right).

Clearly, the combination of separation and cohesion dictates the factor by which the flock is dispersed—that is, the average distance between entities and the spread of the flock—and the alignment dictates the general direction of the flock as a whole. Another way to use the result of the cohesion calculation is to apply it to the result of the separation algorithm, rather than directly to the heading or velocity. In this way, the effect of moving away or toward other entities can be mitigated or enhanced by the result of attempting to steer toward a specific entity.

Those entities on the group's fringes will be kept close to the pack, and those in the middle will be pushed and pulled in a variety of directions, thus keeping them flowing in the right direction. Of course, the various factors that dictate the extent to which these algorithms are applied will directly affect the dispersion of the flock—and, by extension, the size of the "cloud" of entities.

Combination

Flocking principles are not usually applied alone. For example, applying only the alignment algorithm might lead to collisions and will yield a result where, although the entities do flow, they do so in an artificial, robotic fashion. Without the separation or cohesion factors, it is impossible to introduce local variations to increase realism. So it is necessary to apply two or more factors, unless other directional algorithms are also being used, such as for models other than flocking. For example, a driving-simulation engine might not apply all the factors all the time, because there will be instances when the cars are singularly following the track, using the racing line for guidance.

Also, all of the chosen forces act on all entities in the flock at the same time, so calculating their positions becomes something of an algorithmic choice in itself. The application of the direction- and velocity-adjustment algorithms needs to be translated to entity movement from one position to another and take into account all of the other entities. There are three possible options:

1. Calculate new positions sequentially, allowing the effect to cumulate;
2. Calculate the new positions from existing positions as new locations, thereby always holding two positions per entity (now and next);
3. Calculate new positions from old, choosing entities at random, otherwise the same as option 1.

The disadvantage of the first approach ought to be obvious; there is a ripple effect, which means that some entities that are handled early in the cycle will be treated differently. This is because the effects will accumulate as entities' new positions are used as the basis for calculations in other entities' new positions. In other words, the point of reference for entities that are treated later on in the cycle changes due to the recalculation of the peer entities. This effect can be mitigated by using option 3 or, equally well, by providing an additional, random-placement function. This will have the effect of mixing up the neighbors, so that two neighboring entities in space are not neighboring abstractions in the list of entities at the time of positional calculation.

For example, imagine that the entities are represented by an array of 100 elements, which are then populated with positional information such that neighboring entities in the flock are also neighboring entities in the array. The trickle-down effect of recalculation through the array will tend to mean that those near the end of the array have a different set of calculations applied to them, because they will be based on elements from earlier in the array that have not been subject to this cumulative effect.

Now, rather than assigning the coordinates sequentially, consider them assigned with a random spread, or that the evaluation of the effect of the algorithms is carried out at random (picking elements at random positions); then this problematic effect can be mitigated. However, the approach is not perfect. In fact, option 2 could be considered to be the best because it uses existing data for all entities and calculates new positions based on existing (current) positions and not possible (next) positions. This requires holding two copies of the flock in memory, which may or may not be convenient.

Static versus Moving Flock Targets

In the preceding discussion, there was no rule stating that flock targets have to be the same type of entity as those entities from which they take their references. It is a given that a flock of virtual birds will consist of a collection of entities that are all birds, but it is possible to introduce other points of reference.

Consider what happens if one of the entities fails to respect some of the flocking rules. What will happen? Hopefully, the entities that encounter this rogue element will exhibit enough emergent behavior (following the flocking rules) to flow around the misbehaving entity. The flock will still follow its rules, even when one of the members does not.

This is because we hope that the entity that is misbehaving has less of an effect on an individual than the cumulative effect of its peers. There will be a tipping point, of course, beyond which chaos will emerge in the system, as the number of errant entities outweighs the well-behaved ones, and we just have to hope that some kind of order will re-establish itself.

These reference entities could even be static points that do not move, but provide guidance for the flocking algorithm. They could be entities of the same type, or they could be entities that can communicate with, or at least be sensed by, the members of the flock (such that the flocking algorithm can be applied).

Static points refer to entities that do not move and that signal something to be avoided. This in turn leads to local variations as the flock moves around them. Advanced applications could also introduce variations in the entity's behavior, such as low cohesion and high separation factors, which will affect the behavioral model—an example of environmental behavioral modification. This means that the flock can also be steered using single points with high separation factors or strings of points that all repel the flock (or alternatively, try to drag it along a specific route). The channeling of the flock provides a way to manipulate groups of objects—though not necessarily a big flock of birds or school of fish—or small collections of entities.

So now we have a very powerful and easy-to-implement dynamic flocking algorithm that can be reused for other purposes. Since the algorithm has been designed, tested, implemented, and debugged and is based on understood and tried-and-tested artificial-life algorithms, the impact of using A-Life in the game will be reduced. For example, a racing game might use constrained flocking by employing points at the inside and outside of the track, with different combinations of separation, alignment, and cohesion to help steer a vehicle. After all, the game will probably use some kind

of collision-avoidance flocking algorithm, and leveraging it to avoid collisions with barriers makes design sense. Since the static reference points that represent the corridor of movement do not move, the vehicle will itself be forced to move as it approaches the designated points in the track.

Finally, an aspect of simulation might be needed to allow the AI that controls the vehicle to do some forward planning (see the sections "Movement and Interaction" and "Follow the Leader") and provide a virtual flock leader that the vehicle will follow around the track. Effectively, the vehicle becomes a very aware flocking entity—modeled with respect to the engine, tires, brakes, road surface, and so forth—that is constantly trying to follow invisible reference entities that are both static and moving.

All of these components—individual separation, alignment, and cohesion values and the entities (static, moving, and the peer entities)—come together and produce strongly emergent behavior for which the outcome will be difficult to trace back to the causes. And because the algorithms being used are based on easily adjusted factors, genetic algorithm techniques can be applied to change the "personality" of the vehicle.

This allows the designer to take the model one step further and adapt the personality of the flocking entity by varying its separation, alignment, and cohesion values, as well as velocity, reaction time, and other simulation elements. Any game that can use flocking (or flocking-style) movement algorithms, as described, can also modify their effect by using GA techniques.

Using GA to Modify Basic Algorithms

In order to appreciate how this works in practice, we are going to simplify the approach somewhat by keeping the factors that we discuss both vague and to a minimum. The actual movement of an entity needs to be modeled based on its innate capabilities and properties, following the design of the game. Simulation will be left out of our discussion. Instead, we will concentrate on picking data values for separation, alignment, and cohesion to illustrate the application of genetic algorithms and artificial life. This technique can be applied to other algorithms, not just flocking, but flocking is an easy example to understand and grasp.

The key is to understand that the GA does not modify the underlying algorithm, but changes the context in which it is applied. In terms of the flocking algorithm, this equates to giving each rendition of algorithm a slightly different character. There are two possible approaches:

1. Picking a "character" value that has as a reference a collection of behavioral offsets; or
2. Allowing the genetic algorithm mechanism to pick the behavioral offsets directly and not specify a "character" explicitly.

In both cases, genetic algorithms can be used to extrapolate the behavioral aspects of the entity within the flock or provide meaningful variations on a specific theme—such as an aggressive or slow flock, or languid or jittery movement. This is the core of the genetic algorithm mechanism. In the first case, however, the designer

provides a collection of named sets of individual values for separation, alignment, and cohesion. These are then used to populate an initial set of possible input data for the flocking algorithm.

This dataset can then be used to breed successive and related, but differing, datasets, using standard genetic algorithm techniques, such as crossover and mutation. Applying this theory could produce something as simple as a carbon copy of the initial dataset, and each of the values can be tweaked by a random amount (simple mutation). From here, there are many mechanisms that can be used to generate a collection of parameter sets. For example, the parents could be combined in three different ways to produce three children, of which one is dropped. This requires that each generation yield an odd number of children per two parents, of which two might be strict crossover and the third a random crossover and mutation. When we apply this technique repeatedly, a suitable population of individuals can be built up.

More variance in characteristics can be achieved in several ways. For example, the system might ramp up the mutation and/or crossover vectors so that all the genetic data that is exchanged is mutated by a small or proportionally large amount. This amount can be chosen via a random-selection or algorithmic (such as logarithmic) function. In this way, the resulting characteristics can be quite closely controlled and predicted (within certain bounds).

We can also expand the number of values that make up the genetic data. In our example so far, there are only three, but other pieces of input data are used in calculating the new position of an entity in the flock, such as picking values for acceleration, deceleration, turning radius, and maximum velocity. Applications might include parameters that make sense in other gaming environments—for example, RPGs might also need to include parameters like strength.

The maximum acceleration and deceleration values will affect how swiftly the entity can react to requests made to avoid, align, or stick with surrounding entities. Of course, a specific reaction-time setting might also change the way the entity will cope with these requests.

The turning radius limiter will also play a part in calculating the entity's new position in space. A tighter turning radius can also make the entity more maneuverable and therefore more capable of reacting to the separation, alignment, and cohesion factors.

In creating the flock, then, values can be picked that create subtle variations in capabilities, giving two aspects to the entity's character: behavioral desires and behavioral (physical) capabilities. Both sets of data can be manipulated using genetic algorithm mechanisms, producing a digital genetic string of about eight factors that can be used to create offspring (using the previously discussed techniques of crossover and mutation).

Left to its own devices, of course, the flock will not go anywhere, because it has no instructions for anything other than the status quo. Individual entities might swirl around, and the flock might drift aimlessly as a result of the general forward movement in a given direction, but there will be nothing purposeful about the movement. To make the flock move in a meaningful way, at least one member

needs to have more knowledge than the others—that is, the leader. It is the leader's job is to direct the flock's movement, but as we shall see, there are several ways in which the follow-the-leader mechanism can be applied.

Follow the Leader

The follow-the-leader mechanism is not a new one. The effect has been included in many video games over the years—from racing games to *Centipede*-style games, where segments of a digital insect follow its head. These also allow for renomination of the leader, which splits the *Centipede* into multiple segments.

Other games exhibit pseudo follow-the-leader mechanisms for lifelike "wandering the streets" character animation. These are not true examples, because the crowds are not a flock, per se, although they do need some flocking characteristics to avoid collisions.

The follow-the-leader algorithm can therefore be combined with flocking under certain circumstances, such as in the previous racing example. Each vehicle has a virtual leader that it follows around the track and is bestowed with AI to allow it to plan maneuvers (such as passing). Each vehicle also avoids barriers (composed of static flocking elements) and other vehicles (moving flocking elements) while avoiding crashing and manipulates acceleration, velocity, and so on. All of this could, theoretically, be delegated to algorithms derived from the flocking algorithm, as long as the vehicle-modeling aspects are taken care of. The result should be a fluid, natural racing model; but it will require tuning and AI support to ensure that everything goes according to plan. All that the flocking and follow-the-leader algorithms enable is the ability to choose a dynamic path through the circuit.

The result takes the instinctive forward-only motion and tempers it with steering mechanisms akin to opposing magnetic forces or fluid dynamics. As the car moves around the track, it will avoid barriers and other vehicles, using the flocking algorithm rather than some kind of robotic decision system. The result ought to lead to a more fluid, natural racing model.

The actual follow-the-leader algorithm is very straightforward and only requires that a sphere of influence be created around a given entity in the flock. Like the sphere of influence encountered in the previous discussion on separation, alignment, and cohesion, it is an area within which flocking units will have an effect on each other. Whereas before they only tried to align themselves with each other, now additional weight is given to one particular entity, regardless of the distance from its peers. That entity is the leader, and all the entities within the follow-the-leader sphere of influence will try to align themselves to the direction of the leader.

To apply this, we must adjust the alignment of each of the affected entities so that they orient themselves with the leader, as well as take into account the general heading of their peers. The effects of separation cannot be ignored, either; this prevents collisions, and cohesion will keep the flock together.

As the leader changes direction, so does the flock within its sphere of influence. The rest of the flock will take its cue from these initial "early adopters," and the change in direction will propagate from the edges of the initial sphere out toward

the other flock members. The direction of those outside the sphere of influence will then tend toward those closer to the center of the sphere of influence. Subsequently, the whole flock should gradually move in a given direction—dictated by the leader.

This can all happen in the blink of an eye, however, if one thinks of a shoal of fish turning as one to avoid a predator. Clearly, the notion of *gradually* is subjective. The principle still remain the same—instinctive responses just happen more quickly than reasoned ones. The fish have input from many sources—their peers and their own senses, for example—all feeding into their simple, instinctive, immediate reaction.

The larger the sphere of influence, the more quickly the flock is likely to respond. This might transcend line-of-sight or nearest-neighbor-style mechanisms designed to communicate changes from the leader to other flock members. Small changes by the leader, such as obstacle avoidance, might not have a trickle-down effect on the rest of the flock, but a larger change in direction (such as following a path) ought to. This allows the leader to operate as an entity in its own right, while also relying on the traditional flocking mechanisms to manage the rest of the flock when obstacles, corridors, or other variations in the game environment are encountered.

Changing the Leader

The selection of the leader is, of course, entirely transient. Except in certain circumstances, flocks can have dynamically allocated leaders. In squad play, for example, the leader could take the form of one of the squad members (a squad being a small flock) who shouts "Over here!" and runs off down a corridor. The rest will tend to follow him, and this is equivalent to changing the leader.

If the leader changes, then there might be a moment of subtle chaos as the flock reestablishes itself. The larger the flock, the more potential there is for chaos (due to emergence) before the effect of the renomination establishes itself through the members, and the flock takes off in the new direction. This can be observed in real-life flocks, such as birds; the more frequently the leader changes, the more chaotic the flock will act.

In some cases, it can be quite effective to change the leader among those that are closest to a nominal head of the flock. This can create a more fluid effect as the tide of individuals moves toward a common direction. As leaders are picked from the nominal front of the movement, the flock will tend to disperse, and as long as those leaders are constantly renominated, the flock will remain dispersed. These transient leaders should all be based on the same AI algorithm that guides their choices regarding general direction.

Once a single leader holds sway longer than any others, then the flock will tend to funnel as the effect of the alignment toward a single common direction (as dictated by the leader) lessens among entities that are farther from the sphere of influence. Since the leader is the flock member that knows the desired direction best, it will pull slightly away as the effect of its velocity and direction are averaged with the "flock noise" when its peers calculate their own velocities and direction vectors. Therefore, the input received from the leader becomes less powerful the farther away an entity is from the source. If each entity takes the average of the alignment

values of its peers, the alignment of the leader quickly becomes contaminated. Of course, this effect can be mitigated by adding more weight to the leader's input.

If the leader and the common direction change simultaneously—for example, the new leader heads off in a different direction—then the flock can be redirected. This will tend to be exhibited as the flock pulling gently toward the new direction (led by the newly nominated leader), rather than the whole pack suddenly performing a U-turn. Therefore, a more natural, fluid change of direction is effected.

Multiple Leaders

If there can be a change of leader, then it stands to reason that there can also be multiple leaders within a flock. Whether this is a good idea or not will depend on the game being implemented, but it is certainly possible. If a flock does have multiple leaders, it can cause the flock to separate or flocking elements to interchange. Two previously disparate flocks might come together to form a single flock and then separate again, each mini-flock following its own leader.

This effect can be exploited by using a system of focus and filtering, whereby individuals only allow the signals of other individuals of the same group to influence their alignment value. Naturally, this allegiance can also be transient, with units swapping flocks according to a variety of in-game variables. In addition, the units must also make sure that they do not collide with entities that are not part of their specific group, and so the separation value needs to be retained. In this way, entities are taking some of their (friendly) input from their own flock and some (foe) input from entities outside of the flock.

To do this, a simple neural network might be necessary to weight the various algorithms' impact so that the resulting position/velocity change (acceleration or deceleration) is a result of all possible inputs. This allows some of the weights to be more urgent than others, thanks to the use of logarithmic and exponential functions. This is a reasonably complex operation that enables dynamic flexibility in the basic flocking algorithm and can be used to provide behavioral modification to individual units. It might not be practical for large numbers of units in a flock, but for certain controlling units or small flocks (squads), this might be a useful mechanism. From this it follows that a flock can comprise a mixed collection of entities: dumb units that just adhere to the flocking algorithm, some moderately intelligent units that provide local control and communicate with other mid-level units, and one or more leader units that provide direction to the flock.

Whether used for flocks of birds or as the mechanism for controlling large numbers of units in, for example, a war simulation game, the reuse level for a tried-and-tested algorithm is high. This reuse is possible by altering a few properties, making a known good algorithm applicable in different situations.

The Player as the Leader

A last interesting point is illustrated in the game *Burnout Paradise*. Traditional track layouts are eschewed in favor of an open-environment racecourse:

"…doesn't even have an endpoint beyond a ticking clock, your opponents scattering and converging around whatever route you choose." [EDGE03]

The key to this dynamic, as described in *EDGE* magazine, is that the player chooses the best route through a city. In this case, the best means the quickest. The other cars then follow that route, racing against each other and the player toward the finish line. This makes the player the leader, and the rest of the racing entities (the flock) take their cues from the player, rather than sticking to a traditional course. This is a good example of dynamic behavioral modification that uses player input to feed an existing, implemented, well understood, and reliable game mechanic.

In cases where the leaders are followed by the whole population and not just their nearest neighbors, the result is more akin to squad play. Part of this requires that there will be some element of communication between elements in the squad. This is especially the case when leaders are communicating with the others and the multitude is simultaneously trying to listen to the leaders independently. To avoid confusion, there had better be some effective controlling AI behind the artificial life synthesis.

Squad Play

Squad play can be considered a special kind of flocking with the same elements of separation, alignment, and cohesion—but it is an extension of the basic premise, which models instinctive behavior artificially. There needs to be some element of reasoning built in to the underlying mechanism. Applying squad play requires the player to be constantly monitored, and the squad reacts intelligently to what the player does. This might also mean that the system AI learns to anticipate what the player is about to do, based on experience.

The squad-play mechanism has a use in multiplayer games where not all squad members will be controlled by humans, for example. The human players will expect the behavior of the other members of the squad to react correctly and realistically. Again, observation is a good technique, but behavior that is generated by it has to be tempered by the behavioral limits of good practice within the system. Observed behavior cannot just be applied to units under the system AI's control without some reference to the rules of the game. Just because a human player breaks a rule or acts in a way contrary to the spirit of the game (for example, shooting a member of his own squad), this does not mean that the system should try to replicate this unacceptable behavior in its own in-game representatives.

The other possibility, of course, is to instill the opposing squad with flocking/follow-the-leader algorithms in order to give rise to an effective opposition. Of course, this must be combined with AI pathfinding algorithms, which might further be augmented with A-Life to allow for some variations. The other layer of the squad algorithm needs to include communication between squad members. The end result is a combination of effects—from explicit communication to observed and copied patterns of movement, to a kind of instinctive movement algorithm in which the squad travels as a single unit.

The application of these algorithms can be as simple as their intended in-game use, based on the tried-and-tested formula of player versus the game. Each squad member has his logic for movement and grouping:

> *"…block and dodge they do, exhibiting levels of intelligence, and the dev team made a point of the fact that enemies attach together, rather than falling back on the game cliché of advancing one by one to their death." [EDGE04]*

This is undoubtedly a step up from the usual wave upon wave of zombies that present themselves, either one at a time or *en masse*. If we add AI to the mix and allow a certain awareness of location to creep in, and if not correctly tested, some unfortunate side effects can creep in that the player can exploit:

> *"It's too easy to exploit the enemy AI, through sniping, for instance: Enemies will run to where fellow foes were killed, ready to be sniped themselves." [EDGE05]*

So blind copycat flocking in squad play is not something that should be encouraged—or at least, the situation might be solved with implicit communication; the foe is dead, and something has killed him, so there is no need to rush over and get killed as well.

There is, however, a balancing issue here. Soldiers will often run to help a wounded comrade. It may take a few hits before they realize that there is a sniper around, and they should stop rushing over into the line of fire. Someone might recognize it and shout, "Sniper," thereby warning the others. This would be a realistic and lifelike mechanism to address the above.

The illustration here is that balancing and flocking have to go hand in hand and, once again, tune the game toward having fun, while keeping in the realism.

Finally, communication will lead to actions or reactions. Whether it is implicit (observed) or an explicit *communiqué* from one unit to the other, this needs to be handled in a way that is both intelligent and lifelike. Much of the preceding section dealt with using a lifelike algorithm (flocking), but remember that there is an AI side to it, as well.

In fact, communication can be handled much like the follow-the-leader algorithm. There is a leader that dictates where the squad moves—this is true whether the player is the leader or whether the opponent is the leader—and those commands have to be translated to in-game position play, according to the available algorithms.

A good example is *Advance Wars 2: Black Hole Rising* for the Game Boy Advance, in which the player can direct events, but the individual units move around with apparent lives of their own. Of course, this movement is necessarily simplistic, but the underlying mechanism is there.

In cases that have prospective multiple leaders, such as a number of units vying for attention, care must be taken to provide an algorithm that handles the selection of the nominated leader. We have discussed selection algorithms earlier in this book, all of which use the same technology as neural networks for selecting the "noisiest" entity. The criteria used for the selection are in the form of a weighted

network, the evaluation of which yields the identity of the unit that should be selected (which, for example, ought to be the one with the most authority, successful track record, or forcefulness).

A-Life Control Systems

Artificial life algorithms can also be used to create systems that define interactions between the system and in-game entities, as well as interactions between entities. It is assumed that the video game developer has already created the simulation aspect of the system and is looking for alternative ways to deploy it. For example, in a driving game with the framework in place (including the AI that drives the cars and the model of their physical interaction between the wheels and the road), the game will work, but lifelike behavior will add to the experience. That is, we can introduce additional variations to the in-game properties that control the way that the game plays.

This can be extended to the kind of A-Life found in games such as Clear Crown Studio's *SpriteLife* and *WildLife*, as well as *Black&White* or Maxis' *SimCity* and *The Sims*. These use the same technique of interplay—differently weighted physical and behavioral factors, but more in-game resources dedicated to the AI and artificial life. Some of those resources will be dedicated to modeling individuals, like in *WildLife* (Clear Crown Studios, 2005). Here, each animal is governed by its individual characteristics, which are selected by the player. In the case of *WildLife*, this is accomplished with sliding scales that govern each animal's looks, needs, and likes. When set up, the animals are modeled as in-game automata. They have lifelike qualities, such as thirst and need water, hunger and need food, and so on. Their success does not appear to be connected to their appearance, however; it is linked to the attentiveness of the human player.

More recently, *Spore* (created by Will Wright) uses very advanced A-Life gaming systems. Players can customize their individual entities before introducing them into various environments. Initial demonstrations and interviews with the game's creators suggest that in this game, it seems that the "design" of the creatures has a direct effect on their success at surviving in their virtual environment. However, the balance appears to be in favor of the player—there are very few penalties for badly designed automata. The game progresses from cellular automata to tribes, civilizations, and eventually space flight and exploration, all the while offering the player a rich tapestry to choose from—but with very few downsides to bad decisions. Is the end result more than just a collection of different interactions between scripted entities with differing capabilities, or does the game embody some of the philosophy behind *Creatures* or *Black&White*? Or is this technique just a pretty design tool? It could be a whole lot more. The driving force, however, remains the same. Under the hood, the same principles are at work as in other Sim-style games.

First Principles

First, the designer needs to decide what principle factors affect the control systems in the game mechanics. In driving games, for example, it is clear that the interaction is in terms of turning the wheel, pressing the brake, pushing the accelerator, and so on. Games that have a direct correlation between the player's actions and in-game effects in this way will be reasonably easy to abstract as a set of factors that affect control and success. In other simulation games, however, it will be much more difficult to pin down exactly what factors affect the control system, especially in life simulations that take into account aspects such as food and energy.

Furthermore, most game types can adapt A-Life principles to their control systems. Even shooting games have aspects of the control system that can be manipulated using A-Life principles to make them more lifelike. Examples include weapon reload times, hesitation, shaky aiming, and other real-life, human characteristics. These can all be applied to the player character as well as nonplayer characters to make the experience, via the control system, more lifelike and immersive.

It is, therefore, a design decision as to which parts of the control system will be subjected to A-Life modification—and above all, whether these modifications will have an effect on the difficulty balance of the game. This leads to further possible refinements, such as whether these factors should then be accordingly modified during the play session as a way to adjust the balance dynamically, depending on player experience.

Difficulty Balancing and Variation

The practice of balancing/varying difficulty is not new in video games. Most games have some form of dynamic way to adjust the parameters to constrain the player's experience and elicit ever better performances from him. For example:

> *"Since the difficulty level increases whenever the ending screen is displayed, better accuracy is required and less time is allowed." [EDGE06]*

In the same game, the developers have also made an attempt at providing variation to an otherwise mundane example of the genre:

> *"…although the sight of the same enemies jumping out from behind the same trees at the same point in the game may grow tiresome by the 15th or 20th time, somehow it's an altogether different experience if you know that, next time, they might be wearing panda suits." [EDGE07]*

This comment might be a little tongue in cheek, but it does illustrate that even a game that has limited possibilities can sometimes be enhanced with a some creativity. Advanced technique can also serve to counteract the tedium that constant boss battles in a shooting game can bring about:

> *"Do bosses have to seem impossible and then prove tedious when their patterns have been learned?" [EDGE08]*

Here again, it is clear that behavioral variation will enhance a game. The resources needed (compared to the game as a whole) are also not particularly great. For example, Yoshinori Ono, creator of *Street Fighter IV*, indicates that a simple in-game mechanism that addresses the balance in favor of the weaker player can reap great rewards:

"…more obvious things [that] we're doing like the revenge gauge—if you're taking a lot of hits then the revenge meter will fill up quickly and let you strike back powerfully." [EDGE09]

The in-game moves are the same, but the underlying mechanism changes their damage quotients. The dynamic alteration of one of the key factors in the *Street Fighter* game mechanic (damage) illustrates the use of these techniques in real games. However, this is clearly a double-edged sword; adaptive behavior in an A-Life–enhanced version of the game might take advantage of the computer learning—such as holding back and taking damage—and use it to build up a ferocious revenge attack. This leads to the question of balance. As Yoshinori Ono continues:

"That gives beginners a chance to hurt someone that's more skillful."

Clearly, if the less-skillful opponent turns out to be the computer, then the human player is going to suffer as the computer holds back, takes damage, and then builds up an attack that the player cannot stop. Badly balanced, that might make the game unwinnable, which would be an example of a bad application of the underlying technology.

Implementing A-Life Control Systems

The following examples are taken from common gaming genres and show where AI and artificial life can be used to enhance the playing experience. It is important to remember that these examples are only illustrative and are not intended as complete descriptions of games. They are intended to illustrate points that show the utility of the techniques discussed in this chapter. The genres chosen are fairly broad in scope to provide the best possible coverage:

- Vehicle racing
- Sports simulation (soccer)
- First-person-shooter
- Role-playing game

In each case, the games are examined from the point of view of the player character and the observed behavior of nonplayer characters during the play session. These NPCs may be under the player's control, opposing the player, or autonomous teammates, depending on the genre and game environment. Each example illustrates a specific control system that can be modified using A-Life principles of GA and GP. In addition, examples have been chosen in which the collection of rule-based systems and other AI will lead to emergence—and hence, reasonably intelligent behavior.

In the next section, we extend this discussion to break down the actual GA and GP paradigms. In doing so, we will tackle, using concrete examples, simple puzzle games that illustrate AI, reasoning, memory, and behavioral modification. Here, the examples are more complex, but the basic underlying principles will be the same. In effect, we will look at what is possible and then move on to how it might be achieved.

Vehicle Racing Games

As previously discussed, vehicle movement is governed by two sets of factors: direct vehicle control by the driver (human or virtual) and the reaction of the game environment (simulation) on the vehicle. The latter forms the simulation part of the game and provides vital information to the player about the expected behavior of his vehicle.

These factors that affect vehicle movement need to be broken down into two sets of data. The exact combination of individual data units is beyond scope of this discussion. However, they can be seen as similar to the entity modeling used for the flock—acceleration/deceleration, heading (turning), and so on. In addition, there are physical factors like friction (or lack of it), which might lead to skidding. These are, in turn, functions of various properties of the vehicle. Friction, for example, is a compilation of factors such as the vehicle's weight, the wear on its tires, any drag from various spoilers, and so on. The interaction of these two sets of factors is what makes the simulation part of the game; without each being in place, the game ceases to be lifelike.

The factors can be manipulated to change various aspects of the model. Take friction, for example: As the car burns fuel, its weight decreases, which will have an effect on its friction against the road. Depending on the accuracy of the simulation model, the player will have to compensate for this weight loss. Other physical effects, such as tire wear, will have a similar effect on the balance of the vehicle model.

Now, some of this might be out of reach for an arcade-style racer, but for more serious simulations, they become a necessity. In addition, many games allow the player to choose the level of sophistication that he wants the vehicular model to exhibit—starting with fairly simple things like automatic gears, up to more complex effects such as custom tires and driving aids such as automatic traction control. When added to the complexity of designing different vehicle models with different handling, it becomes clear that there will be a lot of data manipulation required. This helps the implementation of artificial life, however; the more data that can be isolated to describe behavior, the more it can be toyed with to create different effects.

Player Control

First, it is important to note that the vehicle characteristics will have a direct effect on gameplay—that is, different cars will have different handling possibilities, both dynamic and static. The static properties, which tend to be set at the start of the race (the vehicle's life), include things like maximum acceleration (limited by the engine), speed, braking capabilities, and drag from spoilers. They might even be open to customization by the player. The dynamic properties, which can change during the play session, might include the effects of weight or tire wear. In addition, there might be

the effect of crash damage and so forth. All of these can be modeled in terms of their effects on the principal factors that weigh into the simulation of a moving car.

Second, the car can be altered subtly by changing the way that the player is able to interact with the game. This includes specifying characteristics that allow the player to have a direct effect on the vehicle's performance. For example, the speed at which the virtual steering wheel can be turned might be different from one vehicle to another. This will affect the turning radius, which might be helped by power steering but hindered by flat tires. The net result is that the player has to learn the capabilities of his car before racing it.

Obviously, the combination of these three sets of control factors in the system produces the end behavior. A small change in any of them will potentially yield a larger change in the final experience, due to the emergence resulting from the repetitive application of rules using these data. In addition, changing them will produce subtle variations that are "lifelike" or, in this case, "realistic." Take, for example, the various engine tweaks that are available in games like *Gran Turismo* and *Need for Speed*. The player might be able to opt for lighter car bodies, improved tires, or various other customizations.

For additional realism, the control system (or its effect) can be modified by environmental factors and random events, such as road quality, crashes, or mechanical problems with the vehicle. This can be done dynamically, of course, by a controlling AI system that just tweaks individual factors, either temporarily or permanently. It is important to remember that for each of the three sets of factors—static vehicle properties, dynamic vehicle properties, and control and environmental properties—the same underlying technique can be used: The behavior of the vehicle can be altered just by changing the core factors that affect the way it is modeled.

It is the job of the AI to figure out which factors must be tweaked and by how much. It is the job of the designers to make sure that a workable subset of possible factors is available. A game like *Gran Turismo 4* is going to use a lot more of these. The vehicle component of a game from another genre, like *Halo*, will use fewer.

Observed Behavior

On the system side, the computer is controlling all the nonplayer vehicles, as well. While they should all drive in a sensible, controlled manner, they should also have certain characteristics and ways of reacting to the environment that are lifelike. This makes it possible for the player to anticipate what each NPC will do in a given situation, while allowing the vehicles to exhibit personalities that make them more lifelike. The emphasis here is on "controlled unpredictability" that feels natural, rather than applying faultless driving models.

Anything that does not enhance the player's experience is a waste of time and effort, so a fine line exists between obsession with realism and providing quality entertainment. In addition, if the player feels that the opponents are unrealistically well equipped or talented, he may (even unconsciously) begin to feel that he is wasting his time.

Many of these behavioral tweaks are available as A-Life modifications to the control systems. This reduces the impact on the developer, as it reuses existing, well-understood properties that go into the production of every racing game. For example, if the designer creates an ideal racing line around the track for all competing vehicles to follow, behavioral variations can be introduced that allow deviations from that ideal line. The power of emergence will cause other rules to be evaluated differently—such as braking, acceleration, or steering patterns—which will produce further behavioral variations.

This ideal line can be determined either through coding a series of static follow-the-leader-style points or through observations received by a gifted driver, or encoded in another way. The first two have the advantage of lending themselves to manipulation through A-Life mechanisms.

In addition, the combination of flocking with interactions between the vehicles will have an effect on each vehicle's progress around the track and the way they interact. The basic flocking algorithm can also be modified to allow drivers to have distinct characters—more aggressive ones driving closer to their opponents, for example. This last is not a theoretical pipe dream; it has been implemented in the racing-RPG game *S.C.A.R.* Other examples of behavioral modification lead to a less-than-perfect racing line, such as in *Gran Turismo*; opponents *seem* to react to the skills of the player by occasionally mistiming corners and slamming into barriers. (Though it seems that the worse the player drives, the worse the opposition does, too. And the better the player drives, the less chance he has of winning the race, unless he has a supremely powerful car.)

In addition to this observed behavior, it is also possible to modify the basic control factors, such as the mechanical characteristics (capabilities) of the vehicle and the decisions made by its driver. Again, these characteristics can be modeled by simple manipulation of the direct control systems. To emulate slower reactions, just change the speed at which the steering wheel can be turned or alter the pressure that the brake pedal needs before it reacts. Changing these parameters can emulate the characters of the drivers, in addition to any adjustments made to the flocking algorithm that leads the driver to make certain decisions regarding his opponents. (We also mentioned that a traffic queue in a racing game is an example of emergent behavior. The application of the flocking algorithm will alter the likelihood of these queues emerging, because it will have a direct impact on the ability of a car to pass another; the farther behind the car becomes, the less likely it is to take the risk.)

The other artificial-life aspect is that genetic algorithms can be used to produce datasets that are varied in a sensible fashion (having been bred from the "perfect driver"). In addition, genetic programming can be used to change the driving-AI's decision process before an action is attempted.

There is a benefit to taking this approach. A varied collection of virtual drivers can be put together without having to encode each and every one of them. Many racing games need hundreds, if not thousands of opponents, and using GA and GP to create unique NPCs may well be the way of the future.

Soccer Simulations

There are two sides to soccer simulation—managerial and player-manager—but in reality, there are very few differences between the two. Each has aspects that need to be controlled, and players in each game have more or less the same set of vital statistics that govern the way that they perform on the pitch.

Much of our discussion can be applied to any sports simulation game that boils down to a set of statistics. The only difference between the manager and player-manager games is that, in player-manager games, the player can actually have a direct effect by taking control of one of his team members during the match, rather than just standing on the sidelines.

In a modern game such as *LMA Manager*, the game is played much as it would be in a player-manager or player style, the key difference being that there is no human-controlled player on the pitch. All of the players (both sides) are controlled by the computer. This means that a pure management soccer simulation is just a collection of observed behaviors that the player can alter by changing the static elements of control, while the dynamic elements are purely under the control of the computer. It is a game of statistics that, without artificial life techniques, would rapidly become predictable and clinically precise.

The static elements (simplified for our purposes here) might include player capabilities (skills) as well as tactical elements, such as formation. Some of these will be set by the designer (real-life soccer stars have some capabilities that all players are familiar with), and some are modified by the player. Dynamic elements are things like player energy levels, morale, and the various factors that affect the ways that players interact. These can be based on flocking algorithms or specific placement algorithms, depending on the situation.

This all sounds relatively simple. The reality is that there are many small interactions between skills and physical capabilities. In addition, many decisions must be made as to what to do next at a low level (pass, run, tackle, shoot) and higher levels (intercept, long pass, and so forth). The result is a very complex control system that needs good, solid traditional AI to keep the balance of play but some form of artificial life to keep the player interested. Many factors, from random events to the animation and manipulation of in-game avatars (representing the soccer players, managers, referees, fans, and so on), will contribute to the quality of the game. And due to the nature of AI and artificial-life systems, each piece of the jigsaw will contribute to the overall emergence of the system and have trickle-down effects. Some of these will be startling and nearly impossible to predict. Here is a simple example: A yellow card has an effect on the aggression of an individual player, and when coupled with a seemingly innocuous challenge, increases his frustration level, which in turn leads to a change in the way that he plays and the risks that he takes.

The Soccer Player's Control

We start with the assumption that during the game, the player may take control of one of several individual team members on the pitch. Each will have different capabilities and skill sets and will therefore "play" slightly differently. The control scheme might vary. Some games might adopt the direct-control method, whereby the controller scheme (joystick, keyboard, mouse) is mapped onto the player's capability model. This means that direct action by the human player will lead to discrete, in-game action on the part of the virtual soccer player (run, pass, shoot).

On the other hand, the indirect control method might have an interface that is geared toward the selection of players and hint at what they should do next. These hints could be like the football "plays" made famous by the *Madden* series. Each play is an indirect instruction for the team to perform a given series of actions.

The interaction with the opposition is left up to the computer, and the human player has very little chance to intervene. Therefore, the success of the play is geared toward whether or not the statistics play out. However, the *Madden* games also allow the player, if he wishes, to take control.

Both of these control systems mean that the in-game behavior of a virtual player will change when the human player takes control of it. Subsequently, the unselected team members' behavior will revert to the observed behavioral patterns discussed in the following section.

As in the virtual-racing model, artificial life principles can be used to modify the control system to simulate different aspects of the physical and logical capabilities of individual players. It is therefore possible, from a discrete collection of predetermined behaviors, to build unlimited random sets of players or give the designers the ability to define deviations from the standard player model by using properties abstracted from real-life players.

Some of these attributes will change over time and be modified by interactions with other players during matches. Again, it is the emergence of the entire system that gives rise to lifelike behavior, much of which depends on the effectiveness of the classic, rule-based AI that controls the game as it is played. This will be exhibited primarily in the observed behavior of the system.

Soccer Observed Behavior

First, the observed behavior takes the form of simple reactions to the human player's actions and moves. This will be managed by a rule-based AI, perhaps augmented by variable behavioral techniques to introduce variances. Second, the AI will follow and execute plans dictated by the player as part of a general strategy. This, combined with the individual characteristics of the players, will lead to naturally emergent behavior when taken into account with the reactive behavioral model. It will also reflect the virtual player's own skill set, which will drive its individual behavior.

Finally, using the principle of changing the leader in a follow-the-leader behavioral model, it ought to be possible to model the ebb and flow of the game. Clearly, the ball can also be considered an object with characteristics in its own right. All of this will drive the game forward.

Flocking and other AI/A-Life behavior modeling algorithms can be deployed to make sure that the general rules are respected, but there will likely also be many different schemes that the designer chooses to apply that are just as innovative. The key is to prevent monotony and predictability while remaining true to the capabilities of each player.

These suggestions are presented as guidelines, something to make you think. Clearly, video games will need more than these basic principles, but the driving, underlying A-Life behavioral algorithms deployed in a bottom-up fashion provide a starting point. The main hurdle is in avoiding the danger of the gameplay becoming just an application of statistics. Real-life soccer tournaments often include surprise results and upsets that, while they might not change the overall outcome, have the effect of making the game a more lifelike experience. Artificial life constructs can also be added to further enhance the game, such as player and fan interviews, newspaper reports, and so forth. Techniques to address these "perks" are dealt with in "Video Game Personalities," later on in this chapter.

First-Person-Shooters

Perhaps the easiest genre in which to appreciate general AI/A-Life, the FPS is also the most demanding on the processor. Not only is the play area high-graphics and high-sound effects intensive, but there are usually multiple complex entities that have to be rendered at the same time. This all places a certain load on the available processing capabilities.

Any AI and A-Life that are deployed must therefore be as processor thrifty as possible and not use CPU cycles needlessly. It seems that advances in processor power have always led to ever-increasing levels of realism on the graphics/audio front, with underlying AI and A-life mechanics lagging slightly behind.

Despite this, many games have scripted AI that allows the designer to be quite creative with his A-Life. For example, say we generating in-game scripts based on the A-Life techniques discussed in Chapter 5. Since the scripting mechanism is there anyway, it would be a shame not to try and exploit it by adding a spark of artificial life to build behavioral scripts with different behavioral patterns. There are also (in general) some immediate parameters that control behavior and that can be easily modified using A-Life algorithms. These tend to relate to things like physical characteristics, speed of movement, reaction time, and levels of guile and intelligence. In reality, a lot of behavioral-pattern modification can be done purely by altering a few parameters (GA) while leaving in place the main algorithms (scripts) that control the actual behavior. This is rather like the previously discussed racing simulation.

If we assume that the parameters govern the way that the algorithm is applied, then any restriction that changes the way that the algorithm is carried out will also have an effect on the final behavior, because these actions will feed forward into the rest of the process—simple emergence. A very easy example is to place a restriction on movement, which makes the opposing entity (a monster) slower to carry out a request. When the controlling AI routine (or scripted FSM) determines that a state

change should take place, it is possible to artificially slow the response time. This will make the monster act differently than one that has a faster reaction time.

Two identical FSMs that receive the same incoming triggers will exhibit different behavioral patterns. Because the reaction times restrict the actions that are carried out, even if the behavioral models are identical, there is a good chance that the sequence of events and state changes will also be affected.

This is the essence of genetic algorithm theory, and it can be equally well applied to other gaming genres. This technique is particularly good for action and FPS games because it has low overhead and high reuse. Each script can contain a section that defines parameters that can vary for "classes" of behavior, which can then be integrated with other scripts in a building-block fashion. Only the data is varied, not the algorithm itself, to produce behavioral changes. This leads to high reuse and low processing overhead—all using well-understood technology. Together, these qualities combine to make a low-impact solution, such as will be discussed in Chapter 10.

FPS Player's Control

A-Life can be used to modify the contact that the player has with his in-game character, much like a mix of the racing and soccer simulations. This contact will change the game's balance and the way it feels. This will comprise a mixture of mechanical and physical properties: simulated firing times, magazine changes, and random events (even such as simply dropping a magazine). These combine to make the game feel more real and also provide challenges for the player to overcome, depending on his observed skill level.

Never forget: If a model does not enhance the game, then it is a useless waste of CPU resources and can even damage the balance of play. If a player exhibits exceptional skill during gameplay, then the controlling AI can introduce variations to test his capabilities. On the other hand, these random variations should remain lifelike and in character; for example, a rookie might drop a magazine during reloading, while a professional won't, at least not as often. These kinds of random events will affect the way that the player plays the game.

Another example might be the "shakiness" of sniper aiming, which might change with an increase in experience as the player's character advances through the game. His aim might become less affected by stress under fire, for example. This is an application of the same algorithm, using different data—a type of artificial-life programming. There are many other options, especially when vehicles and weapons are used as props. The mechanical properties, for example, comprise the way that weapons react, as well as sound and graphics effects. There will be ample opportunity to vary the standard, generic models.

Most A-Life use will be in the observed behavior category—artificial life observed by the player as well as A-Life observed by the system, which is then replayed through entities. For example, in *The Club*, a first-person-shooter from Bizarre Creations (SEGA):

"Each of the game's eight different characters has a very distinct feel. Kuro's low-resilience/high-speed combination, for example, will require a very different approach to that required by Renwick's all-round balance." [EDGE10]

This clearly shows the effect that simple changes in the player's capabilities have on the overall balance of the game. By the same token, even simple control systems can be interfered with to introduce variations. For example, consider the following control scheme:

"During battles, the player simply points the right stick in the direction of the foe they'd like to attack.…Simply shifting the direction on the stick will change targets…" [EDGE11]

A variation on the above, using artificial life techniques, would be to make the speed and accuracy of the target change, depending on the character being controlled. This will add immersion and realism to a control technique that might seem clinical and unrealistic.

FPS Observed Behavior

As in racing games, the computer is usually in control of all the nonplayer entities. This means that the player observes their behavior from two different viewpoints—either as opponents (enemies to overcome/outwit) or as allies (friends to help out in an intelligent way). For example, the security guard in *Half-Life* is a friendly NPC that (as long as it's alive) faithfully follows the player's character around and helps out on occasion. The NPCs are there to help or hinder and in doing so could, for example, be tailored via scripting. The effect is that the player will notice that each NPC is slightly different. They might be based on similar AI behavioral patterns, but minor variances will make the game more interesting.

These variances can even be attached to physical appearance, something that games like *Spore* use in quite an advanced way to influence the capabilities of individual entities. The player quickly attaches meaning to the appearance and is able to more easily predict behavior from the outward appearance of an in-game entity.

The trick when implementing these behavioral algorithms is to concentrate on a low-impact solution—for example, change datasets, rather than try to make self-modifying code that swiftly alters itself on the fly. Making changes to weighted values in a neural network, or just changing a few-rule based parameters, is often sufficient. The result will be much the same, but the processing overhead is lower, making it perfect for use in an action game. The pace of the game is likely to be so fast and the environment in which it is being played so restrictive that the player will not notice the difference.

Both of the techniques will be covered further in Chapter 10, when the A-Life moves off the drawing board and onto the workbench and is introduced into some game design and development situations in a low-impact way. Recent applications of artificial life have been described in *EDGE*:

"Our character first shoots at a corpse to distract the creatures, which choose to feast on the blood rather than attack the player." [EDGE12]

Now, this example shows some (possibly preprogrammed) natural behavior that could be arrived at by balancing desire (for blood, in this case) against the need to kill the player's character. The observed behavior of the creature emerges as a result of the application of several facets of the creature's own attributes. This illustrates an important point about AI/artificial life in video games: It is sometimes more natural and easier to allow behavior to emerge, rather than try to preprogram neat tricks into the behavioral models. If artificial life has been part of the design strategy from early on, these kinds of happy coincidences will likely occur on a fairly regular basis. That is not to say that they cannot be predicted or even brought about, but the additive effect of multiple behavioral traits and rules can sometimes bring better behavior models than the slightly artificial application of layers of finite state machines.

Role-Playing Games

An RPG is often a balance between pure strategy and FPS-like action. In broad terms, they fill a gap in the market, being built on a solid story with multiple characters and can involve interaction like inventory management—items in a backpack or the ability to make and spend money. Usually the player takes on a role (an avatar) or controls multiple entities in different roles and makes in-game decisions within the confines of that role. The capabilities of the player are linked to the character that he chose at the outset and that he might choose to augment over the course of the game.

For example, magicians are not capable of physical fights with orcs. To attempt to do so would not lead to a terribly successful outcome for the player's character. However, teaching the magician to fight might just save him when he comes up against an enemy that is immune to magic.

The player can naturally shift the focus of his playing style, depending on the features available and the entity's characteristics at a specific moment in time. Thanks to the human reasoning system, we have the ability to look at a large number of variables and make quick decisions based on prior experience. A player's style at the start of a game might concentrate on deploying weapons, spells, and so on; thereafter, decisions might be made based on other attributes, such as charisma, health, or energy.

Now, on the computer's side, these pieces of information can be fed into a network of weighted decision points, which resemble a fuzzy finite-state machine, a simple neural network (like a Hopfield net) or some kind of scripted decision engine. The important point is that it should be capable of mapping inputs to an output. The output should be some form of behavioral outcome, such as an action or state change, which is based on the way that the inputs are evaluated. The rules that give rise to that outcome are similar to the snap decisions humans make when reviewing their immediate strategies. For example, if the player sees that his energy is low, and a teleport spell is available to a wizard being pummeled by a giant, he

might decide to cast the teleport spell rather than stay and fight with fireballs. The internal decision network (or matrix) has been skewed by a lack of energy, shifting it from "fight" to "flee." The teleport is chosen over running away because the giant might be faster.

Getting a computer to make the same decision might be difficult without hard-wiring this particular reaction. However, it is possible to create a behavioral network that will gauge the effectiveness of a behavior and adjust it toward another tactic—be it random or as part of an informed decision-making process. For example, if the wizard is being pummeled, and he moves away, thus reducing his beating, then the decision matrix might lean toward that action rather than standing and fighting. If the giant seems to be capable of moving at least as fast as the wizard, it is logical to choose teleportation over running away.

As long as the properties of the various in-game roles, objects, and weapons are set properly, the application of simple rules can lead to the correct level of emergence that allows these lifelike decisions to be made. This is A-Life as an extension of AI; the AI provides the basis for the lifelike behavior to emerge—the attributes and personalities of the characters—and the ways that A-Life modifies the strict application of the AI rules are what makes the game more realistic.

RPG Player's Control

Some RPG games act more as high-level simulations—for example, battle simulations like *DEFCON* (Introversion Software) or any of the *Advance Wars* series (including the Wii incarnations). This leaves the computer to decide exactly how the player's commands should be carried out.

The state of the art seems to lean toward using FSMs for this kind of interaction. It just goes to show that a clever combination of FSMs, layered from tactical to strategic and mixed with the player's actions as well as and reactions to the player, can give rise to emergent behavior that suggests that something else is at work. Here, simple A-Life can be used to choose an appropriate sequence of commands in order to make up an overall pattern. Then it is a small step to use behavioral modification to introduce inconsistencies or variations, thus making gameplay less predictable and more lifelike.

At a very high level, the NPCs might even choose the moves themselves, with the player just providing the basis for the behavioral mode, leaving the computer to figure out how it should be carried out. This is similar to the early opening battles in games like *The Seed*, in which the player conducts battles with a point-and-click interface prior to movie-like simulations of the outcome.

Observed Behavior in RPGs

Of course, where the RPG genre can be made to stand out is in the level of interaction between the player's character and other characters in the game environment. Usually, these interactions are based on finite state machines that dictate how the player and NPCs can interact. War simulations, such as *DEFCON*, also apply FSMs at different

levels to give rise to lifelike planning and strategy play. The *DEFCON* AI, for example, has a tactical AI component that is based on an FSM that controls the lower-level functions. This is coupled with high-level strategic AI routines that use an FSM to trigger state changes based on specific criteria (go to a given coordinate, for example).

This is not much different than the standard entity management available in many RPGs. Where they can shine, however, is in the areas of communication and appearance—two sets of attributes that will be covered later under "Video Game Personalities."

A-Life in Puzzles, Board Games, Simulations

These three genres are good examples of how to apply AI and A-Life. In this section, we will be use them for easy-to-follow applications of A-Life. The games tend to be simpler and simulations easier for explaining the various techniques. However, as was previously seen in this chapter, the techniques used to solve problems in these genres can be applied to pretty much any kind of game. The techniques themselves don't really change—just the circumstances under which they are applied. This includes the input data, behavioral environment, and output data.

Before looking at some concrete examples, we should elaborate a little on how AI differs from A-Life approaches to these kinds of games, though the two are closely related. Take the childhood game tic-tac-toe, an easy game to understand and with rules that are very simple. Players take turns placing an X or O on a three-by-three grid board until one player gets three of a kind in a row (see Figure 8.3).

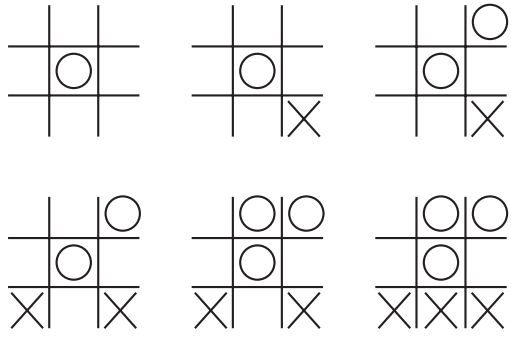

FIGURE 8.3 Tic-tac-toe from start to end.

It should be fairly obvious that a programmer can use AI to create an algorithm that can play the game adequately enough—for example, narrow down the various possible moves between players by using AB pruning. We looked at this kind of search-space AI in previous chapters, and it is a perfectly valid approach. On the other hand, it is also possible to use A-Life to do this in a different way. For example, a neural network can be used to leverage the trial and error that leads to constant stalemates.

If the computer plays against itself for a long enough length of time, it will eventually find a sequence that leads to a win or stalemate from most starting moves. This will require that the system is endowed with appropriate pattern-matching, association, and pass/fail criteria, but the result ought to be successful, regardless of move "look-aheads."

So in this first example (pure AI), the computer is given an algorithm and told to apply it and evaluate the results. The designers *gave* the computer the algorithm, but this does not detract from either its suitability or cleverness, only that it is an AI solution—and a perfectly valid one, at that. However, in our second example, a suitable (although probably not optimal) algorithm will evolve through trial and error. Now, if the artificial life approach that will attempt to put algorithms together and check their validity is a genetic algorithm, then the starting algorithm is prescribed. On the other hand, if a genetic programming solution is chosen, then the designer can only give the system the ability to look, reason, move, and recall previous sequences of moves and hope that artificial life can breed itself a good solution. Underpinning both is a reliance on a fuzzy network of some kind that will learn the effectiveness of certain approaches and how to apply them in the future by mapping input data to output results.

This second (A-Life) approach was made famous in the movie *War Games* (1983), although David Bischoff's book version is easier to follow. In it, a computer system is engaged in a simulation-gone-real game of thermonuclear war. Its creator makes the computer play thousands of games of tic-tac-toe against itself, which eventually leads to the conclusion that the "only winning move is not to play." The machine then takes a look at the unfolding game of global thermonuclear war and surmises that, as in the game of tic-tac-toe, there are unlikely to be any winners. This is an example of applying previous knowledge to a new, but related, problem space.

We cannot hope to deliver that kind of functionality within the confines of this book, but there might be some way to make a system effectively learn how to play a decent game of tic-tac-toe. Some of both approaches will most likely be used—rather like starting with an opening book move in a chess game. In reality, as we have seen, all gaming systems have some AI and fall back on other techniques to provide realism and help the learning process through trial and error.

Bearing this in mind, our point of view will be amended slightly: The genetic algorithm approach to tic-tac-toe is to take an algorithm(s) and modify the extent to which the variables that are fed to it are applied (look ahead, recognizing end games, and so forth). In addition, the controlling AI (state engine) can be applied in a fuzzy way by changing the attributes that dictate the state changes that trigger a transition between the applied algorithms.

The genetic programming approach learns to create an algorithm from smaller parts that will achieve the same result—play a good game. Those building blocks may well be similar to those mentioned above, and the lower level they are, the more the GP result will drift away from its GA sibling. In other words, if GP is deployed with the same algorithms as the GA approach, but the system is given the power to recombine them as it sees fit, then it may well end up constructing a tic-tac-toe engine that

is similar to the GA version. However, at the other end of the scale, if the GP system has to learn how to recombine pieces of the solution such that it needs to rebuild the algorithms, it will make a lot of mistakes along the way. Indeed, it may never reach the same level of competence via simple trial and error.

In video game programming, remember that we defined GP and GA as two watered down versions of digital genetics that follow the previous approach.

- GA = Choice among algorithms and parameterizations
- GP = Creation of algorithms, using abstract representations

An entity that chooses between different known good algorithms with slightly different (non-optimized) parameters in order to find the best approximation by digital reproduction is using a GA approach. The reproduction might be achieved over time within a single entity (simulated learning) or with multiple entities derived from one or more parents (simulated evolution). Also, an entity that creates its own algorithms from a collection of operations (decisions, actions, and so forth), using different mixes to find a solution, is relying on a GP approach. The reproduction definitions still hold true; the same underlying create-and-recombine paradigm is used.

Battleships will serve as our example of a puzzle, board game, or (with some extensions) war simulation. It is well understood by most people and will be familiar. The underlying principles can also be reused in any game that features placement of entities, probing by firing, and strategic destruction.

Battleships

We looked briefly at the way in which AI and artificial life algorithms can be deployed in the simple two-player puzzle game, Battleships, in Chapter 7. At that time, we did not really go into the underlying rules and strategies, so here is a starter guide to the game for the uninitiated. It is important to understand the underlying rules and game flow so that the same view is shared when it comes to solving the problems presented when using AI and A-Life in an innovative fashion. The game itself is a reasonably simple board game with some logic behind the gameplay, but it is not as complex as, say, chess. This makes it easier to describe and manipulate using AI techniques.

The Game Rules

Even for those who unfamiliar with Battleships, the rules will seem vaguely familiar. Many board and video games that are based on the strategic destruction of an enemy use the same basic, underlying principles. This applies as much to traditional board games like *Risk*, for example, as it does to more recent games like A*dvance Wars*.

Like *Advance Wars* (but unlike, say, chess), Battleships relies on the opponent's pieces (ships) being hidden. The goal is to locate (by firing) and destroy all of the opponent's battleships before he destroys yours. Before looking at a computer equivalent, we will describe the gameplay and assume that there are two players, each equipped with pencil and paper. On the paper (which each opponent keeps hidden from the other), there are two grids, as shown in Figure 8.4.

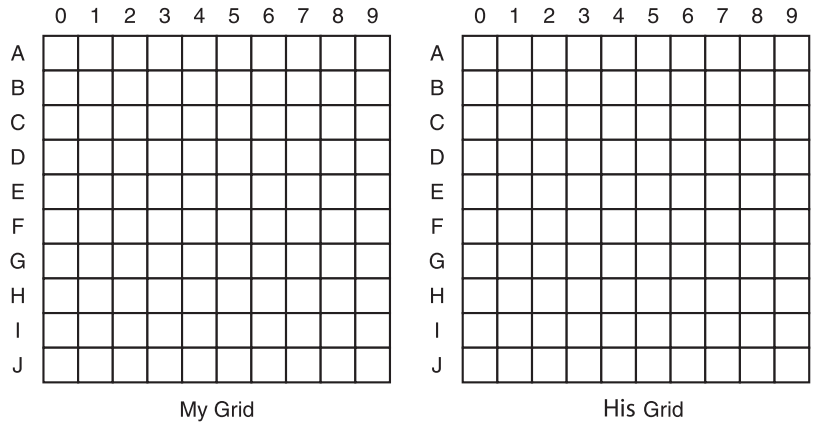

FIGURE 8.4 Battleships game initial grids.

The initial state of play has two empty grids. Each player has a copy of the grid and proceeds to place his craft on "My Grid." The players traditionally have one five-unit ship (aircraft carrier), one four-unit ship (battleship), two three-unit ships (frigate and submarine), and one two-unit ship (minesweeper). The first rule is that ships can only be placed horizontally or vertical, not diagonally.

Having placed their ships, the players then take turns calling out coordinates, following the row-column convention. This makes the top-left corner A0 on the grids. The second rule is: If a shot hits the opposition's vessel, then the player can continue to call coordinates until he misses. The first player to eliminate all of his opponent's ships wins. It is also customary to inform your opponent when a craft is destroyed and what type it was (which can be determined by noting its size).

Those are the rules. Notice that it is possible to abstract the playing model as a collection of finite state machines that can manage the overall strategy of a single player. For example, Figure 8.5 shows the initial transitions between placing ships and playing the game.

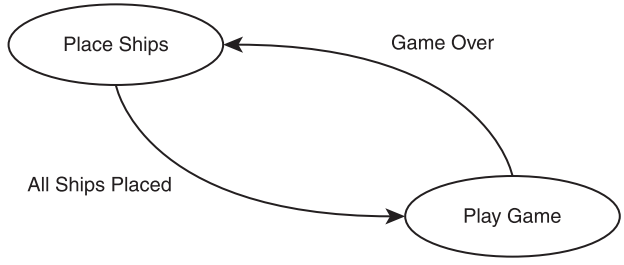

FIGURE 8.5 Battleships FSM example.

A sub-FSM of the Play Game state can be constructed in a variety of ways, depending on the designer's view of the states. In order to do so, we should now look at how to build the AI engine.

The AI Engine

The following techniques can be applied equally well to any game that can be defined in terms of sets of existing known algorithms, such as search or move tables. These algorithms are assumed to have some form of parameterization that governs the way they are applied. In addition, games that deploy specific AI paradigms and strategies that can also be parameterized, and which are not part of the state of the art, can also use these techniques. Traditional AI techniques would probably be easier to understand, but any logical sequence of steps that can be fed preset data as a set of parameters can be parameterized, as we shall see. We will assume that the game is played in the following three stages:

1. Placement of craft.
2. Locating opponent's craft by firing.
3. Destroying the craft.

Clearly, the second and third steps need to be repeated until all the opponent's crafts have been destroyed. For this reason, Figure 8.5 groups them under the state Play Game.

Each of the three steps (placement, location, destruction) can be defined in terms of distinct strategies (algorithms). That means that inside each state of an FSM, there is an algorithm that determines the outcome of any actions carried out as part of that state, which causes a transition to another state.

So in the locating step, there might be an FSM that governs the location of a good place to fire the next shot, based on the application of one of several algorithms in a weighted decision matrix. Similarly, the destruction step needs to identify start and end states, as well as the transition between locating (which may use the same algorithms) and firing.

The algorithms being applied can be optimized using datasets to modify the way in which they are used. For example, the placement of crafts needs to choose between:

- Rotation (horizontal/vertical), and
- Placement (location row/column).

In addition, there are variable data that define the distance of the unit being placed from the playing edge, distances between craft, and so on. All these values make up a set of statistics and measurements that can be abstracted and used as part of the AI/A-Life required to play the game. Clearly, for the placement algorithm, the genetic algorithm approach will be the best fit; however, for the location and destruction algorithms, either a GA or GP approach is possible.

The GA Approach

Since GA uses datasets to modify the way in which a possible optimal solution is applied, the first task is to isolate those specific pieces of data that will be used. These are the data that will be varied when the time comes to apply the algorithm during gameplay.

Using GA in the Unit-Placement Phase

The key pieces of data, as previously mentioned, are:

- Rotation,
- Placement,
- Distance from edge, and
- Distance from other craft.

Other variables to be tracked can be added, such as average craft placement (spread), number of vessels placed horizontally or vertically, and so on, but whether or not these additional statistics have any value would need to be tested. Now that the data have been gathered together, we can proceed to determine how it should be applied. The first attempt might yield a random-placement algorithm:

1. Choose an orientation at random.
2. Choose a location at random.
3. Check that it satisfies the placement criteria.
4. If yes, select the next craft. If no, choose another location.

Testing will show that there might be combinations of placement criteria that make it impossible to place all the crafts while respecting the rules. So there will need to be a way to prevent the algorithm from entering into a selection loop that it cannot exit. However, the basic algorithm still holds true and will arrange the crafts in a valid pattern according to the constraints placed on them by the dataset. The question is: How does this relate to genetic algorithms?

The answer is in how the data is manipulated. GA assumes a causal link between the outcome of the algorithm (as measured against success criteria) and the settings that are applied to it during execution. These settings become the digital DNA from which other groups of settings can be created. When a causal relationship is always assumed, it is possible to apply strict survival-of-the-fittest success criteria. If a collection of settings results in a win, it is kept in the gene pool. If not, it is discarded. The settings for a game are created from two parents in the gene pool and then applied. At the end of the game, the settings are evaluated and kept or discarded. If the selection algorithm has resulted in a collection of settings that is identical to one of the members of the gene pool, then its success indicator is incremented.

In this way, only the least successful are removed when the evaluation stage is performed. One thing is clear: It will take a lot of games before a causal relationship is either proved or disproved. For a simple game like Battleships, it is questionable whether there will be enough real data to make that decision. Chances are that the player will become tired of the game before this happens.

On the other hand, given that there are many iterations of location and destruction in a single game, applying GA to these two algorithms ought to make more sense. The weight of evidence used to create the gene pool is greater, making the outcome more reliable.

Using GA in the Unit Location and Destruction Phases

There are many algorithms for unit location. Each one embodies a specific strategy for the location of ships on the grid. For the sake of simplicity, we will restrict our discussion to two algorithms.

The first is very easy; it selects a coordinate at random from the available spaces that have not yet been fired upon. The only possible variable in this case is the criterion that dictates how far from another, previously tried shot the next attempt should be. This yields a single set of data—the distance (with grid wraparound) from the various "misses" within which the next shot may not fall. Figure 8.6 shows the effect of a value, chosen at random.

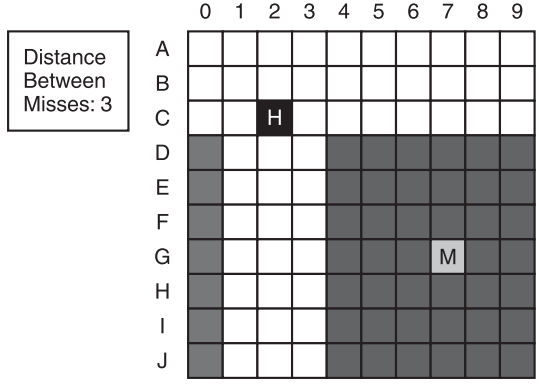

FIGURE 8.6 Battleships hit zones.

In the figure, the only cells in which the next shot may be attempted are the unshaded areas. Anywhere there is diagonal shading may not be fired upon. The success criteria are measured in terms of a hit or miss.

There is a final caveat: As the game progresses, there will be fewer places on the grid to fire shots. Therefore, the Miss distance values will need to be reduced (approach zero) to make sure that a shot is still possible. The GA process needs to be applied to the *starting* values to make sure that the optimal solution is found for the game's opening as well as closing stages. A truly flexible solution is to reduce the Miss factor as the game progresses; the number of squares that have been visited increases in proportion to the total number of squares available. We will look at another way to achieve this shortly.

First, however, let's look at what happens when the AI system detects that a shot has been fired and has to choose another location to fire upon. There is really only

one solution that makes sense, and that is to fire within the radius of the scored hit. The algorithm just tests above, below, left, and right of the hit. When a miss is scored, the state engine reverts to waiting for the opponent to fire, goes back into the unit location state, and flip-flops between waiting and firing until a hit is scored.

This algorithm does not have any real parameters. We might isolate possibilities, such as firing y squares above or below or x squares to the left or right, but this is unlikely to make a concrete difference. As an experiment, the algorithms were tested using a simple implementation of the Battleships program and the initial results recorded. The basic three algorithms tested were as follows:

Algorithm #0: Control (random location, no memory, dumb firing pattern).
Algorithm #1: Control plus memory (remembers locations tested).
Algorithm #2: Reducing hit zone.

Algorithm #2 is the GA-style algorithm, which uses a reducing hit zone as previously described. Unfortunately, it does not fare well, as shown in Table 8.1.

TABLE 8.1 RESULTS OF THE GA-STYLE ALGORITHM

Algorithms	CPU #0	CPU #1	Average Game Length
0 versus 0	50%	50%	185 total moves (92 each)
0 versus 1	20%	80%	99 total moves (49 each)
0 versus 2	37%	63%	146 total moves (73 each)
1 versus 1	50%	50%	79 total moves (39 each)

Clearly, it is the firing pattern and not the test firing location that is integral to achieving a good result. Fortunately, the goal of this exercise was not to create a GA Battleships program capable of beating a human player on a regular basis.

There is another tactic that has been used by Battleships players in the past with some success. It involves choosing a point on the left or top edge of the playing area and testing coordinates in a diagonal line. When the opposite side of the grid is reached, another line is started at a certain offset from the first cell. Commonly, the first square tested is in the top-left position.

This algorithm can be abstracted to a simple case, using an *x* and *y* offset to calculate the new position to test from the last miss. The implementation picks an *x* and *y* offset and proceeds along a diagonal line (or a straight line if *x* or *y* is equal to zero), checking positions for enemy crafts. In doing so, it would maintain the success rate of the combination (hits to misses) as the success criterion, which is used to either discard or store the combination for future use. This is a GA approach that can be

enhanced by varying the input parameters and that might have an effect on the overall outcome. The offset method also hints at how the genetic programming approach might be created.

In addition, the game will begin to adapt itself to the player's style over time. It might learn that the player always positions crafts in a certain way and adopt a straight or diagonal line approach of varying gradients accordingly. Or the designer might decide to feed some of this information back into the ship-placement algorithm, for example. Many variations are possible, including other kinds of input data. For example, the algorithms presented here do not take into account the number of vessels left to be sunk or the ratio of tested to untested squares in a given area of the grid.

The GP Approach

Genetic programming modifies the way in which a *solution* is created from collections of commands, decisions, and data manipulations. The goal is to find a solution, even if that solution might not be optimum (although, as will be explained, it can be improved by using GA). This is an experiment designed to expand on a few of the points brought up in our analysis of genetic algorithm and genetic programming paradigms. It is best to start with an easy game concept like Battleships before trying to apply the paradigms to other, more complex games.

First we need to define a scripting language—that is, a set of commands that can be used to create the program tree. Using the offset example from "The GA Approach" section, it is possible to express the problem domain in terms of a starting point and an offset, leading to the following set of commands:

`Fire (x, y)`	Fire a shot at position x, y (returns result `true` or `false`).
`Was_Hit (x, y)`	Test position at offset x, **y** to see if it resulted in a hit.
`Is_Valid (x, y)`	Test position at offset x, y to see if it is valid.

This is enough to implement basic command trees, but not rich enough for the placement phase of the game. While genetic programming could be used for this phase, it would require additional information on the length of the boat to be placed, as well as some special data on the placement of other crafts. This is beyond the scope of our discussion; for now, we will assume that the placement algorithm follows a random distribution of ships that are bound by constraints relating to their spacing. The GP solution really comes into its own when the location and destruction phases are considered.

Using GP in the Unit-Location and Destruction Phases

In genetic programming, we usually construct the program to be evaluated as a tree in which there are branches (operations) and leaves (values). This makes it easier to manipulate and negates some of the problems associated with rearranging commands, because branches can be moved from one tree to another.

Figure 8.6 shows a sample program tree that determines the coordinates for the offset of the next location to fire upon. It assumes that the `Fire` function will act on the x and y offsets of the `Is_Valid` function that returns `true`. If both return `true` ("1"), then it will likely fire on the first one that does so.

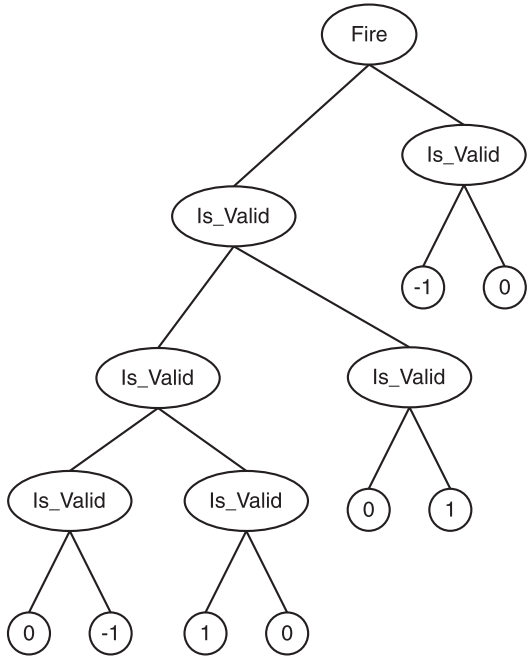

FIGURE 8.7 Ship destruction program tree.

In Figure 8.7, the x and y offsets have been shown as the inputs to the `Is_Valid` functions as values 1, 0, and –1. The result of each `Is_Valid` function is then fed into the one above it; the outputs, therefore, become the inputs to the next layer up.

Clearly, the complete solution for a program tree that is capable of solving all possible scenarios within the chosen domain is almost a book unto itself. However, the underlying principle remains easy to grasp. Using the few commands that are at our disposal, we can put together complex trees that are capable of sighting a target square and firing upon it, as well as know what to do if a square is hit. These can be broken down into individual programs and their structures used to generate entire populations of trees.

Genetic programming is great for solving problems that have a number of possible solutions. For example, when trying to locate a good candidate square to fire upon in a Battleships game, the process is easy:

1. Create a random population of program trees.
2. Test them against the problem domain.

3. Choose the best, following one of a number of methods, and discard the rest.
4. Breed them together to create offspring solutions.
5. Repeat from Step 2.

The breeding part of the process is easy because we are using program trees. To perform genetic crossover, for example, two branches are selected (at random) and swapped. Child A then receives a copy of parent A with one branch of parent B, and child B receives a copy of parent B with one branch of parent A. Mutation, another key part of the genetic programming paradigm, is also easy. A branch is removed and substituted with something entirely randomly generated, but which is still syntactically correct. Over time, assuming that the fitness criteria are correctly established, a Battleships playing algorithm should emerge that plays a decent game—within the confines of the language that has been implemented.

Mixing GA and GP

Mixing GA and GP allows the programmer to create a solution using GP and then manipulate the datasets that are applied to that solution to further optimize it. Take Figure 8.6, as an example. The coordinate values can be considered a parameter set and adjusted using genetic algorithm parameters. If one of the solutions bred out of the GP paradigm happens to be a variant on the diagonal-search algorithm previously mentioned, then the offsets become the data that are input using GA. The datasets are then bred together, using crossover and mutation techniques, in an attempt to further optimize the solution.

From here, it is possible to define the available strategies in terms of known algorithms and their parameters. The parameters can be varied in an attempt to find a good solution, while simultaneously using GP to create an optimal calling sequence for the algorithms. This approach limits GP use to sequential calling, a device that merely cues up algorithms that have been determined by the game designer—a valid approach to the problem.

But what does it mean for video games? In short, on the one hand, it is another way to introduce behavioral patterns that change; on the other hand, it allows us to adapt gameplay to the playing style of an individual. As we saw with the Battleships placement strategy against the actual playing strategy, the amount of data available for training will limit the effectiveness of these paradigms.

There is a slight danger when mixing GA and GP; we are varying two input parameters simultaneously, and the final result might not actually converge in a stable manner. If testing proves this to be the case, then the developer must be ready to backstep and adopt one approach over the other.

Beyond Battleships

As mentioned, when these paradigms are applied to a simple game like Battleships, they are easily understood, but it can be hard to see the point. After all, Battleships is a game that is conducive to pure AI models. However, any game that requires placement and location of units by firing (ammunition, radar) can use these techniques to

introduce variable behavior patterns. GA or GP might be used to select state engine models or directly change the parameters of a known good algorithm.

Games such as *DEFCON* use plenty of AI for choosing paths and weapons. At a higher level, different profiles can be used with different aggression levels or playing standards to try and provide a varied experience for the player. This goes beyond our current example, but it illustrates possible extensions to the basic game.

According to *DEFCON* developers, the AI is a multilayer affair, with tactical AI controlling the individual units and an overall strategic AI controlling the general sequence of events. These behaviors are all managed by FSMs. Individual game units also have the ability to make behavioral decisions—not only those controlled by the game, but also those sent into battle by the player. At this level, too, FSMs control how they react. The player is not always expected to give them direct commands, and it would be inappropriate to let them be sitting ducks; therefore, they are bestowed with some basic intelligence. All of these mechanisms can be combined with GA and GP techniques, such as those used here in our Battleships discussion, and would yield good results.

Finally, *Civilization* in all its incarnations provides profiles that exhibit different behaviors. Due to the fact that this game has a softer approach that leans toward nonviolent, diplomatic tendencies, there are even more options available. Strategy games like these have so many different aspects and variables that they will produce complex models, and these models can be manipulated very easily. Any game that manipulates such a large amount of data has the opportunity to use that data for learning—and thereby make a more interesting experience for the player.

Video Game Personalities

So far, this chapter's discussions have been fairly complex. Therefore, let's round off the edges a bit with some artistic ways that artificial life can be used in video games. The "personality" of a video game refers to the appearance of in-game entities, the way that they are represented, the way that the game sounds and looks, and the impression that it leaves on the player (feels). Examples of how artificial life techniques can make a game come alive include the following:

- Animation
- Sound and music
- Scenery
- Interaction through conversation

The key to creating the illusion of life within the game is to leverage AI, backed up with an application of that AI that is nonlinear. In other words, in the same way that behavioral pattern modification allows video game entities to *act* differently, this section deals with ways to make them look and sound different from each other, also, while still being cut from the same cloth.

The kinds of techniques presented here often fall into the category of "procedural content generation." Adding realism to created content in order to make it

lifelike is just one application of artificial life that can be combined with its generation and thereby improve the immersion for the player. For example, multiple avatars can be created from a single template by varying their attributes. This is an extension of the *Yahoo!* avatar system that allows the user to build his onscreen appearance by changing a set of attributes. While the customization options are very varied and can easily be used to generate random avatars in a game, adding a touch of variation makes the end result much more believable.

This is the key to applying artificial life to procedural content generation—the ability to offer variation. So our final set of examples will deal with breeding artificial life constructs, avatars, and other in-game personalities for use in video games, along with ways to improve the environment by using processor power, rather than design time.

Animation and Appearance

Appearance and animation exist to hold the attention of the player and support his suspension of belief. If an entity's appearance and the way it is animated do not gel, then the spell is broken. Reports of observed playing of *Spore* in *EDGE* magazine, for example, show that the developers at Maxis understood this. In addition, there is a link between appearance and behavior, as well as between appearance and abilities, as the following observation shows:

> *"…no matter how strange the arrangement of limbs and joints on your creature, it will be animated appropriately…the form of your animal will directly affect its abilities; mouth-space determines whether or not it can eat meat, and the construction of limbs alters its chances in combat and social possibilities." [EDGE13]*

Now, the *Spore* interface allows the player to build his own entities and then place them into a virtual environment, much like *Creatures* and *Black²White*, but without the digital chemistry. What is exciting is that the entities' creation process could be extended as an artificial life algorithm. Imagine the implications of this; consider that the *Spore* environment offers a way to simulate the progress of a single organism in a realistic environment. There is nothing, technologically, that prevents this from extending to provide an environment where multiple organisms compete. After all, we only need to select a number of criteria to randomize, create a first population, and then put them all into the environment and see how they fare. As long as there is a valid set of criteria by which success can be measured—such as combat against the player's own in-bred set of organisms—it is possible to breed ever more competitive examples.

Again, there are many ways that this can be further fine-tuned, such as by taking input from the user and copying his creature-design ideas, so that the end competition is more exciting for the player. The player will see his own creatures echoed in the opposition, and this will help add an aspect of ownership to the game. Simple collections of variations on a theme that are purely aesthetic can also

be created. Our *Spore* example equates form with function, but if the game developer only wants to add some visual effects, then the following example should set his feet on the right path.

Deviations from the Average Face

The art of creating caricatures in an artificial fashion is summed up very succinctly in the following quote from *The Tinkertoy Computer*, by A. K. Dewdney:

> *"The program compares the photograph of the target face with an average face stored in the memory of the computer. The features that differ most from the average face are scaled up in size." [DEWDNEY01]*

Depending on which features are scaled up and which are left alone, and by how much the deviations are allowed to creep in, it is possible to use this technique to generate multiple and varied faces. This is one way that an avatar model, such as that used by *Yahoo!* and *Second Life*, can be scaled up to allow more customization. If the user (player) selects features that represent his own facial features, and the underlying algorithm produces rough caricatures by comparing his face with an average face, the player could get a result that is even more like his real face.

Again, using user input to populate the game universe can lead to collections of virtual avatars that are built up by caricatures of existing players. Taken one step further, those avatars can be crossbred, and the results will look natural, being based on existing caricatures of real faces, but each one would be unique. The digital DNA in this case is the set of deviations from the standard average face. Experimentation has indicated that the best way to store these is as a single set for half of the face and then mirror this set for the other half of the face. From here, a very light caricature applied to one side will help make the face seem more lifelike.

The caricature points that make up the average face also serve another purpose—expression. It is possible to create a set of standard deviances from the average face to indicate, for example, smiling or frowning, and these can be applied to the set of data that represents the given specialization of a face. This means that with three sets of data, and only one of them variable, the developer can make *any* generated face smile or frown. There is no need to store sets of smiling or frowning faces for each individual, just a deviation from the average for each expression.

On its own, this might not sound very earth shattering. However, when combined with the ability to generate random believable faces at will, it becomes a much more interesting approach. For example, if the game allows for varied races, then it can store these as deviations and populate a collection of avatars that fit certain characteristics.

The possibilities are endless. Beings (as in *Spore*) could be created with arms, legs, clothes, and so forth based on caricatures of deviations from some nominal standard. Each point that is isolated as being a deviation then becomes one that can be altered and/or animated.

Examples of A-Life in the Gaming Environment

Now that we've looked at the application of artificial life techniques to individual characters, let's briefly turn our attention to examples of artificial life in procedural content generation. Many of the following examples are taken from games that are either in production or development, and some exciting new technologies are emerging. The premise remains the same: Generate a large number of lifelike examples without processor-intensive generation routines and without large quantities of artwork and modeling assets. The main paradigm also remains the same: Use digital DNA, crossbred datasets, and other genetic algorithm techniques. There are two approaches that we will see examples from:

- Use digital DNA directly as a measure of deviation (as discussed in the average face example).
- Use digital DNA as datasets to be fed into a generation algorithm.

While the two approaches seem identical, remember that the first merely aims to reproduce (with variations) an example of an existing sampled in-game entity. The approach tries to create something entirely new, using a specific algorithm. Where they are almost identical is in the way that the initial dataset is propagated, which follows the standard genetic algorithm techniques of crossover and mutation—crossover to mix two parents into a child, and mutation to introduce small, medium, or large random variations into the characteristics. It is interesting to note that the parameters chosen to feed the mutation and crossover algorithms can also be represented in artificial life terms as a dataset that can be manipulated using the same technology. Indeed, if there are success criteria attached to the evolution of the resulting dataset, it can serve as an additional mechanism in the survival-of-the-fittest paradigm.

Graphics and Scenery

The graphics and scenery in modern games represent a huge amount of processor-intensive resources. Even those set in enclosed spaces, like dungeons, require the application of textures and lighting in order to make the end result much more believable and immersive.

A technique used in *CryEngine* (from German studio Crytek, the brains behind the revolutionary *Far Cry*) maps textures onto low-polygon models. This enables the engine to achieve stunning realism with relatively little processor overhead. Of course, the results are relative to the detail resolution; it still takes a fairly powerful machine to produce the end result, but the technique is scalable to other platforms. One other aspect of the technique is that it can be exploited, using artificial life techniques to produce variations. So not only can each in-game entity be regenerated cheaply from stored data sets, but it can also be doctored during the rendering process to change its appearance.

Without getting into the details of the technical implementation, this process would entail manipulating polygon points and their textures. A simple example

might be to vary the tone of the texture in order to change skin tone, or to apply different hair color. More advanced changes might involve subtle variations in the underlying polygon model, coupled with a tweak to the surface mapped texture.

When dealing with multiple polygon models of human heads, we have to be very careful; the more detailed the result is supposed to be, the more scope there is for error. So testing will be paramount; a stream of heads and bodies will be generated and humans asked to single out the ones that are just too much of a deviation. However, the technique can be extended to other things that are part of the scenery, like creating a forest based on deviations from a standard tree. These tend to be slightly less complex models and hence are easier to manipulate. Plus, there are likely to be more trees, mountains, and rocks than people and vehicles in a game, so it would be better to use these techniques for the more plentiful, simpler objects.

The human mind is also great at filling in assumed detail. This means that we can get away with a lot less detail for things that are recognizable and plentiful. So, a forest of trees, for example, can be less detailed, because the player is unlikely to be concentrating on each one individually. However, when less plentiful things pop up, like avatars, the detail needs to be much higher because it is likely to be where the player's concentration is at that moment in time.

As in the average-face example, this is all based on an allowed deviance from an average model, within the limits of caricature. In the case of trees, however, the models are in 3D, which means using a set of polygons as reference, rather than points on a plane.

This is an example of dataset propagation within a standard model (the first approach), but there are also examples of dataset propagation in algorithms (the second approach) in games that are currently being developed. Furthermore, some revolutionary new middleware uses artificial lifestyle algorithms to generate cities.

This middleware is based on an original paper by Yoav Parish and Pascal Müller called "Procedural Modeling of Cities" [PARISH01] in which many algorithms are proposed for creating a city. All aspects are dealt with, from road systems to population density, division of the land into lots, and creation of individual buildings.

This is apparently the basis for new implementations. For example, Gamr7 is middleware for procedural city creation. According to EDGE magazine:

> *"Gamr7's technology sees buildings filling the spaces around the main roads. The different activities that occur in the buildings then further define the pathways of smaller roads, which in turn define the positions of more buildings." [EDGE14]*

This mimics the way that life works and, therefore, could arguably be considered an example of an algorithm derived from life (artificial life) that defines more lifelike algorithms. Furthermore, there are clearly sets of parameters that can be derived and recombined to produce countless examples of (noncomplex) cityscapes, perhaps even on the fly.

Introversion Software is also working on similar ideas to produce cities; however, they note that for 30,000 buildings, their generator can take "several minutes"

to produce a new city. [INTROVERSION01] Given the load times that some platform owners must put up with, this might not seem unreasonable.

The reasons to include this advanced artificial life technology in video games are slightly different than arguments over behavioral pattern generation, which was to provide a way to learn from the player, counteract him, and create a more interesting gaming experience.

An interview by Gamr7's cofounder, Bernard Légaut, for *EDGE* magazine points out that it is not just about realism, but the size of the environments demanded by players. Even if parts are repetitive, gamers expect a play area that is virtually unlimited. And these techniques can help create those unlimited vistas by generating them on the fly with A-Life techniques. As Légaut says:

> *"The worlds game developers build are becoming too big. They are very time-consuming to create…the only way to do it is to use an army of artists who take millions of photos, but that approach doesn't fit into every budget." [EDGE15]*

So although the investment in development time might be relatively large, there are savings on the artistic design side. The true power of the techniques mentioned here will not be realized, however, unless the design and development teams come together to effect their application. Only then is the true impact of artificial life in procedural content generation likely to be felt—by both the gamer and the accountants.

Interaction through Conversation

As a final example of using artificial life in video games, we turn our attention to communicating with the system using natural language. This includes feedback from the system to the player, as well as conversations with the system as a way to progress through the game's unfolding story. Previous chapters have discussed this topic in a roundabout way. The approach hinges on being able to construct believable phases from collections of words that *may* be strung together in a specific pattern.

The creation of the phrases requires as input data a network that has been trained on examples of phrases that convey the required meaning. It is possible to step beyond this to allow free generation of text from a huge collection of possible words, but this route carries its own risks—chief of which is that the smallest deviation from the norm could produce rubbish. The theory is that if the computer can locate the subject(s) in a sentence, a believable piece of conversation can be produced by linking the target words with examples of words that have previously been used in the same phrase. This practice is dangerous because the potential for generating gibberish is very high. However, as a way to generate examples of possible sentences within strict confines, it works very well. The higher the level of control, the better the results, although there will likely be fewer possible variations. The algorithm itself is very simple and can be used to great effect, just by manipulating the connecting weights in a network of trained nodes.

Creating Prose

The entire language-creation process draws on every aspect of AI and artificial life that we have seen in this book. It begins with the training of a network of nodes that connect words together. Each node represents an entry in a sentence (a word pair), constructed as a network of weighted probabilities that have been derived from an observed set. For example, suppose we were to analyze the following phrase:

"The quick brown fox jumped over the lazy dog."

This would yield a set of word pairs, as shown in Table 8.2.

TABLE 8.2 WORD PAIRS IN QUICK BROWN FOX

Word Pair	Number of Pairs
(start) the	1
the quick	1
quick brown	1
brown fox	1
fox jumped	1
jumped over	1
over the	1
the lazy	1
lazy dog	1
dog (end)	1

According to Table 8.2, it is clear that only one phrase can be constructed from the resulting network. However, only one variation has been analyzed. If we add the following to the knowledge base:

"The green orchid grows under the towering maple."

We would start to see some variations, as shown in Table 8.3.

TABLE 8.3 VARIATIONS ON A THEME

Sentence #1		Sentence #2	
Word Pair	**Number of Pairs**	**Word Pair**	**Number of Pairs**
(start) the	2	(start) the green	1
the quick	1	green orchid	1
quick brown	1	orchid grows	1
brown fox	1	grows under	1
fox jumped	1	under the	1
jumped over	1	the towering	1
over the	1	towering maple	1
the lazy	1	maple (end)	1
lazy dog	1		
dog (end)	1		

If we put the contents of Table 8.2 into a weighted decision network, we begin to see the effect of the analysis of the word pairs:

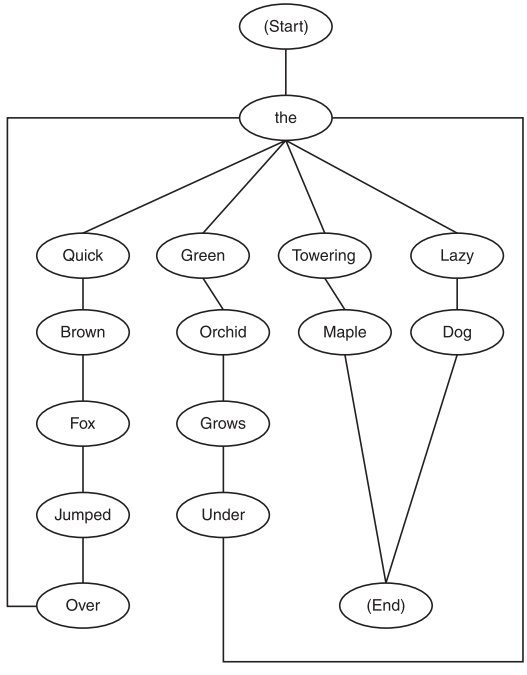

FIGURE 8.8 Sentence tree.

From the network in Figure 8.8, we can construct phrases such as:

- The lazy dog.
- The towering maple.
- The green orchid grows under the lazy dog.

This shows the importance of creating networks of related prose in an attempt to mitigate the chances of producing gibberish. The network is generally applied in reverse, so the algorithm becomes simple, as explained by Dewdney regarding *The Tinkertoy Computer*:

```
repeat
 r ➔ random
 determine pair follower
 output follower
 first ➔ second
 second ➔ word
until someone complains
```

When a random number *r* is selected, it determines a follower by adding together the probabilities stored for each of the words that follow the given pair, until those probabilities first equal or exceed *r*. [DEWDNEY02] Clearly, by using this approach, genetic algorithms can be brought into play by fiddling with the probabilities and creating child solutions that are slightly different from the parent. So we might have two characters—one given to verbose oration and the other more restrained, and ask them to generate prose on the subject of, for example, a maple tree. Having identified the adjectives in a network of possible word combinations, we apply a expansion weight to one set and a restraining weight to the other, so that the frequency of adjectives is reduced for the entity not given to flowery prose. Therefore, the first entity would describe a maple tree as:

"the towering maple tree,"

And the other entity simply as:

"the maple tree."

The inclusion of more adjectives enables further possibilities for verbosity. There is also a genetic programming equivalent to this approach that relies on a sentence tree that can be referenced by one of its nodes and used to construct sentences. Using this tree, we can pick up a subject—say, fox—and construct sentences around it by using genetic programming-style techniques. This requires that the tree be constructed like the tree in Figure 8.8, so that branches can be swapped to change the sentence construction.

Reacting to the Player

Finally, these techniques can be combined with others to enable the system to hold a pseudo-conversation with the player. *EDGE* magazine, in reporting on play sessions with Frontier's *The Outsider*, points to a possible approach:

> "The Outsider's *conversations seem positioned somewhere between adventure game dialogue trees and context-sensitive squad commands." [EDGE16]*

The dialogue trees can be manipulated in much the same way as any other tree structure in terms of passing from one node to another (using weighted links) and the ability to swap branches between parents in creating children. As an additional bonus, artificial life constructs can also be used to test the resulting prose against a known good dataset. This is necessary because when reacting to the player, we will again encounter the emergence of behavior in an environment that is either controlled to the point that the result is predictable or so open that errors are inevitable.

So when testing against the known network of possible word chains and using this as the criteria for success, it is possible to limit the excesses of implausible behavior. This can (at least in theory) be fast enough to perform in real time, but doing so might be limited by the performance of the platform and the other tasks being concurrently processed. The likely level of emergence to be encountered is hinted at by David Braben, founder of and developer at Frontier. Braben's comments are good reference material for topics related to procedural content generation, having been part of the programming team (with Ian Bell) that brought us *Elite*, which crammed an amazing amount of procedurally generated content into 16KB of user RAM.

> *"The two aren't that far apart from each other, really, and that's the point…when you're shouting story-related things to each other in battle…it gets really interesting." [EDGE17]*

Despite Braben's comments, it is the experience of industry professionals that natural language generation systems remain quite limited in their capabilities. Although some simple generation trees might make their way into games as a kind of sideshow, the more advanced systems are not yet ready for commercial deployment. The result is that much computer-generated prose in games will remain quite prescribed and repetitive but still be much better than none at all.

SUMMARY

This chapter has tackled several different kinds of games, from shoot-em-ups to board games, simulations, and role-playing games. Our attempt has been to show examples of artificial life that draw on AI and can be implemented in a number of different genres and situations. It (along with the previous chapters) should provide you with enough information to implement games enhanced by A-Life techniques. Realize that there are no complete solutions offered here, because there are

no complete solutions. Each game, each designer, and each developer will add his own layers of technology to the end product.

Specifically, the designer will decide where A-Life will hook into the game design in a bottom-up fashion. The developer will offer innovative ways to make use of the data that is supplied by the artists and designers, and in doing so enhance the gaming experience without substantially increasing overhead. Each of the design decisions should be made with A-Life in mind—from deciding what mechanism will generate in-game variations to how the various enemies (if there are any) should react and interact. This provides sound guidance for the developer's framework, on which extensions can be added to render a truly lifelike game environment.

As this chapter has shown, A-Life can be as trivial as offering some nice effects as a flock of birds flies across the screen, or it can be as elementary to the game as an algorithm for deciding what to do next, based on previous game sessions. Some of the applications are aesthetic, others are revolutionary. Only the game's creator will be able to spot which to take advantage of.

References

[DEWDNEY01] A. K. Dewdney, *The Tinkertoy Computer*. W. H. Freeman and Company, p. 205, 1993.

[DEWDNEY02] A. K. Dewdney, *The Tinkertoy Computer*. W. H. Freeman and Company, p. 203, 1993.

[EDGE01] "2D or not 2D: Capcom Puts the Fight Back into the Fighting Genre." *EDGE*, No. 184, p. 51; Future Publishing, January 2008.

[EDGE02] "2D or not 2D: Capcom Puts the Fight Back into the Fighting Genre." *EDGE*, No. 184, p. 49; Future Publishing, January 2008.

[EDGE03] Criterion Games/EA on *Burnout Paradise*. *EDGE*, No. 184, p. 32; Future Publishing, Christmas 2007.

[EDGE04] Liquid Entertainment/Codemasters on *Rise of the Argonauts*. *EDGE*, No. 181, p. 37; Future Publishing, November 2007.

[EDGE05] Slant Six Games/SCEE/SCEA on *SOCOM Tactical Strike*. *EDGE*, No. 184, p. 91; Future Publishing, January 2008.

[EDGE06] AM2/Sega on *Ghost Squad*. *EDGE*, No. 184, p. 87; Future Publishing, January 2008.

[EDGE07] AM2/Sega on *Ghost Squad*. *EDGE*, No. 184, p. 87; Future Publishing, January 2008.

[EDGE08] Wayforward Technologies/Konami on *Contra 4*. *EDGE*, No. 184, p. 89; Future Publishing, January 2008.

[EDGE09] Yoshinori Ono on *Street Fighter IV*, "2D or not 2D: Capcom Puts the Fight Back into the Fighting Genre." *EDGE*, No. 184, p. 52; Future Publishing, January 2008.

[EDGE10] "The Club." *EDGE*, No. 183, p. 33; Future Publishing, Christmas 2007.

[EDGE11] Silicon Knights/Microsoft Game Studios on *Too Human*. *EDGE*, No. 183, p. 30; Future Publishing, Christmas 2007.

[EDGE12] Eden/Atari on *Alone in the Dark*. *EDGE*, No. 181, p. 34; Future Publishing, November 2007.

[EDGE13] Maxis/EA on *Spore*. *EDGE*, No. 181, p. 30; Future Publishing, November 2007.

[EDGE14] Codeshop section. *EDGE*, No. 183, p. 119; Future Publishing, Christmas 2007.

[EDGE15] Bernard Légaut, cofounder of Gamr7, in a Codeshop Section interview. *EDGE*, No. 183, p. 118; Future Publishing, Christmas 2007.

[EDGE16] "The Outsider." *EDGE*, No. 183, p. 32; Future Publishing, January 2008.

[EDGE17] David Braben, Frontier CEO, in an interview on *The Outsider*. *EDGE*, No. 183, p. 32; Future Publishing, November 2007.

[PARISH01] Yoav I H Parish of ETH Zurich, and Pascal Müller, of Central Pictures, Switzerland, presented at SIGGRAPH 2001, http://www.centralpictures.com/ce/tp/paper.pdf.

[INTROVERSION01] Entry on Procedural City Generation for the *Subversion* project. Introversion blog, http://forums.introversion.co.uk/introversion/viewtopic.php?t=1132.

CHAPTER

9

MULTIPLAYER AI AND A-LIFE

In This Chapter

- Managing Emergence
- Enhancing the Experience
- Implementing A-Life in Multiplayer Environments

This chapter diverges a bit and looks at the way that AI and A-Life can be deployed in a multiplayer environment. The consequences of A-Life (and AI) in a multiplayer environment are a little different from other gaming environments, and it is worth taking the time to examine some of the unexpected bonuses that arise as well as a few points to watch out for. There are many types of multiplayer environments, but they can be summed up into three broad categories:

- Multiplayer puzzle games
- Multiplayer real-time environments
- Asynchronous multiplayer environments—that is, Web browser-based games

Multiplayer puzzle games include those styled after the classic *Risk* and are games that have strict puzzle-style rules. Online *Civilization* also falls into this category. These games are typically turn based and require finding solutions, as well as having more traditional interactive elements, such as combat.

Real-time environments are usually based on combat in a semi restricted or open game universe. Games such as *Eve Online* (which are played in a space-oriented arena) or *EverQuest* (a role-playing game set against a fantasy backdrop) require different AI and A-Life than puzzle-style games. They tend to have more complex, open, scriptable interfaces that reflect multiple options for the gameplay that they offer. There are rules that must be followed, but like in a game that tries to limit the player's options at every step of the way.

Finally, asynchronous multiplayer environments have a very unique set of limitations and possibilities because they do not allow very much in terms of in-game player interaction. This has two consequences. On the one hand, it becomes very difficult to tell if the player is human or computer controlled; on the other hand, the environment tends to be very restricted. However, in all three multiplayer gaming models, there will be strong emergence linked to the unpredictability of human players. The interaction of the player characters, NPCs, and game environment, be it in real time or asynchronous, will lead to unforeseen behavioral sequences.

Even when the game environment is strongly managed by a set of strict rules, the emergent behavior of the system (including NPCs and player characters) is often something new and unexpected, simply because no two human players ever react in the same way. This is useful because it can be used to fuel the A-Life that represents the computer opponents—that is, the NPCs or game environment itself.

Then there is the case of the ever-expanding game universe—such as in *Eve Online*, *EverQuest*, or *Second Life*—which brings another layer of emergence to the gaming environment. Not only are new players constantly entering the gaming environment, but they are also carving out their own little areas of the virtual world and putting pieces of in-game assets together in new ways.

Players generally spend most of their in-game time competing against each other, but the addition of A-life, computer-controlled entities adds an extra dimension. These entities can also compensate for a sparsely populated gaming environment that is in its infancy and give the illusion of a more heavily populated game universe.

It is important that the A-Life deployed provide more than just a reaction to player interaction. Although the NPCs will likely drive the process in the same way as in a single-player environment, when multiple players are involved, there is much more scope for creativity and observation. This point is important to understand before continuing on with this chapter: A-Life is as much about environmental input that creates successful (and varied) behavioral models as it is about *creating behavior from nothing*. Given this premise, multiplayer environments offer much more than just a community of human players with varied interactions to play against.

Managing Emergence

The interactions between players, the game system, and computer-controlled entities will invariably provide emergence. Usually this will be strong emergence, as human players are perhaps the most unpredictable collection of in-game entities that the developer will come across. It is nearly impossible to correctly predict in a gaming environment what the players will do, collectively. In other words, while we might accurately predict single-entity behavior, when all the entities are combined, it becomes impossible to foresee the net outcome.

Furthermore, given a net outcome, it also becomes impossible to determine exactly what has led to that outcome. Different human (and computer) players have different behavioral triggers and will sometimes do things that are unpredictable. When all the effects are added together, the resulting strong emergence makes it hard to control the system as a whole.

Artificial life in the multiplayer environment offers a kind of emergence that is different than the natural emergence that occurs when the game puts a number of entities together and allows them to interact. In fact, A-Life in the multiplayer environment ought to provide substitute human players in order to stimulate and fill the gap between an empty and full game world. Depending on the goals of the developer, this might or might not be a problem that needs to be solved. Either way, it might prove that the best approach, from the game system point of view, is to treat all entities in exactly the same way, be they human or computer controlled. One advantage here is that some of the excesses of emergent behavior, from an individual's point of view, will be mitigated by actions taken by the system to limit its behavior, exactly as it would if a human player tried to do the same.

We also assume here that "multiplayer" equates to "online multiplaying"; there are multiplayer games that do not involve using the Internet, such as *Magic: The Gathering, Duels of the Planeswalkers*, which makes use of intranet-based opponents, but the vast majority of the issues and solutions discussed in this chapter involve online multiplayer and massively multiplayer games.

There are also different platforms to consider—from text-interfaced Web-based games to those with a proprietary interface. The latter makes use of the game as a software front end (which is the basis for the majority of big-budget, highly popular games). There is, however, a whole slew of independently produced online Web

browser-based games that can benefit from the techniques (and suffer some of the same drawbacks) encountered when we consider emergence versus A-Life.

Emergence versus A-Life

Emergence is not the same as A-Life, but it can resemble A-Life in a multiplayer environment populated mainly by player characters. In other words, since human players tend to be unpredictable, rules and scripts will be triggered and interpreted (respectively) based on their unpredictable input. No amount of testing can prepare a gaming environment for the inventiveness of individual human players, which is doubly true when multiple players encounter the gaming environment. As a consequence, safeguards need to be in place that can detect and counteract undesirable behavior brought about by the unpredictable nature of the multiplayer environment.

Individual emergence will be high—after all, game-controlled entities will potentially have a lot of behavioral patterns to feed from. This might be in a direct fashion—specifically leveraging algorithms to identify and copy observed behavior—or in an indirect fashion, as a consequence of rules triggered by interactions.

The more intelligent individual entities are (using AI learning techniques), the more they can use human behavioral patterns observed in the game. Even if just a self-organizing neural network is used to absorb the effect of interactions, some of the observed behavior will be acknowledged. After all, this is exactly what humans also do—learn from each other. On the other hand, this level of AI is processor expensive, and in an online multiplayer environment, processing power is in constant short supply. As with many of the examples in this book, the goal is to find that balance between a level of AI that is good enough, keeping it (and its peers) in check, while minimizing the use of resources.

Strict rule-based AI is likely to be the only approach that makes sense for the vast majority of in-game computer-controlled characters. After all, they are only there as fill-ins, and not as substitute players. Those with other roles to play can be dealt with using purpose-specific AI and A-Life algorithms, but NPC emergence and "population control" take this point a little further.

NPC Emergence

Nonplayer characters are entities that take part in the gaming environment as if they are human-controlled players, but they are actually owned by the system in charge of the game. NPCs might comprise cannon fodder, substitute players (for games with a fixed number of participants, such as in soccer management simulations), or characters with special roles, such as shopkeepers, police, and the like.

Players that interact with NPCs will not necessarily know at first that they are NPCs and might treat them as if they are human players. In other words, the NPCs will be treated in an unpredictable fashion. Within certain limits, it might be possible to predict how this interaction will take place, but anticipating this behavior will be difficult, to be sure. This means that, unchecked, the kind of reaction that might emerge may well be undesirable and detract from the game, rather than enhance

the experience. If the NPC is following strict behavioral rules and has to deal with multiple individual player-controlled characters, then the AI used has to be able to deal with this. There are two (extreme) options available here: We can make the entities intentionally very restricted in their behavioral patterns (dumb) and hence predictable, or we can make them very clever and adaptable and micromanage their behavior.

On the one hand, a predictable NPC has the advantage that it will probably not embarrass itself, and the emergent behavior that arises from evaluating many different inputs will be easy to test and control. However, the NPC will be less lifelike than might otherwise be expected from the game, depending on the genre.

In single-player games, this is less of a problem because the player is usually in the position of "one against many." First-person-shooters, for example, can allow their AI models to be less varied, because the lack of unpredictability is mitigated by the weight in numbers. A game such as *Unreal Tournament*, however, pits players against each other, and so the NPCs have to be endowed with more creative AI. These games usually have a finite number of opponents, and so the processing power devoted to their control can be easily planned for and well utilized.

In the other extreme—games that are more or less open ended and populated with a number of NPCs—the resources might not be available to micromanage a very clever and adaptable piece of artificial life. Left to their own devices, the NPCs might begin to behave erratically. This is not a new problem, but it will usually be more pronounced in the multiplayer arena than in single-player games, because the AI will be fed much more input. There is a balance to be struck between lifelike behavior and the extent to which that behavior must be controlled in order to keep it within acceptable bounds.

Rule-Based Emergence

One other interesting use for AI and artificial life in multiplayer video games is as a complement to the player's own character or role (avatar). For example, a space trading game might be enhanced by the ability to build a player-controlled fleet of spacecraft. Their interaction with the game environment will need to be governed by rules that dictate how they are supposed to react in given circumstances. The way that interactions play out between NPCs and the player characters will lead to emergent behavior based on the constant re-evaluation of those rules. The difference here is that the rules (from a collection of possible rules presented at the game design level) are set by the player. The combinations of rules that players will inevitably try will cause their own kind of emergence. Added together, this will lead to a kind of strong emergence that has to be managed separately.

Enhancing the Experience

This section deals with ways in which A-Life can be used to enhance the gaming experience of players who are involved with a variety of different multiplayer game genres—from *Elite* and *Eve Online*–style space trading games to online FPS

and fighting games. Here, the only reason to include A-Life in a project is to enhance the experience for the player and make the game more immersive. Every decision to include some new facet of AI or A-Life needs to respect this goal. Otherwise, including AI and A-Life is a waste, both in terms of design-end development and in-game resources. One thing that all of the categories mentioned in the first section of this chapter share is that the addition of multiple human players makes the games stand out. Therefore, the emphasis on AI is slightly different than in their single-player versions.

The Purpose of AI and A-Life in Multiplayer Games

There have been several online discussions relating to the use of AI in multiplayer games. Some even suggest that one day, every game will be multiplayer, and there will be no need for AI or A-Life at all. [AIBLOG01]

Depending on the game, multiplayer AI and A-Life can have many purposes, all toward the goal of providing immersion. They can be used to challenge players, giving them obstacles to be overcome in order to gain experience without necessarily competing against human opposition—a kind of training.

Then there are the various management functions that need to be AI controlled, such as shopkeeping, assigning quests, or even policing the game environment. In short, there are behind-the-scenes roles that real players (even if capable) cannot fulfill, and these will rely on AI.

Adding a spark of A-Life to the mix will ensure that intelligence can be extended beyond simple interaction trees. This might only be aesthetic, such as changing appearances to reflect the "moods" of the entity, but even this goes a long way toward improving the player experience. Work done by the Synthetic Characters Group at MIT (Massachusetts Institute of Technology) Media Lab, for example, focuses on the interaction between the player and NPC entities. [BLUMBERG01] As Bruce Blumberg notes, the goal was to "understand and build the kind of AI needed to create a sense of an inner life."

In addition, the game might provide minions for the player to manipulate, even if they are not complementary to the ongoing game. These bots should also reflect a mix of rule-following enhanced with artificial life to give them character and individuality, as well as a touch of unpredictability. This unpredictability in the behavioral model would also serve to test the player's skill, making it part of the challenge of the game. Again, however, because the entities are formally out of the player's control, it is the system that manages the impact of the emergent behavior of the interaction of these collections of minions. Some of these minions might be teammates that help the player achieve his goals. The algorithms used here are similar to single-player environment games, and the challenges are the same as those previously discussed.

Finally, in an attempt to mitigate some of the issues relating to disconnections in online gaming, the design might call for an entity that is capable of carrying on when a player's connection closes. There are two sides to this:

- The first is as protection for the player. If his connection should fail for any reason, he might like an AI system to take over and perform simple tasks in his stead. These might include getting their character to a place of safety or just telling other players that he has gone offline—or even continue to play the game until some end criteria are satisfied.
- The second side is to prevent (or at least discourage) intentional disconnects intended to avoid in-game consequences—such as to avoid defeat in the end stages of a heated battle. The mechanism might be the same, there being no way of telling one set of circumstances from the other; but the substitute character should never be more effective than the player in order to avoid the gray behavioral area between system exploitation and outright cheating. The behavior that emerges should sufficiently back up an honest player while not providing unfair advantage for less competent/honest ones.

These cover some of the interesting approaches that have been suggested in the past and some of the pitfalls that a fully emergent system brings about when using AI and A-Life in video games. There are two additional techniques that need to be discussed before looking at the possible implementation of some of these ideas.

Behavioral Cloning

Here, the term "behavioral cloning" refers to using the input from the system's observed behavioral patterns to train a substitute system with the express intent to reproduce the original behavior. Remember that this involves creating a self-organizing neural network that links together the facets of the behavior and then training it by increasing the associative weights in the network. Then it is a simple matter to introduce one or two key "triggers" into the network and read those neurons that are, in turn, activated.

Stated like this, the reader might be feeling that it is an easy proposition. However, truly self-organizing and self-balancing networks mimicking player behavior are probably a long way off. Nevertheless, there are many individual aspects of the player's behavior that can be mimicked relatively easily, just not used to the exclusion of all other techniques.

As long as the limitations imposed by current technology and understanding are respected, there is no reason why the techniques cannot be used in a piecemeal way to enhance various behavioral aspects of the system.

The multiplayer environment contains a rich set of input data that can provide input to the system. This is similar in many ways to the techniques used in the design phase to create the game's behavioral patterns. However, here the risks and rewards are potentially greater. The rewards are many because of the level of variations in player behavior that will likely be encountered; the more players there are, the more different behavioral patterns are likely to be witnessed. Of course, the risks include overloading or overtraining the networks, as well as the risk of excessive behavior in one direction or the bland averaging out of a behavioral model as input is fed in from all sides. So while copying behavior observed by the NPCs and gaming environment is attractive in A-Life terms, it is not without issues.

Some of these can be solved by weighting the response of the network toward the general behavioral trends that it observes, based on the initial observation. This uses the systematic reinforcement of initial impressions. For example, in *Black&White*, the player is able to interact with his in-game entities by rewarding or punishing them (stroke/slap). If his entity does something that the player does not wish to encourage, the entity can be slapped. Conversely if the entity does something desirable, the player can stroke it. Systematic rewarding (weighting) bad behavior leads to an world that is "black," whereas systematic rewarding good behavior leads to a world that is "white."

Taken to the multiplayer environment and the stroke/slap philosophy replaced by a system of first impressions, an entity can feed off of observed behavior and weight it—not toward the most commonly witnessed behavior, but toward the *first* witnessed behavior. Subsequent observations would attach more importance to those behavioral patterns found to be in common with other entities (players), and those patterns would be reinforced in its own behavior. For example, if an entity was initially subjected to "bad" behavior, then it would tend to evolve with a more pronounced aspect of that "bad" in its character. This is a good way to counteract some of the issues encountered when trying to feed of all the observed behavior in the same way, which ends up with a bland average of all possible behavioral traits—both good and bad.

To do any of this, the system needs to be able to record the behavior, measure it, and replay it in a different context. This book presented ample examples of these techniques in Chapter 5, "Building Blocks," which can be used as just another way to leverage the previously covered systems. In addition, there needs to be a mechanism to create or derive variations in that behavior and replay them while checking actual against desired performance/outcome. In other words, in addition to cloning behavior, the system will also use this data to seed new behaviors (reusing techniques from Chapter 8, "Several A-Life Examples," which will be revisited in Chapter 10, "The Application of A-Life Outside the Lab").

Finally, behavioral cloning can be used as a way to prevent "disconnect cheating." To do this, the system needs to be able to weigh a behavioral model toward the real capability of the player. The end goal in this case is to provide a behavioral model that is of the same standard (or slightly less) as the player being replaced. This ensures that there is as little discontinuity as possible in the game, while removing a loophole that could be possibly exploited.

This loophole would be the player performing an intentional disconnect, knowing that his clone is likely to play the game better than he is. Clearly, if the player is aware that the clone is likely to fail, where the player might succeed, the player will be more inclined to stay online and perform to the best of his ability.

Of course, this can be taken a step further, allowing the player to actually stipulate how his clone should behave in the event that he is not there to play himself. Again, though, this needs to be balanced against the possibility that intentional misuse of the facility might occur in order to try and win a battle that the player would otherwise lose.

Preventing Cheating

There is another side to cheat prevention, too. Chapter 7, "Testing with Artificial Life," mentioned that it ought to be possible to use artificial life techniques to prevent players from finding an unfailing way to kill enemies and then use these techniques to unfair advantage. This is common in online games such as *Eve Online* and others, where buccaneering and other practices pervade—for example, killing the same monster repeatedly with the sole goal of racking up enough points to either level up or sell points on the general market. This has the effect of distorting the game, such as when people with real-world money buy better equipment for their in-game exploits, rather than earning them the hard way—through bitter experience. While this might not be cheating per se, it can often be a case of the original intention of the game designer being turned in a direction that he did not expect or want.

It's up to the game designer to either adequately reward the player or find some way to make sure that the player has to stick with it. Of course, we also have to respect the abstract notion of fun and make sure that the game does not become a chore. Failure to do so will inevitably lead to the game's reach being reduced.

The answer might be to introduce AI entities that can learn and evolve—always adding unpredictability and remaining at the same difficulty level. To do this, all we need to do is ensure that the difficulty quotient of a specific creature is adapted to the observed experience and skill of the opponent. There are two ways to do this: either keep track of the experience of the human opponent as part of the interaction data or measure perceived capabilities in real time and try to stay one step ahead of them.

Eventually, however, the entities should be able to be defeated, unless there is such a large difficulty gap between the player and the opponent that he has bitten off more than he can chew. The underlying technology, again, ought to be familiar; this is just another implementation of our previous, flexible approaches to video game AI.

One approach would be to use the underlying scripting engine to create opponents that are very skilled and then adjust their capabilities (speed, strength, reaction time) according to the circumstances. There might even be one single FSM that is used for all entities, and this FSM is merely deployed in different ways, depending on the class, experience, and desired difficulty level.

However, there is also a question of balance here. The developer should not make the game so hard that new players can't advance. Nor should the system be opened up so that players can produce their own bots and rack up points that way. This last suggestion might seem far fetched, but it is neither terribly difficult to do nor as rare as you might think.

Self-Policing

Part of using AI in multiplayer video games is for policing—in essence, guarding against cheating within the rules of the game. Examples include players choosing to pursue an in-game life of crime—allowed by the game engine, but with clear consequences if they are caught, for example, when smuggling firearms into a protected area. AI can be used to make the video game police itself (staying in character, of

course) by providing suitable enforcement entities that exist as part of the underlying system (such as space games that have police ships). This relies on having strict rules. The players must know what constitutes breaking the rules and what game-oriented penalties can be enforced by the policing entities.

For example, in *Baldur's Gate II*, guards are used to enforce the rules of the game. Repeated infractions of the game rules (such as burglary) will inevitably lead to them swarming in and killing the player. A softer example is the police crafts in *Elite*, which will descend on persistent offenders but can be dispatched by a skilled player.

Of course, they tend to come back in waves to wreak vengeance on the player for destroying one of their own, and the player therefore becomes a pariah, unable to dock in any system where the rule of law holds. These are examples of where an easy link can be made between the player's actions and the consequences.

Thus, AI is used to enforce the rules, and artificial life applied to make implementation of the enforcement more realistic and (perhaps) give the player a way to escape punishment if he possesses enough skill. Again, the balance of player versus system will need to leverage AI to make sure that the correct difficulty level is set or if escape is an option.

AI can also have a role in determining when and where the rules have been broken. One thing that AI is good at is making decisions based on pieces of information (in an automated fashion) that a human would find difficult/impossible to absorb all at once. As you will appreciate by now, the multiplayer game is so densely populated with entities, all doing their own thing, that trying to police it in a manual fashion is probably not practical. Add to this the fact that sometimes, spotting an infraction when it takes place is not clear cut; murder is easy to spot, fraud is less so. The crime itself might result from a chain of events that leads to a sinister act.

Of course, a simple expert system or decision network (or even an FSM) could cope with this kind of evaluation and identification process. The fact remains that machines can, when equipped with the kinds of AI covered in this book, perform monitoring and enforcement activity autonomously and reliably.

In a massively multiplayer environment, it might be impractical to monitor the activity of every single player. In a distributed environment, that might not even be possible; so patterns of behavior must be used to spot places where the rules are possibly being broken and warrant further investigation. This is like real-life fraud monitoring. The reality is that as a game becomes popular, it will attract fraudulent behavior—and, paradoxically, the system managers will increasingly come to rely on AI routines to monitor the behavior of players, because it will simply become too difficult to police them manually.

Implementing A-Life in Multiplayer Environments

Having looked at the ways in which AI is relevant to multiplayer video game design, now our attention can shift to the A-Life implementation details inside the game system. Bear in mind that the solution's core consists of the same AI routines covered earlier in this book, such as those in Chapters 5. The A-Life is being used as intended

to add more realism to the game. In essence, there is no real difference between the way that AI and artificial life are implemented in multiplayer projects compared to single-player projects. Multiplayer games just have different goals, opportunities, and possible pitfalls. That is, many of the same principles apply, and the interaction points will be the same, but everything will likely be on a bigger scale. At least from the interaction point of view, the chances are that there will be more individual entities in the system than a single-player game might entail.

The key difference is that in a single-player environment, all opposition is either human or computer, with the human control focused on a single entity or well-defined collection of entities.

In the multiplayer environment, the player is never sure which entities are computer controlled and which are human controlled. The mix of interacting entities at a given point in the game could range from those that are purely computer generated (rare) to those that are entirely other human players (which might be more common, depending on the game). The most likely possibility is that a given collection of interacting entities might be a mix of computer- and human-controlled characters.

There is one aspect to cover before determining how, from a behavioral point of view, artificial life will be implemented. How do we manage the number of in-game computer-controlled entities that are present in the system (population control)? For those games that do not have a set number of entities and are more or less open to expansion, this question is reasonably complex when limited computer resources are taken into account.

Population Control

A multiplayer environment will have a finite population that can be supported by the system, and its number should be easy to calculate. Chances are that it will seem impossible to calculate how many computer-controlled entities can be supported, so a few simulations need to be conducted on paper. The reason for doing this is that it is necessary to calculate the possible cost and ramifications of taking an empty world and filling it with life. At the start of the game environment's life, it will be devoid of human players and seem quite empty. So at first, the balance will be in favor of the computer-controlled entities; humans will be few and far between. This will consume a given level of resources, probably more than if the world is populated entirely by human-controlled in-game entities.

Shooting games like *Unreal Tournament* might only "start" if a complete team of players can be gathered together. In this case, it is quite easy to calculate the impact of a purely computer-controlled team and from there extrapolate the number of games that can be concurrently executed on the server. When the number of supportable games reaches the maximum, a decision must be made. The system must either refuse to start new games or degrade the performance ability of computer-controlled bots in order to conserve resources.

Again, AI can be used to reduce the load on the processor by intelligently downgrading the performance of the bots. This brings into play a problem: The algorithms used to evaluate resource usage and adjust capabilities accordingly also take up

processor resources—which should be kept to a minimum and applied across all in-game, computer-controlled entities. This sounds complex, but the implementation will likely be simpler than the explanation. In essence, all we are trying to do is to make sure that the players experience a good game while enabling as many as possible to play at once.

In these games, there are fixed rules for the number of players that can be present. But in more open game worlds such as *Eve Online*, *World of Warcraft*, *Everquest*, and even independent Web-based games like *Project Rockstar*, which are based on an ever-evolving game world, this might not be the case. For these games, the only limit is the processing power available, which must also consider that some of the offline players might have a continued online presence even after they have disconnected.

Also, there is a certain number of computer-controlled entities that will be required. These include those that are needed to manage the human-system interface at the points of interaction, such as shopkeepers and guardians of the in-game cheat police, and any other system avatars. In addition, a balance must be struck in terms of the resources that are available for the creation of in-game AI-controlled entities versus the required reaction time of the system. So the number of in-game AI entities could vary according to the number of human-controlled players and available resources.

The more AI entities there are, the more resources (processor, memory, etc.) will be needed to manage them and the way that they interact with the system (controlled emergence). If there are many AI entities that are being monitored to make sure that they continue to add value, then there will be less resources available for other processing tasks.

Keep in mind that human-controlled entities cost less in terms of processor power and other resources. After all, they are being controlled by an external entity with its own resources—be it a human interfacing with a game system or a bot running on a home computer. It may well be that the network-connection impact of these players far outstrips any impact that handling their in-game interactions has. So when implementing AI and A-Life solutions, designers have to be aware of this balance to be struck. They must be able to calculate the net effect of all of the AI in terms of available power and then choose the appropriate scheme for managing the behavior. Whatever behavioral management scheme is applied (from the many presented throughout this book, starting with the building blocks in Chapter 5), there will need to be a mechanism for controlling the size of the artificial population.

Managing the Population

The population will be divided into categories. First, there are those entities that provide part of the underlying tapestry of the game environment, like monsters that must be defeated as part of a quest so that the player can level up. These can be run on the server or via the client, but handling them through the client might open up exploit risks.

Next, there will be entities that are required to help the player interact with the game—from in-game policing to trade and other management tasks. These will

usually only be able to run on the server, as they will need the resources of the server to perform their tasks, and these entities are inherently more critical in terms of security.

Finally, there are those entities that fill the game world when an area has too few human players to make it worthwhile. These are the ones that are most expendable; they're only there for window dressing and as such can be relatively resource light and use simple behavioral modification and variation routines that stem from (for example) a basic state engine.

Each class might need to have its own population-control rules. In other words, the controlling AI could have a scheme that it uses to make sure that the system is operating within known limits by manipulating the computer-controlled entities. Those that serve as part of the game or environment are usually static. If they are killed, they are usually respawned (re-created). Some fixed life-span games might not require this, but any open-ended game will need to address intelligent respawning.

The number of entities needed to provide interaction with the game system, such as shopkeepers, should be consistent, also. They will be located at known points of interaction with the system and should probably be protected in some way.

Again, the game's storyline and/or environment might allow them to be removed and possibly respawned somewhere else. These entities allow the player (while remaining in character) to perform in-game actions, such as trade and/or communication of clues.

Finally, the entities that only propagate an empty environment in order to make it more interesting for the players will be entirely transient. There should only be enough to fill that purpose, bearing in mind that they consume resources in both behavioral modeling and management—which presents another interesting problem. The goal of these entities is to be as good as human players. Therefore, if the computer-controlled entities are successful enough to completely fool the human players, what happens if they suddenly disappear?

Like any other player disconnect, this might cause discord in the suspended reality required for the human player's belief in the game environment. So this needs to be dealt with as delicately as possible—perhaps a "phasing-out" of AI-controlled NPCs that are only there to bolster the population and provide an interesting environment for players.

Above all, each entity class will probably need different managing algorithms to control behavior. These techniques have been discussed in previous chapters, and the following presents new possible implementations for those algorithms. In the end, the choices will likely boil down to three generic algorithm types:

- Strict rule-based behavior: fast and reliable.
- Flexible rule-based behavior: more resource consuming and fairly reliable.
- Scripted behavior: flexible, resource expensive, and must be constantly monitored.

Choosing from these is a design decision that must take into account the available resources. In order to help make these decisions, the following discussion will expand on where these algorithms might be used in online multiplayer video games—beginning with strict rule-based behavior.

Strict Rule-Based Behavior

A strict rule-based approach is usually most appropriate for specific interaction points between the game and the player. While this might not seem very different from the examples in Chapter 8, the multiplayer environment raises some interesting AI questions, such as:

- Queuing or lockout (that is, only allow one conversation at a time)?
- Manage multiple interactions (or conversations)?
- Back-office management and action support?

Generally speaking, because the entity needs these kinds of support functions in order to perform its tasks, the front end will suffer. There might not be enough system resources available to make the entity smart, and because of its position as (perhaps) a gatekeeper, robustness and efficiency will win over intelligence.

The entity is not required to have a personality because its rules are set in stone by the game designers. In essence, all the entity has to do is follow orders and interact with the player in a restricted fashion. For example, shopkeepers need only offer their wares, take an order, let the player confirm it, and take the player's money. They are not expected to give any information or hold a conversation beyond their "shopkeeping" domain. Therefore, dialog can be constructed using simple phrase trees that leverage decision networks and other techniques for constructing chains of items—in this case, words to make sentences. At the artificial-life end, there is no need even for any Eliza-like AI that just turns the input around and spits it back out according to a predetermined set of rules. [DEWDNEY01]

As such, the actual outward behavior (artificial life) is not as important as being able to perform the task effectively and efficiently. So the states and rules that must be followed have to be robust and kept within a narrow domain of experience.

On the other hand, the risk of detracting from the player's experience is low because he will have fewer expectations. Of course, there are exceptions, especially in a free-roaming storytelling environment, in which an entity might serve dual purposes, such as both shopkeeper and spy. Under these circumstances, behavioral triggers (pass phrases, gestures, attire) can be used to change the way that the rules are processed, and this transition must be managed, too. However, it can still be based on strict rule processing, because again, robustness within a limited problem domain will be the main goal. In cases where the entity has to provide some kind of opposition or opportunity, however, a more flexible approach is required.

Flexible Rule-Based Behavior

Remember that our first foray into decision networks and finite state machines gave rise to the concept of generic behavior overlaid with customization matrices that changed the underlying behavior without substantially changing the success rate. This approach really comes into its own in the multiplayer environment, where (as discussed in the opening section) there tends to be more at stake than in single-player games. In addition, the multiplayer system will be exposed to a much wider

range of skill levels and, concurrently, than in the single-player game that can evolve with the player.

The flexible rule-based paradigm is useful for those entities that provide some kind of opposition, like the monsters that must be slain in a quest, for example. They must be realistically endowed with fighting capability, be able to use the terrain and circumstances effectively, and provide a challenge. On the other hand, they should not be too successful, because they need to be conquerable and (in many cases) help the player increase his skill level. Variable behavior is needed to avoid the possibility that players find tactics that guarantee success, even the less skillful among them, as this will alter the balance of the game.

Therefore, rule evaluation needs to adapt to the player and possibly even to his proven skill, either before the interaction or during it. The mechanism is as previously discussed: Take a default behavioral pattern and down- or upgrade its effectiveness to match the perceived circumstances. The AI required to do this could range from simply adjusting the entities' key attributes (strength, speed, weapons, damage, reaction time, available resources, and so forth) to dynamically select behavioral "modes" and constantly adjust the weights in the decision network to change behavioral traits. With this kind of adaptability ingrained in the routines, some artificial life begins to emerge, and the entity will enhance the player's experience. It could even be bestowed with some kind of personality that goes beyond the purely aesthetic. This technique could also be a cheap way to provide "population filler" entities to bolster a thin environment and make the game world artificially interesting for the players to inhabit, therefore making the game more immersive.

Given that the entity is operating in a multiplayer environment, there is also the option to use player behavior to "seed" behavioral modification algorithms. In a single-player game, this is equivalent to using player reaction times and other properties to seed random number generators and provide aspects of the behavior that are governed by the choice of random proportional weighted-decision matrices. Again, only the things that make the game more enjoyable should be included, because these algorithms will potentially drain vital resources. The saving grace is that rule-based systems are fairly robust and need very little supervision.

Because they are based on the evaluation of rules, this method is quite resource friendly. It is in the application of the modification routines to the behavioral patterns that resource usage is increased. An added complication can occur when several players arrive (either other players or NPCs) to do battle together as a team. At this point, scripted entities might become a better option, because they allow for more flexibility through the game engine interface.

Scripted Behavior

Scripted behavior is useful when flexibility is required that goes beyond building verifiable sets of rules to govern behavior. There is an inherent overhead associated with processing scripts in that they have to be evaluated before their effects on the game environment can be calculated. This is not new. Scripted AI in single-player

games suffers the same drawbacks, but since the server park cannot delegate a machine for every connected player (to execute the scripts server-side), there is the danger of a resource shortfall. The solution is to either split the workload between the clients and the server or be judicious when using scripted AI and artificial life in online multiplayer games.

Most AI and A-Life in multiplayer games tend to be based on scripted entities in realistic and successful simulations. The scripting can be at many levels, but the most common is simply to leverage information from the game universe and make informed decisions based on that information. For example, *Quake* bots make use of the open scripting interface, allowing players to create AI entities that are quite successful in the context of the game. They can move around, adopt tactics that take into account the immediate terrain, use mapping techniques, gauge effectiveness of weapons, and so on.

The flexibility will only be as good as the language that has been implemented, however, and support for variables (data storage), decisions, and repetition will all be required, as well as a hook into the game engine itself. More flexibility will be required in the multiplayer environment, simply because the entity will find itself in more varied situations.

Perhaps the final test of when AI and A-Life should be used is in cases when it is necessary for the player to believe (however temporarily) that the entity is controlled by another human, or when the entity has to exhibit all the outward appearances of being alive.

As a final note, in the interest of striking a balance between flexibility, robustness, and resource use, the scripting could be applied at a different level. Rather than being provided at the behavioral interface—such as for moving, interacting, or decision-making—the scripting could be used at a level above that and the result of the script's execution applied to a flexible rule-based decision network. This takes advantage of the fact that state machines and rule-based decision networks are robust in that they are unlikely to provide behavioral deviations, thanks to the limited number of states or actions. They can also be proven—or at least every possible combination of events foreseen and tailored to prevent unwanted behaviors and much easier than complex scripts.

They do, however, provide outward flexibility in reaching their goals, thanks to a weighted system in which courses of action are selected according to relative weights of paths through the network. Collections of these networks work together and interact with an ever-changing environment, leading to emergence that exhibits lifelike behavior.

For example, assume that the AI controlling a squad uses planning at the strategic level (to determine the route) and then terrain- and cover-awareness at the tactical level, followed by rules of engagement at the lowest level. These can all be managed using rule-based AI algorithms. In a multiplayer environment, there will be a constant need to re-evaluate these rules based on the unfolding events. In addition, each set of rules can be changed in response to events (such as making the squad scatter under fire—a change to the rule-based flocking algorithm, perhaps), which will, in turn, have an impact on future evaluation of those rules.

Summary

The key points to take with you from this chapter are how AI and A-Life can affect the differences in the level of control that the developer can expect in the game as it is unfolding. In the multiplayer environment, emergence will be strong, and therefore it is necessary to stick with AI and artificial life routines that are self managing. Resource pressures in the real-time processing environment will also have an impact on the kinds of behavioral algorithms chosen, and it will be necessary to estimate and control resource usage for the different kinds of in-game entities that must be modeled.

The introduction of many human players will result in strong emergence, so it will be more important than ever to be able to detect and rectify issues that relate to in-game entity behavior. After all, the multiplayer game is the ultimate testing bed for AI algorithms, and it is almost impossible to predict every possible event or sequence of events. Possibly the biggest compliment the developer will ever receive will be that the players cannot tell when a bot or A-Life construct deployed in the game is computer controlled, and they mistake it for another human player.

References

[AIBLOG01] http://www.ai-blog.net/archives/000114.html.

[BLUMBERG01] http://www.gamasutra.com/features/20060216/blumberg_01.shtml.

[DEWDNEY01] A. K. Dewdney, *The Armchair Universe*. W.H. Freeman and Company, New York, 1988, p. 82. Eliza was one of the first AI-controlled conversation bots, written in 1966 by Joseph Weizenbaum of MIT. In the words of A. K. Dewdney, "ELIZA avoids a great deal of conversational burden by playing the role of a nondirective psychotherapist."

This page intentionally left blank

CHAPTER

10 THE APPLICATION OF A-LIFE OUTSIDE THE LAB

In This Chapter

- Simple Genetic Algorithms
- Breeding Behavioral Patterns
- Datasets versus Functionality

This chapter is about reducing the entry barriers when using A-Life in production-level video games and balancing the inherent risks with the rewards that can be reaped from using artificial life techniques. Many video game designers and developers shy away from AI/A-Life, which they view as largely experimental techniques. In addition, development schedules are always high pressure. Therefore, AI and (especially) A-Life are often put aside as deadlines grow near. This is natural. After all, artificial intelligence/life entails complicated features that are prone to errors, difficult to debug, and hard to balance within the context of a video game. In addition, AI and A-Life can tax an already burdened machine-resource pool—processor, memory, and graphics subsystem. Therefore, our solutions seem destined to balance in favor of easy-to-implement, cheap-to-process, pure rule-based systems. These typically have AI that ranges from initially impressive but easily beaten to relatively poor implementations. A-Life barely gets a second look, except in those games that prescribe its use because there's no other option.

However, the three core issues (experimental technology, hard to implement, resource intensive) are becoming less daunting. In fact, this book has presented ways to include AI and A-Life in a bottom-up fashion that will specifically deal with these three issues. We have also shown that many games use A-Life–style techniques without really trying to take advantage of features like emergence. They have AI that, with a little extra work, could be improved quite a bit for little or no extra overhead.

Now we'll look at some simple, low-impact, low-risk ways in which these techniques can be deployed.

SIMPLE GENETIC ALGORITHMS

Much of the use of A-Life and genetic algorithms in video games is for providing life-like variations from standard patterns as a way to counteract predictability—either to provide better settings or to enhance the challenge of gameplay. In other words, A-Life in a game like *Spore* provides the environment and challenge, whereas in almost any first-person-shooter (such as openly scripted games like *Half-Life*), the behavioral patterns might be changed according to GA principles in order to create a more unpredictable challenge for the player (within certain constraints).

A-Life (and in particular, genetic algorithm) routines in video games are based on well-understood AI principles. They need not be terribly complex or involved, but they must add value to the game or differentiate it from all other games, such as in *Black&White 2* or *Spore*.

In this chapter, we look at GA in two specific roles—as providing variance to static rule-based and scripted systems, and as a way to build behavioral patterns in their own right. This could be seen as classic dataset GA versus genetic programming. Both need to adhere to the guiding principle that a line must be drawn somewhere between the detail required of the AI implementation and the cost to create it. The modeling (physical and/or behavioral) needs to stop somewhere, if only because the deeper into a system the modeling goes, the more processing power is required.

There is no sense in modeling a soccer player's feet, boots, ball contact, and so on directly; instead, the system can decide that the ball will have spin on it when struck and just apply that spin to the trajectory without modeling the process that actually led to this behavior. It's a simple example, but every game will have similar or possibly more complex ones.

Any AI and A-Life that is deployed in a game needs to keep the previously mentioned "line" in mind, because anything beneath it is hard coded. Put another way, the cutoff line represents the limit of our flexibility; anything underneath it cannot be changed (because it's not modeled), but everything above it can be changed dynamically.

A real-world example of this is *Black&White 2*, in which small rules lead to emergent behavior as they are evaluated within a given context. These rules might not be modifiable in themselves (they are "under the line"), but there are examples of where the associations between them can be altered. For example, the interface allows the player to intervene and change the associations between rules that alter the behavior via reward or punishment. The result is that the behavior of the entity employing the rules is changed.

Taken one step further, this could also be done automatically using GA; that is, a system can be created that alters the associations themselves. Of course, our previously discussed AI techniques need to be applied so that the system is aware of what "good" and "bad" behavior is so that the associations can be changed accordingly. This approach is not limited to binary decisions, of course, but to deploy a wider-reaching system, it is first necessary to do some groundwork.

Starting Out

The principal idea when using A-Life and AI together is to find more than one acceptable solution to a problem—the most appropriate algorithm, or the most appropriate behavior to apply to an algorithm. The choice of algorithm combined with the application of that algorithm is then carried out during the play session. The key is that the developer knows and *understands* the algorithms that have been developed, because they are part of the original game design. All that the A-Life part of the solution does is to manipulate those well-understood algorithms. It is not important how the candidate solutions have been achieved, whether through traditional design and development methodology or via some more esoteric route, including those offered here for creating solutions using AI and A-Life, such as dynamic script creation.

We assume here that rule-based AI is being deployed, with little flexibility in the rules but the option to select rules for application in the gaming context. This does not necessarily mean that the behavior is scripted. It might be hard coded, but there is the ability to selectively execute code. Without this option, even the most basic A-Life techniques can not contribute to the behavior unless that behavior can be modified using input data. This implicates one candidate solution for some aspects of the game.

Other traditional AI solutions—such as FSM-based models, be they crisp or fuzzy, networked or isolated—can also serve as candidate algorithms. Previous chapters have illustrated how they can be manipulated to subtly change the way these models are applied. Expert system-based solutions, where chains of simulated questions and answers lead to one of a possible collection of outcomes, can also serve as candidate solutions, although in these cases, it might be desirable to interfere with the decision-making process at several points (weighting the final outcomes) to provide varying behavioral patterns. Some outcomes might favor the player, and some might favor the computer, depending on which candidate algorithm is being evaluated and in what context.

The deployment of AI/A-Life in a game begins with the premise that the desired outcome must occur in a way that is variable. This might not mean that it is entirely random, but it could be geared toward a more specific behavior under certain circumstances. This is important for several reasons. Since A-Life deployment is *on top of* existing, well-understood candidate solutions, it is necessary to keep the selection process as low-impact as possible. "Variable" behavior means that each unit in the collection will have the same basic behavior but will differ in details.

Take fighting units in a medieval war simulation as an example. Each will have basic behavior governed by a set of (possibly inflexible) rules, the evaluation of which will lead to an outcome—successful or not. Starting from the lowest order of modeling, these behavioral patterns can be broken down as follows:

- Animation reflecting behavioral action—different warriors will perform each action (cut, thrust, defend, run away) in a different way.
- Variable per-action behavioral model—different actions might be designed to achieve the same general goal (again, cut, thrust, parry, and so forth).
- Variable tactical action choice—different possible tactics (stab and riposte, parry and thrust).
- Strategic plan—defensive, offensive, cowardly, or rash.

This list is presented based on a single entity. The behavior of the mob needs to be taken care of, as well. The emergent behavior that will occur when they are placed together can also be managed using behavioral patterns, by defining rules to control their interactions and thereby dictate how a collection of entities presents itself. At this level, too, the same kinds of techniques can be deployed—breaking down the desired group behavior into directed per action, tactical, and strategic groups—but for the sake of simplicity, our discussion will stick with single-unit/multiple-action modeling. Furthermore, we will usually discuss these in isolation, except where interaction can be used to illustrate a specific technique. When the techniques are deployed in real game designs, it quickly becomes apparent where the group can be treated as a single unit and its behavior managed as such. It will also become apparent where this is not possible, and the group behavior will be left to emerge from a series of interactions with the game environment. An entire book could be written solely on emergent systems in video games, but our goal here is to suggest ways in which they can be deployed and managed on a grass-roots

(below the cutting-edge) level—hence the slight simplification when discussing how behavioral modeling can be affected by artificial life algorithms.

In addition to behavior, there is the actual appearance of in-game units—the way that they look and are animated. There is a relationship between the way that a unit will move and the way it is constructed (*Spore* and *Darwinia* are great examples). A-Life can be deployed to make sure the units do not look the same, which will also have an effect on how they move. In addition, themes can introduce their own variety, so that different types of the same unit will have different static appearances, such as skin tone or hair color/length.

All of this can be used for games ranging from soccer simulations to space combat, RPGs, and MMORPGS. The behavior of each entity can be handled at both the individual and group levels and provide not only variations in behavior, but learning via new, more successful behavioral patterns that result by design or chance (genetic algorithms).

Humans are the ultimate opposition for a video game. They can learn faster and adapt existing experiences and information more readily to new situations than can a computer system—even one as complex as a video game. This means that humans can predict with stunning accuracy what will happen next. A game that does not address this by providing memory (learning) or variable behavior risks being substandard among the evolving, cutting-edge games. Artificial life is what can make these techniques so useful. It does not introduce variances at random but deploys them based on the experience of the player and current game session situation.

The technology deployed need not be cutting edge, as long as its application is innovative—that is, "outside the lab." Our emphasis is on tried-and-tested AI rule-based solutions (as is deployed in many games), which is modified using genetic algorithms and artificial life. It is important when deploying simple genetic algorithms to choose the right model from the start and map the data that provides the input and output in the model to values that can be easily manipulated using GA and A-Life techniques.

Decision Networks

A decision network is a collection of connected nodes, and each node point provides some form of decision or action that leads to another decision or action, which might be the start (looping network) or end (definite network) of the behavior process. The terms "looping network" and "definite network" are meant, respectively, to distinguish a network that repeats itself continually from one with a discrete end. Both can be found in many video games—from the *PacMan* ghosts to guard behaviors in modern FPS games.

From our decision-network starting point, a template can be customized by entity instances in order to provide variations in behavior. Bear this in mind, and the discussion that follows will reveal some powerful A-Life techniques.

The three kinds of decision networks will be discussed, any of which can be definite or looping. Typically, some lend themselves to one or the other and are more likely to be used as either definite or looping decision networks.

- State-based (FSM)
- Rule-based (expert systems)
- Associative (neural networks)

A state-based network will usually loop, even if only intermittently, meaning that there may be a pause between the processing of one action or event and the action or event that causes the network to reset itself. In other words, there might be a resting state that is only activated as a result of something occurring (an activation event), but the network itself never needs to be explicitly reset. It just oscillates between resting and running. This will become clearer when we look at rule-based systems.

Typically, rule-based networks tend to be definite; they have start and end points, and the path that leads from start to end is designed to reveal one of a possible set of outcomes (like in an expert system). This is usually done within one time slice, whereas a state-based network might span several time slices. These rule-based networks therefore need to be reset before they can be used again or run once in a discrete time slot. The ability to "reset" is a requirement, because a video game is normally a continually executing process. Interruptions can occur at any time, and if a process takes a certain length of time to execute, then it may well find itself unable to complete its task.

Unlike some other programs, there is never really a time when a video game is doing nothing. The exceptions are those puzzles that wait for the player to move, like chess—and even in puzzle games, the computer could conceivably be processing its next moves in the background while the player is thinking. Therefore, the system needs to know where it is in a given process (which may be suspended at any time) if that process transgresses its time slice. For a state engine this might be easy enough, so long as the state transition is sufficiently fast. For other, longer processes, a reset might be needed instead.

Resetting puts the "program counter" back to the start of the network in cases where the network cannot be completely executed in one game time unit. Without a reset function, the context would be lost each time the network was suspended and resumed. This is applicable to both state- and rule-based networks, but it is not true of associative networks, which store their state as part of their datasets. In essence, an associative network is a living mechanism in which the data is a critical part of that mechanism; without it, nothing can be achieved.

Rule-based and state machine networks tend to be static, although we have already seen some ways in which these static underpinnings can be augmented, sometimes by layering associative networks on top. Associative networks are often overlooked because they store their context data as part of the model. They are one of a class of neural networks that take their cue from our understanding of the way that the human brain works—by association. In an associative network, all the nodes are interconnected; other than that, they are just neural networks.

A neural network loops in the sense that it usually evolves continually, but it is also a definite network in that it is usually executed to arrive at a decision in one unit of game time. So this makes them very useful in video games. Any changes to

the context of a neural network are stored in the weights and node values, as well as the other properties that govern neural network mechanics. Modifications can come from within or without and will effect change in the network's behavior. We'll concentrate on simple, chain-style networks that are easy to implement in tandem with traditional AI algorithms.

Markov Chains

Our example here is a modeling one and is key to the next section on detailed implementation of a genetic algorithm starter kit. The modeling technique has been used to evaluate and create datasets based on observation. By this we mean that a population is examined, and new, presumed instances are created based on the observed characteristics of the entire population.

A Markov chain, also known as a Markovian linked list, is a simple concept that is the base of one of the most effective, simple AI and genetic algorithm techniques. It can be considered a simple neural network, in which nodes are attached to each other by varying strengths of association, which influences the chance that a specific path to a neighboring node is chosen. Figure 10.1 shows a schematic of a Markovian network. The nodes represent things like actions, pieces of data, or characteristics of a specific entity. The percentage values represent the chance that a path from one node to another is chosen.

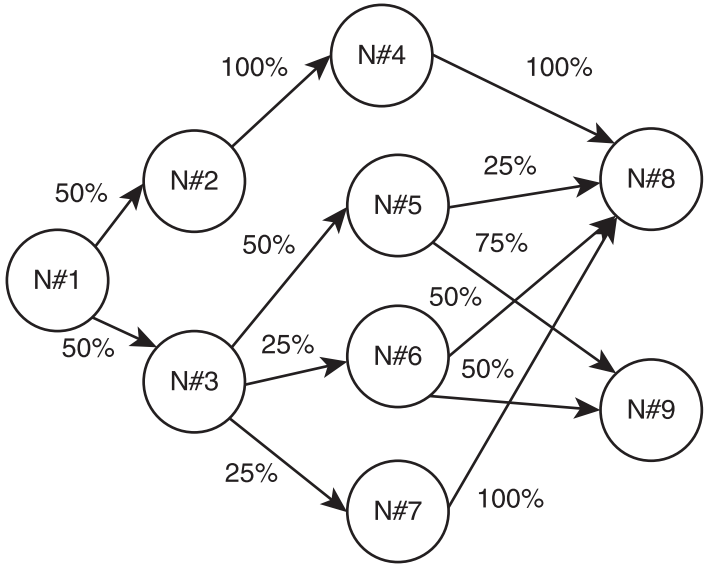

FIGURE 10.1 Markovian network.

The paths represent the only possible passages through the network. A node may have one or more paths possible from itself to another node, or it may have none at all. The sum of all the paths from a node must be zero (path end) or 100 percent. (As a side note, the difference between a Markovian network and a simple weighted decision network, as discussed in Chapter 5, "Building Blocks," is that a Markovian network is designed to be built from input data, whereas a weighted decision network is created by design. All of these techniques—neural networks, finite state machines, Markovian lists, and so forth—are, of course, related.)

The network is generated based on the analysis of a set of data. This set of data is a population of known good examples of combinations of nodes. The analysis gives rise to a network of nodes, each one of which is assigned a value derived from the dataset. In addition, each node links to the other nodes in a way that is governed by the probability that these nodes exist side by side in a given dataset. This ensures that impossible (or at least, illogical) combinations of nodes do not contribute to the resulting network, and that the network is weighted toward the most common combinations.

The resulting network of nodes can then be used to generate a dataset from the probabilities by linking the nodes together and generating a chain of values. The algorithm starts at a given input point and follows the path through the network by evaluating the percentages at each decision point. As each node is encountered, a value is output from the algorithm, and this value represents the value of that node.

These values can be data units that are used directly, or they can have a coded meaning (interpreted as an action). They can also be directly applied within the code—that is, they're linked to the code directly, to the extent that they can be script functions or even compiled code that is untouchable after the application has been built.

One application is to use hard-coded Markov chains that are created from the analysis of input data that generates the computer code. This could be as simple as a populated array of data that is then supplied to the video game in much the same way that trigonometric lookup tables are used in spatial coordinate calculations. The twist is to use GA to modify the hard-coded lookup table to produce deviations from the chain. In this way, when the chain is rebuilt with variations, it will be different from the input data, but representative of it.

The classic example is to use Markov chains for generating names from letter tables. This technique is referenced in many books, including *Infinite Game Universe: Mathematical Techniques* [LECKY01]. There are two ways to implement this in code: as a two-dimensional array of numbers in which each number represents the strength of the association between two items, or as a linked list of nodes in which each link has a number associated with it. The latter is clearly more flexible and provides a reusable solution. The linked list is also capable of producing networks of linked nodes that are rather like neural networks, so we shall concentrate on this solution. Also, a linked list is useful when it is not known how many items will likely be required in the final collection of nodes. When mapping the datasets that represent the game design to the abstraction that will become the data table for in-game use, it might not be clear exactly how many variations and values are going to be needed, so flexibility serves in this respect, too.

We assume that the resulting collection of nodes will be implemented in C++ as a collection of structures. Each node must consist of a value and a collection of links to other possible nodes. This might look as follows:

```
struct NODE
{
    char cValue;
    LINK ** oLinks; // Array of pointers
};
```

The `LINK` object needs to have a relative strength (either a counter or some kind of calculated weight factor based on 1.0) and a `NODE` object as a target in order to create a weighted network. If repetition is allowed within the network, then `NODE` could end up as the "parent" of its own `LINK` object. In other words, the node would point to itself. But we cannot dispense with the `LINK` object in these cases, because the weighting factor is still required. The `LINK` object might be defined as:

```
class LINK
{
    double dLinkStrength; // Association strength
    NODE * oTarget; // Pointer to a node
};
```

Typically, there will be a start node and an end node (which will have no value), providing an entry and exit point to the network. It's up to the designer to decide whether recursive links will be allowed, though as a safeguard, there should be a hard limit on the recursive depth. The network (at this point, just a chain) can be contained in a C++ structure, as follows:

```
struct CHAIN
{
    NODE * oHead;
    NODE * oTail;
};
```

Populating the list requires that the algorithm analyze (in our word-generation example) a set of words. Each word is read in and broken into characters, and each letter is added to the network, either increasing the strength of association between two nodes or adding completely new nodes to the network. At the end of a word, the node that represents the last letter is linked to the end node (`oTail`). This sequence of analyze and update is repeated for as many words as the dataset contains.

Generating a dataset from the network is a two-stage process. First, the network has to be stored somewhere. In a hard-coded solution, a program uses the network to generate a piece of code that reestablishes the node values (letters) and weights

that connect them (`LINK` objects). When the code is compiled, the network is then reconstituted. However, any modifications (based on additional training) cannot be permanent unless the new network is written to a datafile. The design of the video game will dictate whether this additional functionality (save/retrieve) is necessary.

Generating words from the list is an easy proposition if no double (or more) letters are allowed. All that is required is to start at the null node `oHead` and select one of the links at random, with the decision weighted toward those links with higher proportional weights. The network is traversed, and the values of the nodes at each step are recorded as letters in the resulting word. Once a predetermined word length is reached, the next letter that goes to the end node, `oTail`, must be chosen, regardless of whether or not it has a high proportional probability.

If repetition *is* allowed, then the process is only slightly more complex. It will be necessary to know what the maximum repetition index is for each link that references back to a specific node. For words in English, the letter-repetition index is likely to be two at most; few words have more than two identical letters in a row. To set a repetition index, another number can be added to the link for cases in which the letter is allowed to link to itself. Furthermore, it is possible to continually train the network by inviting a human to rate each word for its "naturalness" as it is presented. As each one is rated, this information can be fed back to the network, thus training the weights so that the general quality improves. Changes are then made on the basis of invented words, creating a clear advantage over the original dataset used to prime the system.

Before we abstract this model to include other ideas, it is worth remembering that there are a number of settings in the Markovian linked list that can use AI or genetic algorithm techniques. These could be specified in the design phase (and equally well in other abstractions not related to creating natural-sounding words), and they can be set algorithmically. Classic AI techniques or some hybrids involving genetic algorithms can take the place of the human analysis and adapt settings accordingly. These settings (for word generation) include, among others:

- Minimum and maximum chain lengths,
- Numbers used to pick letters, and
- Repetition limiter.

Some of these choices might have an obvious impact on word generation but no impact on other areas. Our minds have been trained on so many word patterns that we are quite good at spotting words that just feel wrong in our own language. Other data sequences that the human mind has not been trained on will not as often catch inappropriate chain lengths or repetition indexes. However, these sequences might be subject to other restrictions that need to be handled in the same way, and these are design decisions.

So while we noted that the English letter-repetition index is likely to have a maximum of two, this might not apply to other forms of data and is part of the reason so much care has been taken to explain the underpinning mechanics of the Markovian linked list. An example of an alternate form of data with a higher maximum index can occur in the behavioral Markov chain.

Behavioral Markov Chains

Rather than using items of data, such as letters, Markov chains can string behavioral items together. Each value is mapped to some facet of behavior exhibited by the system, such as a state change trigger, a decision point, or a direct action. This uses tried-and-tested technology, so the result should also be effective; however, the result will only ever be as good as the input data. If a particular behavioral model—such as an FSM implemented as a script or another form of behavioral pattern—is known to only include *possible* behavioral outcomes, then any generated behavior will follow those same lines. On the other hand, if the source data includes every combination of behavioral state change, even those that are illogical, then the resulting trained network will invariably include relationships that ought not to exist. Therefore, as with all neural networks, training is one of the most important aspects of the implementation.

The advantage here is that behavior can be less precise than when forming words from letters, so the approximation is as good as the input data. Many cases do not require quite as much data training; it will often be possible to train the network by example and extrapolate behavior from there—or even hand-design the weighting in the Markovian network.

The amount of training required and the overhead of the initial design will depend on how many nodes (state values) and combinations (Markov chains) there are in the system. The beauty of the Markovian system described here is that, once implemented, it can be extended very easily and therefore can be implemented very early on in the development cycle. The system can also be applied at many levels.

The basic unit is pairs of nodes—one linked to another, although one node can link to many others—but this can be augmented by isolating pairs followed by additional nodes. This makes a triplet, which necessitates some minor adjustments to our previously described two-dimensional network. In essence, in order to represent triplets as a pair followed by a single node, the implementation requires two networks that can communicate. One network stores the possible pairs, and the other stores an index to each pair followed by single nodes. A better solution would be to make nodes represent pairs, with multiple pairs linked to single-value nodes. The caveat is that the network will grow exponentially as pairs are added, as there will be several instances of values representing different combinations of pairs. In the two-network solution, both the strength of association and possible pair combinations are stored in a network that contains only enough nodes to represent the total number of single instances.

Whichever method is chosen, the theory is exactly the same as in our word-generation example. Input data is examined and then used to create a lookup table that can then be used in the program code to generate behavioral chains. This lookup table can be either directly coded or living within the system.

The Markov approach also means that the result will never include two items that have never been placed side by side in the source data. Of course, this assumes that the training data is perfect. It follows, then, that every combination (chain) of behavioral units that make up the behavioral pattern is true with respect to the

observed behavior of the input stream. This makes it a powerful technique for copying another player's behavior and also has some uses in learning systems.

Statically coded datasets have an additional set of advantages. Because they can be complied into the game code, they save on external resources, reduce the possibility of cheating (via game file hacking), and are inherently faster than dynamic networks.

Animation and Appearance

Of course, behavior is only half the story. Other aspects add to the A-Life experience, which also require some measure of AI in order to achieve a convincing result. These include the animation of individual entities or collections of entities' static and dynamic appearances (visual effects, modeling, textures), as well as movement.

Again, there are many traditional techniques that can be applied in a cutting-edge way to increase the game experience for the player. In the interests of a low-impact solution, these are usually techniques that can be layered on top of existing tried-and-tested technology.

Animation

The ways that the in-game character models carry out actions can be varied by using techniques that employ animated chains. This allows the designer to define possible action sequences and combinations and store them so they can be replayed in a variety of different ways.

At the easiest level of implementation, a straightforward network (such as previously described) can allow the creation of action chains based on pure probability; we just redefine the letter data as action identifiers. This could be refined by allowing overlays of probability values per entity type, modifying the basic probability network by specifying offsets to change the weights on each link in the network. Thereby, different entity types will perform actions in different ways, because the generated chains are created based on different probabilities.

Again, the power of this technique is that variety can be implemented without introducing improbable or impossible action sequences into the chain. As a bonus, we can bestow true character on different entity types, which will enhance the player's experience as he learns the likely action sequence for each type of entity. However, in keeping with the nature of the video game, A-Life ethos of removing predictability, we must always maintain the chance that the entity will do something surprising.

This technique of layered networks can be applied at different levels for the animation and action parts of the game engine. Only one behavioral chain network is stored, along with sets of offset weights for each of the links that represent variations from the default behavior *per entity type*. At the lowest level, each movement could be carried out in a way that varies, depending on the character—for example, an ogre stabs slowly and hard, while a dwarf stabs fast and with less force.

At the medium level, movements can be predefined for each character type or class and strung together in a series of action sequences. Or they could be altered

according to different levels of other criteria, such as aggression or health. Perhaps the designer of a fighting game decides that a damaged opponent would be more likely to carry out defensive movements, rather than attack offensively. Implementing this would just require a layer of weighted offsets applied to the Markovian behavior network.

It is the chaining together of action sequences that becomes the selection process. Based on the input of various examples of action sequences, the network is created. Perhaps these could even be recorded from a play session in which a skilled player took a specific character class through a variety of movements, either alone or against an enemy. Again, it is the training of the network that becomes important in rendering lifelike behavior.

Of course, the application of artificial life also means that the behavior network can be altered during the play session, perhaps based on the observation of ineffective action sequences. All that is required is to use AI techniques to subtly alter the weights in the network.

At the highest level, the selection sequence just strings together strategy, based on objectives (go to a given point, fight, run away, and so forth). At this level, all of the underlying action/animation sequences can be predefined or varied algorithmically. This algorithm might vary behavior according to a strict selection process with a guaranteed acceptable result, even if it is not particularly lifelike and is somewhat predictable each time.

If the level of sophistication warrants it, and if there is time in the production schedule, this technique can be applied at every level and provide a very rich, lifelike series of animations. Some might be purely aesthetic, or some might have an impact on the game (such as, put a sword or gun in the character's hand and attack the player). Even if the applications are purely aesthetic, they will have very little cost in terms of additional design and development, while enhancing the experience. This is, after all, the goal when using artificial life in the video game.

Take, for example, the after-race animation that drivers in *World Rally Championship II Extreme* perform. The camera moves inside the vehicle, and the driver and co-pilot go through a series of predefined movements that are the same every time—cheering for a first-place finish and head shaking for anything else. While not particularly Earth shattering or important in the grand scheme of things, it does begin to grate on your nerves after a few races.

Since all the groundwork already exists, a quick end-of-project task can add some variation to these movements. A collection of animations can be defined for the models representing the driver and co-pilot and replayed in different sequences, depending on the outcome of the race. Again, this could work at several different levels—individual movements or sequences—depending on the level of sophistication that the production timetable allows. These variations would not serve any great in-game purpose, but it would add an extra dimension of lifelike behavior to the in-game characters.

The basic implementation of the *selection process* is the same at any level and can be anything from a simple Markovian linked list to a full-fledged multilayer network that alters itself in response to the gameplay. Let's look back at the simple

driver animations; they can be varied according to the difference between the last race position (or average race position) and the current race ending position. Using this data, the characters could look ecstatic at gaining a third-place finish after finishing tenth on average and be complacent with a string of first-place wins. This is another example of a low-tech solution applied in an interesting and novel way to increase the A-Life content of the game—and then there is the appearance of in-game entities.

Appearance

A whole book could be written on this subject alone. There are many different facets to modeling the appearance of in-game entities—facets that depend on the game, the modeling techniques used, and the available processing power. As such, we will assume in this discussion that the in-game characters are humanoid. However, these techniques can be applied to any entity, whether machine, creature, or planet—as long as it has a number of possible defining qualities. For the sake of simplicity, we will assign our humanoids a set of easily identifiable qualities:

- Hair color
- Hair style
- Eye color
- Skin tone
- Shirt style and color
- Pants style and color

The list could be longer, but the above provides quite enough for illustrative purposes. If our set of humanoids is split into two tribes—say, Pirates and Settlers—then it is also possible to define a set of possible values for each attribute. For example, all Pirates might have black hair, or all Settlers might wear long pants. What is important is that the sets of data are sufficiently disparate as to allow the player to identify a Pirate or Settler visually.

Next, the designer can link combinations of attributes together globally. For example, there might be an attribute network (following the same lines as the previously described behavior networks) that handles physical characteristic combinations. This network of probabilities will allow certain combinations and exclude others. So there might be a chain that allows:

Red Hair ➔ *Blue Eyes* ➔ *Pale Skin, but precludes*
Red Hair ➔ *Black Eyes* ➔ *Dark Skin.*

These networks need to be built up for the various attributes and on a global level. At the end of the process we might have a network that dictates some of the attributes for Pirates as a Markov chain and the same for Settlers. Then there would be a collection of networks (probably only two) that allows combinations and probability values for physical characteristics on the one hand and clothing combinations on the other. Generating a character then becomes a two-stage process. First, select

the key characteristics by generating a chain from the appropriate network (Pirates or Settlers), and use this chain to populate the other attributes. For example, the initial chain for a Pirate might be:

Black Hair ➔ *Torn Shirt.*

From this, there are two secondary chains to create, using the global networks for physical characteristics and clothing. It might be that Torn Shirts are either white or gray, with 75 percent of them being gray. These might then be linked with only Black Trousers, so 75 percent of the time our Pirates would be clothed in Gray Torn Shirts and Black Trousers.

These techniques, while simple, can be applied to any set of characteristics. Moreover, they can be altered by using genetic algorithms in interesting ways to generate new populations from existing ones, and these are fed back into the game system. In this way, a player can influence the selection criteria and always end up with his Settlers being clothed in the selected color, or be able to breed out certain hair colors. To do this, we need to discuss genetic algorithms in more detail.

Applying Genetic Algorithms

It is easy to explain where genetic algorithms are important in relation to Markovian generators. If a Markovian generator is used to create an entity from a properly trained Markovian network, then it represents an entity that may or may not be from the source population. In other words, the resulting chain might be a mixture of various input chains, or it might be an exact replica of an existing chain from the input (training) data.

The smaller the number of variables used in building the network, the more chance there is that the generated entities will be copies of those from the training dataset. This dataset is the source population that was evaluated in order to produce the network of weighted links and nodes. For example, if a dictionary of common words is used as the source, then many different nonwords can be produced. There will also be a tiny population of common words alongside invented ones. If, on the other hand, sentences are used with a limit of, say, 10 words, then the chances of re-creating one of the input sentences is quite high. By the same token, the chances of creating an entirely new word sequence (sentence) might be reasonably low. These proportions will change, depending on the number of input sentences used as the source.

The Markovian generator, then, can be used to generate populations of varying sizes and qualities. Each node in the network might be a piece of data to be applied in the algorithm, making the entity behave or look a certain way; it does not matter which. What does matter is that these effects are generated more or less at random, based on sequential probability. The possibility of generating two identical or even related entities is therefore negligible. Of course, if pseudorandom techniques are used, then potentially the same two sets of entities could be created from scratch, but it would be impossible to re-create any one of them at will without reseeding the generator.

A related factor is that the network remains stable, and the resulting population cannot influence the generation of future examples in any way. Genetic algorithms solve this by allowing the programmer to select from the population of entities according to some criteria, and then merge them to create new examples that are then thrust into the general population. The most successful will survive.

So the Markovian network can be used to generate a string of node identifiers that can then be manipulated using genetic algorithm techniques. The power of this technique is that the input can be static (at the start of each game), and depending on how the game is played, both the Markovian network and population of possible entities (strings of nodes representing behavior, appearance, or something else) can be in a constant state of flux. They will react to the player's interaction with the game. If there are multiple players, then these interactions will be very complex, but the goal of the artificial life in this case is to adapt to whatever is thrown at it, based on a predetermined, known good set of solutions (from which the Markovian network had been generated).

Applying genetic algorithms is not as tricky as it might sound, and the benefits can be quite large. It is essentially a form of feedback into a network, but it can be performed in a low-impact way when coupled with the networks discussed earlier.

The next section deals with how behavioral patterns can be generated using genetic algorithm techniques. This "genetic algorithm starter kit" is designed to help you more fully understand the relationship between data and algorithms.

Basic Application

At its simplest, a genetic algorithm applies a set of data to a possible solution algorithm. Remember, however, that GA also includes genetic programming, which is the creation of command sequences (programs) that are candidate solutions. So the basic application of GA and Markov chains generates datasets from Markov chains (or Markov networks) as a chain of data to be applied to an algorithm that represents a (probably not optimal) solution. This solution might just be an algorithm that exhibits some form of behavior.

For example, in Chapter 8, "Several A-Life Examples," we described a possible nonoptimal solution for the location of ships on a Battleships-style grid. The basic algorithm is simple enough—test-fire on locations from top to bottom, with spacing of n cells, leaving a space of m cells between lines of fire.

In this case, n and m are the two pieces of data that are important, and the application of genetic algorithm theory assumes that there is an optimum combination of values that can be determined by trial and error. The trial-and-error process simply tests combinations of values at random, keeps those that prove effective, and discards those that do not.

Effectiveness in the case of the Battleships-style game might simply be a record of the average number of attempts it takes before a shot hits a ship. Those combinations of n and m that show a below-average number of missed shots can be kept. Over time, as grids of varying configurations are analyzed using the algorithm, the sets of n

and *m* that prove effective can be retained and used to create new combinations—such as by holding on to the *n* and varying the *m*, or vice versa.

At the end of the training period, a set of possible solution data will be collected and can be deployed in the game. This process can be repeated for all possible seek-and-destroy algorithms until a workable collection of settings has been found. However, it is possible to take this a step further with more complex datasets, and the results can be fed back into the network.

Advanced Applications

A step up from the basic GA application is to extend the generation capabilities to create both datasets and command sequences in the genetic programming style. Sample programs can be fed into the learning system and used to create a workable set of possible command sequences in the scripting interface's structured language. As before, those outcomes that are successful are kept; but this time, instead of adding them to a collection of possible solutions, they are fed directly back into the network, thus reinforcing the paths through the network by increasing link values. This is similar to the feedback provision in neural network training.

This can also be extended to self-training systems that learn by example. They may make a lot of mistakes along the way but will eventually become quite intelligent. The training can be done during either the development or play session, or even during the testing phase.

Having embraced this technology, you should now be ready for the final part of the A-Life paradigm as far as video games and this book are concerned: digital DNA and breeding solutions. It sounds daunting, but is really no more difficult to understand than the rest of this chapter.

Breeding Behavioral Patterns

The second application of genetic algorithms in video game AI involves selection, mutation, and crossover to create behavioral patterns for in-game automata. These three principles are derived from evolution theory; existing digital DNA sequences that make up the characteristics of a member of the population are used to pass on good qualities, suppress bad ones, and introduce random variations in the hope of creating more effective members. We shall call this process *breeding*, and it is an integral part of many artificial life implementations. Remember that the actual information being bred will be used for different purposes, depending on whether or not it is applied as GA (data fed to algorithms) or GP (genetic programming sequences of commands, or "programs").

The digital DNA and breeding principles are the same. Take two parents and recombine their digital genetic material so that the result is a combination (or perhaps mutated combination) of the source. There are a number of ways that this recombination can be done. The one chosen for the purpose of this discussion is known as genetic crossover.

Before we look at crossover, however, there are a few things to bear in mind. First, the behavioral pattern's representation must be open to genetic algorithm paradigms. Pure source code implementations, in which the algorithm is fixed at the time of compiling and building the game, will limit the use of genetic algorithms. This is because only the data that is fed into the static algorithms can be changed; the state changes (for example) themselves cannot be modified. On the other hand, the sequencing can be altered by using a suitable driver module that can arbitrarily call the compiled functions, based on an incoming stream of data.

To use AI and A-Life outside the lab in a noncutting-edge, safe, low-impact way, this might be exactly what a game needs. It is a good compromise when dealing with the complexities of genetic programming and allows the flexibility of genetic algorithms.

State-Based Genetic Crossover

The first way (arguably) to apply artificial life and digital genetics outside the lab is by using genetic algorithms to produce variable behavioral patterns in order to counteract some of the predictability that AI models sometimes produce in video games. The starting point is a state-based, static AI system. The classic FSM approach from Chapter 5 would be a reasonable model to follow. To recap: The FSM allows an entity to move through states in an orderly fashion, based on triggers from the system. The sequence of events in the classic FSM remains fixed, although variations can be introduced when several options are available, and these are chosen according to a weighted decision-making process.

State engines of this kind are good at providing a behavioral model that fits predetermined notions of acceptable behavior. This might be at a fine-grained level of control (individual movements) or a coarser, higher level, such as selecting a behavioral "mode" state and chaining modes together. If two sets of behavior can be isolated (two FSMs), then they can be connected with a third FSM that can select between them, creating a network of acceptable behavioral transitions. Alternatively, we can select between different FSMs, based on a pseudorandom or mathematical probability–based selection process. This gives us three possible ways to use FSM networks for providing variable behavioral patterns.

There is another way that is low impact and can be added at any point in the development cycle. However, as frequently stated in this book, adding artificial life (via genetic algorithms) later in the development process is frowned upon, even provided the groundwork has been properly prepared.

It is assumed that the process used for behavioral scripting allows for the sequential processing of a set of commands, and this sequence can be altered externally. In other words, the behavioral models are manipulated in a scripted fashion, including the conditional processing of script commands and passing data to them where necessary. Without this preparation, the following approach will not work; it

needs to be able to hook into the sequential calling sequence that provides the basic behavioral model. This might take the form of a plain text file that is accessible to all, as in *Quake*, or it can be a closed binary model.

Each state or state change must be encoded as a specific function in the source code—a unit of code that can be called at will from within the correct context, be it a C++ class or some other mechanism that packages state data and appropriate methods into a single unit. The abstraction of the state changes must then be encoded correctly so that the changes can be interpreted during the play session. It is this abstraction that provides the genetic data that will be used as part of the genetic algorithm process.

This is, therefore, a bridge between genetic algorithms that manipulate data and genetic programming that manipulates underlying algorithms. It is a GA-like approach because it operates on a dataset that will be applied to an algorithm (the script engine), but has a GP-like impact, because the result is a miniprogram.

Now that two sets (there could be more) of state change sequence data (along with appropriate triggers) have been established, the next stage is to perform genetic crossover of the units so that a child with some of the behavioral characteristics of each parent is created. The two (or more) parents can be created as part of the design process, of course. In fact, it would be natural to use this as a starting point, rather then begin in chaos with a collection of completely randomly populated state-change sequences.

This is slightly more involved than the straight genetic crossover paradigms seen in previous chapters. The possible pitfalls are mainly caused by the paradigm's design, which is to modify behavior, rather than simple attributes.

First, being state based, it is necessary to retain the triggers that cause state changes and congruence with the various states that the entity might find itself in. This is really a design issue, because it is at the design level that the initial state changes are worked out. This means that if there is no way of getting from state A to state F (because they do not share the same trigger in context), then a problem will exist, because the state change will never occur. There are two solutions; the hard one is to use a network of state changes in a Markovian-style behavior network. This way, only possible state changes will ever be represented in the generating network. However, it will be difficult to visualize how the data used to create the calling sequence is converted to that sequence. Instead, we assume that the collection of state changes are all possible neighbors in any state chain. For this reason, the design must ensure that the sets of possible state changes are compatible. Given that they are, Figure 10.2 illustrates a simple schematic of state data.

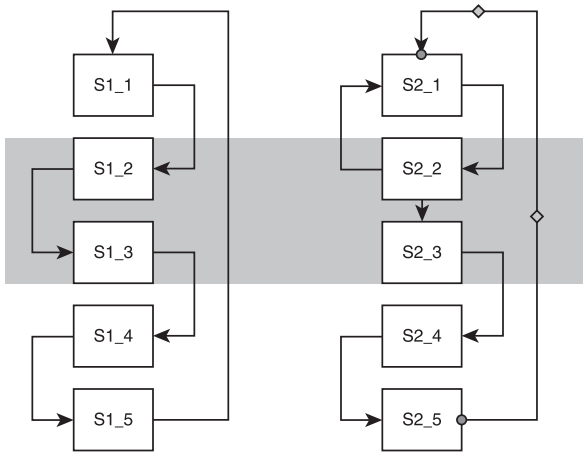

FIGURE 10.2 Simple state digital DNA.

The gray box in Figure 10.2 indicates the four elements that have been selected to participate in the genetic crossover operation. In this case, the choice is purely arbitrary, but in a real implementation, they should be selected according to a suitable algorithm. For example, they might be specific areas of self-contained functionality or some other logical block.

Each of the boxes labeled S*x_n* represents a state. The *x* is the DNA chain that it belongs to, and the *n* refers to the specific state. Later on, this will help us identify the units of execution that have been swapped.

The arrows represent triggers that can cause state changes. In instances where there are multiple arrows, this indicates that there are multiple triggers. Being based on an FSM, these triggers cause the entity to change from one behavioral mode to another. For example, S1_2 has one possible state-change trigger that causes the entity to go from state S1_2 to S1_3, whereas S2_2 has multiple triggers—back to S2_1 or on to S2_3. The diagram might look complex, but it is a simple example. Real state-change diagrams can be much more complex, but they can usually be broken down into smaller sections that can be combined together.

The principle of genetic crossover is implemented when the units of digital DNA are swapped from one chain to the other, changing their order in the process. The data units are *crossed over* from one set of data to the other. Assuming that the two chains are identical (have identical parents), this would result in S1_2 being swapped with S1_3 (and vice versa) and S2_2 being swapped with S2_3. So far, this is much the same as other genetic crossover operations.

Triggers complicate the issue on the question of whether the chains are left *in situ* or swapped along with the states themselves. It is possible, although not always wise, to leave the triggers in place.

The answer will depend on several factors, and the design should reveal which path to take. In general, if the triggers cannot be used in the context of each state

change, then they must be swapped. However, in cases where the triggers can be applied to any of the state changes, they can be left in place. Figure 10.3 shows the children of identical parents with triggers left in place. All that has changed is the order of the states. It is here that we must be careful that the triggers make sense in the context of their newly assigned states. If not, and they never fire, then the FSM engine will stall. As an added note, it is also possible to swap only the triggers that cause the state changes, thereby altering the way that the behavioral model reacts to external events. The same caveats exist as for the association of trigger to state change.

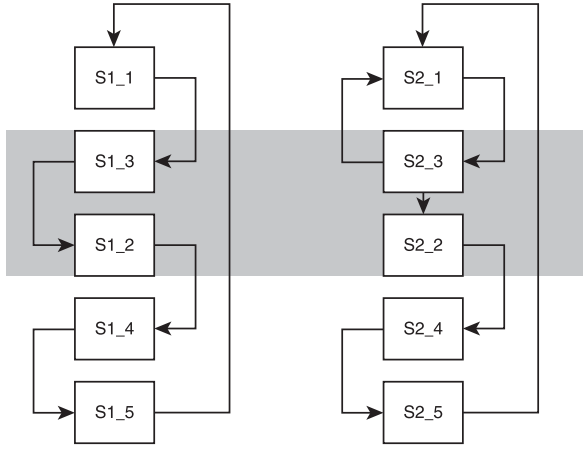

FIGURE 10.3 Children of identical parents (*in-situ* triggers).

Clearly, if state changes are swapped blindly, there is the risk that the new triggers no longer make sense. The state definitions might not allow for a specific trigger to occur in the context of a given state, based on the part of the behavioral model that it is supposed to represent. For example, in a soccer simulation, if S1_2 is "dribble with the ball," and S1_3 is "shoot on goal," there might be a trigger based on the goal being in range. Swapping the two units and leaving the triggers intact will yield an invalid result. The trigger from S1_3 to S1_2 would now be the test for the goal being "in range," which does not make sense in the new context of dribbling with the ball.

The new scenario might then be as follows: The player is running from one location to another (state S1_1). The ball is passed to him, and the trigger action "ball capture" causes a state change to "shoot on goal" (S1_3 after crossover). Now, at this point, either the ball will be fired at the goal or the player will move within range (trigger) and then move into the dribbling state (S1_2). Normally, the correct sequence of events would have the player dribble to the goal and then shoot. This flow makes no sense, because the triggers are probably unavailable to the new states. The player might end up dribbling around forever without ever shooting at the goal.

Figure 10.4 shows the children of identical parents with swapped triggers. At first glance, it may seem as if there is little difference between the noncrossover and crossed-over sequences; after all, the triggers are the same, and the states are the

same. If the order changes, then it would seem to only serve to shuffle the behavior. However, on closer inspection, you can see that the entry and exit points have changed. This is especially evident in the right-hand sequence, which, when hand-executed, reveals that the behavior is quite different from the original.

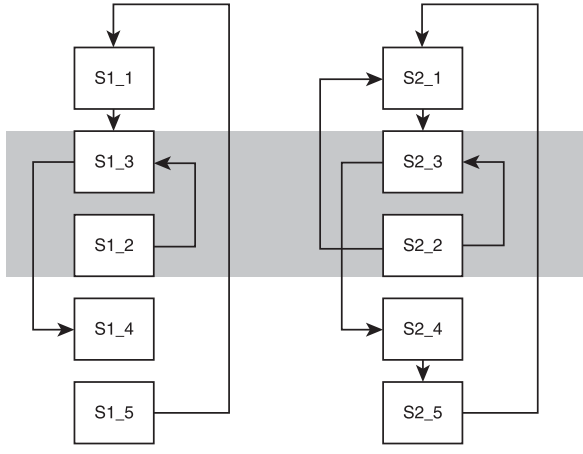

FIGURE 10.4 Children of identical parents (swapped triggers).

Now the congruence remains, and it is only the sequence before and after the swapped sequences that is different. This might be trivial, but genetic algorithms are all about making little changes one at a time and testing the results. Big changes tend to lead to unacceptable, strongly emergent behavior that is more difficult to control.

It is also possible to perform genetic crossover with nonidentical parents and adhere to the same principles as before regarding trigger and state congruence. The nonidentical parents should be both from the same behavioral area, although they might be from different classes of entity.

Figure 10.5 shows how the state units (again with or without the triggers) can be swapped to yield a child from two nonidentical parents. In this case, the child will end up with some of the behavioral aspects of one parent mixed with aspects of the other parent.

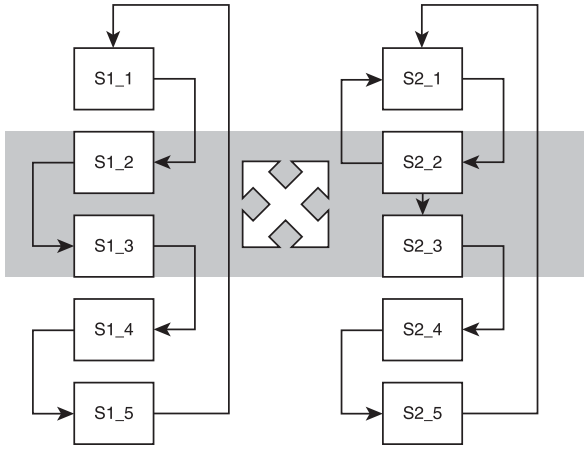

FIGURE 10.5 Nonidentical parents.

In this case, since the parents are not identical, it is probably even more important to consider whether or not the triggers should follow the state units from one entity to the other. They probably should; but again, it is a decision that follows no hard rules beyond the requirement that triggers remain serviceable in their new context.

In addition, because the parents are not identical, it is possible to create multiple children using different pieces from different parents. However, for the sake of simplicity, assume that in the following expansion of the crossover process, we start with the S1 framework and mix in bits of S2 to yield a single child. The algorithm chosen to start the crossover (with S1 or S2) or to create one child of each kind will be dependent on the game design. Other aspects, such as the length of the sequence to select and at which position the sequences should begin, are also design questions to be answered on a project-by-project basis.

Following the sequence of state changes from top to bottom and swapping the items in the gray area (from Figure 10.5) lead to the following state block sequences:

S1_1 ➔ *S2_3* ➔ *S2_2* ➔ *S1_4* ➔ *S1_5, or*
S2_1 ➔ *S1_3* ➔ *S1_2* ➔ *S2_4* ➔ *S2_5.*

Next we will assume that the output triggers are being swapped—in this case, one that flows away from a state. In other words, for a unit that has two output triggers and one input trigger (like S2_2), one output trigger will be remapped to the first unit of the child block, and the second trigger will be remapped to the next unit in the child block. The input trigger (which is the output trigger from S1_1) remains unchanged; it causes a state change to the second unit of the child sequence, which, after the crossover, just happens to be S2_3. So while the state and trigger remain in context, the newly mapped state has changed. This is called *output trigger retention*, which is explained further in the next section. The result of applying this is a hybrid of Figures 10.4 and 10.5, shown in Figure 10.6.

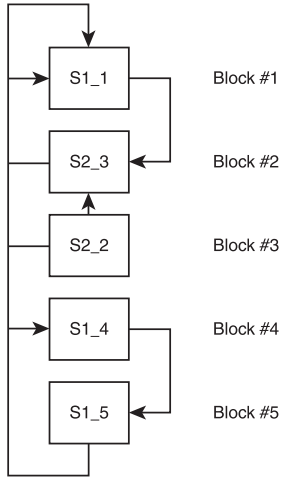

FIGURE 10.6 Genetic crossover child S1_A.

To arrive at Figure 10.6, the first thing we do is create a list of output triggers in terms of the source state blocks and their targets, and the position of the target block in the sequence map. This yields a collection of triggers, as follows:

S1_1 ➔ *Block #2*
S2_3 ➔ *Block #4*
S2_2 ➔ *Block #1 and S2_2* ➔ *Block #3*
S1_4 ➔ *Block #5*
S1_5 ➔ *Block #1*

Next, the blocks are placed in the appropriate order and the state change triggers added to the resulting sequence map. The result is Figure 10.6, complete with in-context triggers retained through the specific algorithm.

Trigger Retention

It is important to remember that there are three possible modes of trigger retention:

- Unswapped triggers,
- Swapped output triggers, and
- Swapped input triggers.

Unswapped triggers might not be congruent with the state contexts, because they are left in place when the state changes are swapped. The new states might not make sense in the context of the trigger, or vice versa. We saw this in our previous example and chose to rectify it by using input or output trigger retention (by swapping).

With swapped output triggers, the input flow of triggers will be lost because the output triggers are swapped along with the state changes. Since the context is retained for the triggers, it is likely that correct behavior will result. Whether that behavior is better or worse than the parent behavior is something that must be measured.

For swapped input triggers, the input flow will be retained, but the output flow will be lost. Again, retaining the context for the triggers will ensure congruence to a certain degree, but less than when output triggers are retained.

Experimentation might be the only way to choose an appropriate algorithm—a time-consuming venture if done by actually playing the game. And it might not be possible to simply experiment. If bottom-up development is used, then there might not be enough of the game environment available to experiment with. Some kind of scoring methodology might be needed in the design phase to automatically determine the best approach. Of course, if all the states and triggers are of the same context and make sense no matter what state they are applied to, then it just becomes a question of what works best.

It is also interesting to note that for the preceding example, if input trigger retention is chosen, Block 2 in the child S2_3 becomes isolated because it receives input from Block 2, which, after crossover, becomes itself. This may or may not be acceptable.

The Purpose of Crossover

Crossover creates new examples of behavior that can be used in the play session. Simply copying the blocks between identical parents, for example, would not create any variation in behavior. Mutation can also be used—either as part of the crossover process (by randomly crossing pairs of swapped blocks) or by explicitly changing a block's meaning. As with pure crossover, the goal is to create an entity with a new digital DNA sequence for evaluation by the game engine. This might just be to produce some AI variation, but it might also be the creation of more successful versions of the AI. In games where there is time to learn and evolve during the play session, this can be an interesting cornerstone to the behavioral AI.

Multiple crossover followed by simulation (or actually playing the game) will yield results that can be compared with some kind of yardstick, and the less capable behavioral variations can be discarded. This yardstick will depend on the kind of game being designed, but it must have a concrete measure of success. This is known as the "survival of the fittest" paradigm, made famous by Charles Darwin and his research on the theory of evolution.

Survival of the Fittest

The survival of the fittest paradigm solves the problem of selecting candidate children from multiple possible parents (themselves children), who will serve as parents of future generations. In the natural world, the strongest survive and the weakest perish, simply by the force of nature. In video games, it is necessary to perform some form of simulation to establish fitness criteria and then use that criteria as a benchmark. The whittling away of less successful children can be done on the basis of real

experience within the game environment or expected experience based on sandbox simulated behavior.

The paradigm can be applied during the development phase to build up behavioral models for the game, rather than designing the models from scratch. It can also be applied in the game itself. However, the game environment might not always grant the system the luxury of allowing multiple mistakes in evolution process. This is where simulation can be used to predict which offspring are worth saving.

The survival paradigm can go either way; depending on the criteria, it is possible to degrade performance as well as improve it. This could be useful in certain gaming environments where the system becomes too good for the player to overcome, therefore hurting longevity because of an improperly balanced difficulty curve.

Genetic Rewriting

In addition to using genetic algorithms to change the probability that scripted events will lead to a specific decision path, genetic algorithms can actually create event sequences. In other words, it is possible to create entirely new behaviors from scratch. This is higher impact in terms of design, development, and testing—though not, however, impossible (as we saw in Chapter 8). It is just tricky to apply and more time and resource expensive. The simpler the language and solution required, the easier it will be to generate meaningful event sequences. The key here might be to break the problem down into segments that are as small as possible.

As long as there are rules that govern how the pieces fit together, these can then be used to create strings of behavior that are entirely new, as part of the genetic reproduction. Again, the actual DNA is just data that can be replayed through an engine that interprets it. It is an abstraction of the code, rather than being the code, itself. So it is possible to begin with a collection of possible actions (state changes, triggers, and so on) and fit them together in a multitude of ways. The trick is to keep recombining and sieving the ones that seem to be most successful and use them to create new entity models that are then used to breed more successful examples. In this way, a body of useful behaviors can be built up.

Applying GA to GA

Since the application of genetic algorithms requires decisions to be made about how the GA paradigm will be applied, this can also be part of a GA selection process. In a sense, the system can choose which parts of the GA paradigm to apply, such as length of sequences, swapping points, and mutation. In fact, the Battleships example from Chapter 8 is a good example. The search-by-fire function uses a static value for m and n, the spacing between shots (vertically) and lines of fire (horizontally), respectively. However, genetic algorithms could be applied to dynamically vary these values, once the initial optimum has been found, perhaps as a function of some other value in the system, such as number of lines completed or shots fired.

In a sense, GA is automated trial and error. The processes of selecting from the possible genetic combination and mutation criteria, as well as the extent of mutation

and the lengths of genetic substrings, can all be subject to GA selection. Equally well, other AI techniques could be used to choose these values, such as rule-based models. However, remember the caveat from Chapter 8: Genetic substrings can only be as short as can be recombined while maintaining congruity. For example, if a scripted AI engine is available, then it will likely comprise a number of different possible constructs. This being the case, there are a number of areas where care must be taken not to separate code blocks that have logical dependencies. For example:

- Not separating decision clauses (`if…then…else`)
- Not splitting up multiple branches (`switch…case`)
- Not splitting function blocks (`function…end`)
- Not splitting repetition clauses (`for…endfor`)

Clearly, if an FSM network is used or rule-based AI that includes chains of linked decision points, there are other constraints, although they will tend to be more flexible than scripted AI. Nearly any mechanism can be subjected to rearrangement using these techniques, as long as data can be arrived at that represents the structure.

However, calling trees used in genetic programming will also be more flexible if the goal of the GA is dynamic scripting. Treating a dataset in GA as a script is better served by implementing the whole system as genetic programming, rather than just a series of data blocks. The difference between genetic algorithms and genetic programming might still be slightly confusing, so the last section of this chapter will discuss a low-impact GP approach to GA.

Datasets versus Functionality

This final section is designed to clarify a few points regarding the application of artificial life and other AI-based algorithms in terms of video game engines. Until now, we have bandied about terms such as "data" and "entities," as well as "attributes," "programs," and "scripts" without really nailing down exactly how each one relates to the application of genetic algorithms.

Again, in the interest of a low-impact solution, we assume that the developer has decided to enhance an existing design with artificial life techniques. That is, there is already an underlying AI system in place that is appropriate to the game environment, and it is open to easy manipulation. Generally speaking, AI behavior in video games can be defined in terms of two aspects:

- Datasets (data that feeds algorithms), and
- Functionality (prebuilt source code-defined behavior).

Both can be used to select behavioral models—that is, modify the behavior of the system or choose between existing behaviors, but in slightly different ways. They both also have advantages and disadvantages.

The dataset approach has, for example, the disadvantage that the data needs to be accessible to the system; therefore, it might also be accessible to the player, leading to

a security problem. The data can be compiled into the system, of course, but this means that it will be reset each time the game application is executed. On the other hand, data is easy to customize and change, even during the play session, and the resulting modifications can be made permanent. After all, most if not all gaming platforms allow the data to be saved on rewriteable media (memory card, hard drive), but this does introduce the possibility of external manipulation.

Data is also incredibly easy to generate and apply, and it is very easy to leverage A-Life using genetic algorithms, as we saw in the preceding section. Somehow, leveraging genetic programming via a code tree to create code or even scripts is much more difficult to deploy, and for this reason, many designers choose to adopt the dataset route. However, there are a few things to bear in mind when using data as a decision-processing function, as we shall see.

Functionality represents the actual algorithms, whereas the datasets are typically used for modifying existing algorithms. However, there is significant crossover between the two aspects of GA, as we have seen—for example, in cases where an algorithm is, in fact, a scripting engine. Most solutions will use both aspects in tandem in order to provide the most flexible and easy-to-implement environment for using AI and A-Life together.

Datasets

Security is the first issue relating to the use of datasets versus functionality, and this issue can be solved, at the risk of losing some flexibility, by compiling the data into the application as was noted. This helps reduce the chance that the data can be altered outside of the gaming environment. Of course, clever encoding (perhaps encrypted) will also help solve this but will also increase the processing overhead. Since the main objective of using this form of A-Life is to provide an enhanced experience without substantially increasing the processing overhead or complexity (the "low-impact" solution), encoding might not be a valid solution. However, it might be worth the extra investment just to be able to dynamically alter the dataset and keep those changes permanent. This allows the system to evolve not only with the game session but also between sessions and continue to provide immersion by remembering (retrieving) its reactions to the player's input. This not possible for some platforms in which the data is transient or stored on a permanent medium, unless some form of nonpermanent storage facility is used, such as a memory card or hard drive.

A mix of approaches can also be used. One example is the "default plus offset" method, which uses two sets of data. The compiled-in dataset provides the "default" behavioral model, and the transient data provides a logical offset from that behavior. It is generally the transient dataset that is modified.

There are two ways in which this can be used. The first way is to apply the data as it is to the algorithms. This approach is appropriate when the algorithms use input data directly—how hard, how fast, how much to the left, how many units to build, and so on. These algorithms might be equivalent to functions or methods within the application source code.

The second way in which a dataset can be used is as a decision-making aid. This approach is generally appropriate when the algorithms either have no direct use for data or take their data from the actual game environment, or when the only input data that makes sense is that which is supplied as default. In this case, the data is used to select between algorithms, like a percentage choice or other mechanism (including weighted pseudorandom selection, as seen in Chapter 5). The chosen algorithm may need, in turn, to be supplied with additional data for it to complete its processing, and this data might come from another dataset.

A dataset can just be a collection of individual items—numbers, letters, codes, and so forth. This is the easiest way to use a dataset approach, because it fits in well with neural networks and other mechanisms. Arrays and other data structures make the process even easier to apply.

Applying neural networks or other AI/A-Life processing techniques to a collection of numbers that represent the input dataset is very low impact, and it can be reused. As long as the implementation is nonspecific, it becomes part of the learning process that can be reapplied in future projects. It is completely separate from the game environment itself, and the numbers are meaningless when taken out of context—that is, their meanings change, depending on the context in which they are applied. Therefore, a layer of mapping must be provided for each game that deploys them, but the underlying technology need not change.

Finally, a dataset can be easily recorded, modified, played back, and manipulated. Think of this in terms of being able observe another player's movements, record them, and use them in a neural network to increase the relationships between certain links and diminish others. Sometimes, however, this is not enough. Often, simply processing data does not enable the flexibility that a video game requires, or it becomes too processor intensive to be practical. At this point, it becomes necessary to change strategies and move away from a known collection of functions whose links/choice is governed by a static algorithm. In addition, the static algorithm will eventually run the risk of settling down to a predictable pattern. Being able to manipulate the actual functionality becomes a more interesting solution, even if it does add a layer of complexity to the solution.

Functionality

Much of the preceding discussion can also be applied to modifying the functionality—that is, all the statements relating to the permanence of the changes, such as external versus internal representations, and therefore hold true for all types of data, including executable files and scripts. There are two ways to modify the functionality. One is to adopt a scripting interface that allows the designer to build scripts that can be modified in an AI/A-Life way. The other is to allow the application source code to be built via artificial means during the design and development stages and then compiled as the final executable game application. The same problems and pitfalls apply to both approaches—mainly relating to testing, debugging, and making sure that the behavioral excesses can be identified and checked.

Modifying the functionality is dangerous—it has its risks. The premise for using datasets (all possible behaviors predefined) is lost when moving to an open behavior model. Since the algorithm has the potential to create invalid instances of entity scripts, guarding against this occurrence is a large technical task. During the design and development stages, this might not matter, as corrections can be made manually; but once the game is shipped, nothing more can be done except by means other than actual development.

Using an entirely open approach has some of the same security issues. When we allow the player to change the scripts (or write his own), this offers exciting possibilities and provides a very flexible interface for automated adjustments. On the other hand, safeguards must be built in to make sure that the game, in its original state, can be reconstituted with the same properties that the designer intended. Corruption can occur due to malicious alterations, cheating, or errors on the part of the gaming system when reweighting the decision trees that create the scripts. Therefore, a backup strategy for restoring the game is advisable.

An alternative is to make sure that script changes cannot make the in-game automata any less successful at providing help/opposition. This level of control was discussed in Chapter 2, "Using Artificial Intelligence in Video Games," as well as Chapter 9, "Multiplayer AI and A-Life." Whether this route is practical or not depends on the type of game, target platform, and many other variables.

Finally, genetic programming techniques can be used to directly alter the functionality at several levels. These include algorithm selection (equally well handled by datasets and GA), calling tree modification, and direct script modification/creation. These have all been covered elsewhere in this book, including in some Battleships-style game examples in Chapter 8. It should be clear by now that, while these models are relatively easy to implement, the more complex the game, the more difficult they will be to control.

SUMMARY

By now you are aware that this book does not subscribe to the notion of "bolt-on" solutions, but not everyone who deals with advanced AI/A-Life theory gets it. Therefore, expecting video game developers to plan for and insert this relatively new technology into their game designs from an early stage may, in some cases, be asking too much.

Of course, developers of games like *Spore* and *Multiwinia* or any game that utilizes digital genetics and learning by example, such as *Black&White 2*, cannot avoid using these advanced techniques. For everyone else, AI and A-Life might seem like unnecessary complexity for an already complex system.

However, this chapter has suggested a way to develop artificial life and digital genetics with genetic algorithms as a separate project from the main game—that is, as something that can be integrated into the game via existing technology. Even if the developer has not decided to deploy AI/A-Life from the first design brief, as long

as the underlying technology allows for the manipulation of data, these techniques can be added relatively late in the development cycle.

The level of experimentation that is required to leverage them, therefore, is much reduced. After all, the game *works*, and the various bits and pieces that make it tick are already implemented and tested. The addition of artificial life just ought to be the icing on the cake.

Hopefully, this experience with add-on solutions will help you, as a developer, embrace advanced AI and artificial life solutions more completely in future projects or, at the very least, position you to continue adding the A-Life magic to your gaming products.

REFERENCES

[LECKY01] Guy Lecky-Thompson, *Infinite Game Universe: Mathematical Techniques*. Charles River Media, June 2001.

This page intentionally left blank

INDEX

B

C

D

E

F

G

H

I

J–K

L

M

N

O

P

Q–R

S

T

U

V

W–Z

Journal of Game Development

The *Journal of Game Development* (JOGD) is a journal dedicated to the dissemination of leading-edge, original research on game development topics and the most recent findings in related academic disciplines, hardware, software, and technology. The research in the *Journal* comes from both academia and the industry, and covers game-related topics from the areas of physics, mathematics, artificial intelligence, graphics, networking, audio, simulation, robotics, visualization, and interactive entertainment. It is the goal of the *Journal* to unite these cutting-edge ideas from the industry with academic research in order to advance the field of game development and to promote the acceptance of the study of game development by the academic community.

Subscribe to the Journal Today!

Annual Subscription Rate:

Each annual subscription is for one full volume, which consists of 4 quarterly issues, and includes both an electronic and online version of each *Journal* issue.

$100 Individual
$80 for ACM, IGDA, DIGRA, and IEEE members
$300 Corporate/Library/University (electronic version limited to 25 seats per subscription)

For more information and to order, please visit, **www.jogd.com**.
For questions about the *Journal* or your order, please contact Emi Smith, **emi.smith@cengage.com**.

Call for Papers

The *Journal of Game Development* is now accepting paper submissions. All papers will be reviewed according to the highest standards of the Editorial Board and its referees. Authors will receive 5 free off-prints of their published paper and will transfer copyright to the publisher. There are no page charges for publication. Full instructions for manuscript preparation and submission can be found online at **www.jogd.com**. The *Journal* is published on a quarterly basis so abstracts are accepted on an ongoing basis.

Please submit your abstract and submission form online at **www.jogd.com** and send the full paper to **eic@jogd.com** and **emi.smith@cengage.com**.

For questions on the submission process, please contact Emi Smith at **emi.smith@cengage.com** or Michael Young at **editor@jogd.com**.

www.jogd.com

This page intentionally left blank