All Online Books

Table of Contents
View as Frames

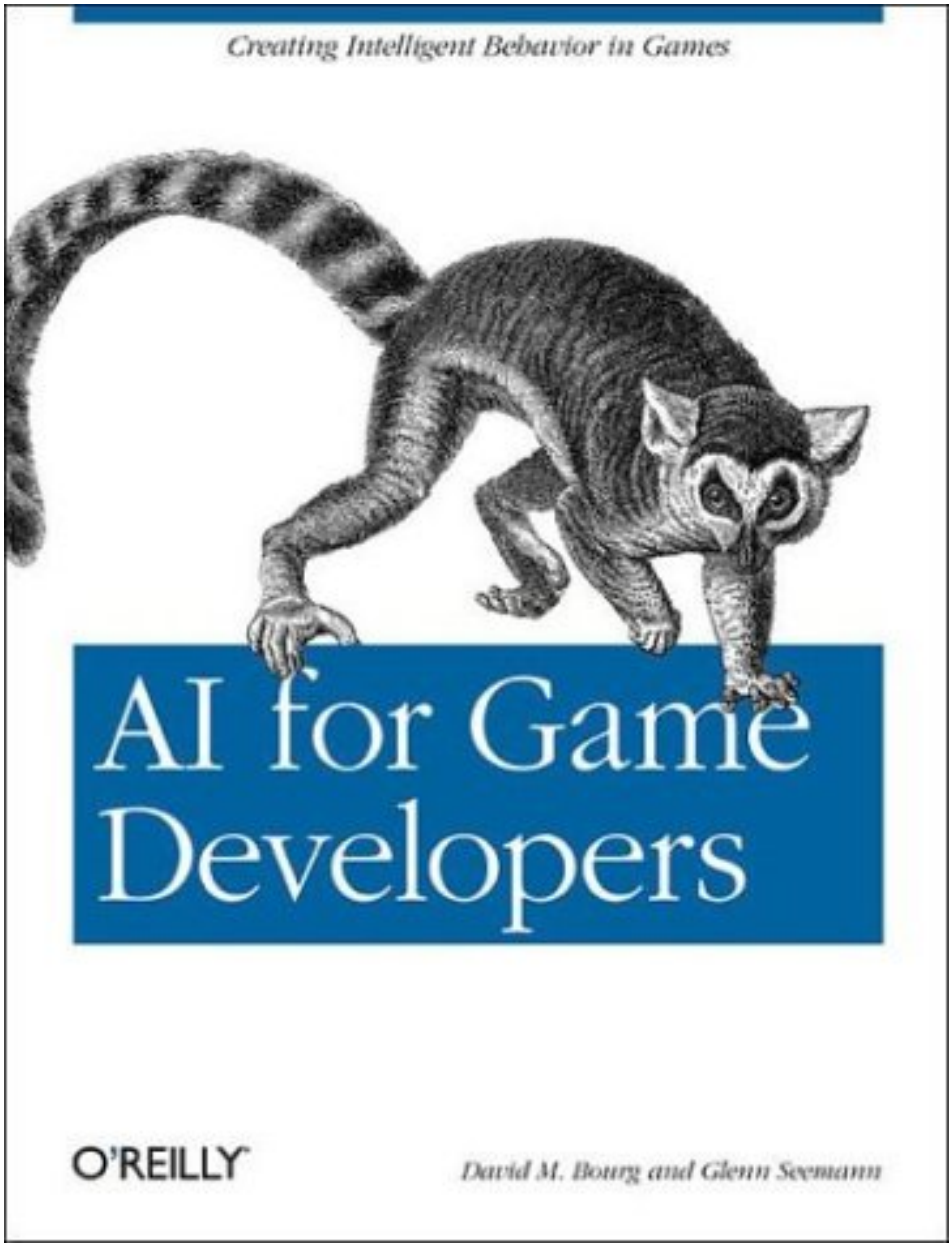


Table of Contents
Reviews
Examples
Reader Reviews
Errata
Academic

AI for Game Developers

By David M. Bourg, Glenn Seeman

Start Reading

Publisher: O'Reilly
Pub Date: July 2004
ISBN: 0-596-00555-5
Pages: 400

Written for the novice AI programmer, *AI for Game Developers* introduces you to techniques such as finite state machines, fuzzy logic, neural networks, and many others, in straightforward, easy-to-understand language, supported with code samples throughout the entire book (written in C/C++). From basic techniques such as chasing and evading, pattern movement, and flocking to genetic algorithms, the book presents a mix of deterministic (traditional) and non-deterministic (newer) AI techniques aimed squarely at beginners AI developers.

All Online Books

Table of Contents
View as Frames

Previous Next

Preface

Recent advances in 3D visualization and physics-based simulation technology, at both the software and hardware levels, have enabled game developers to create compelling, visually immersive gaming environments. The next step in creating even more immersive games is improved artificial intelligence (AI). Advances in computing power, and in hardware-accelerated graphics in particular, are helping to free up more CPU cycles that can be devoted to more sophisticated AI engines for games. Further, the large number of resourcesacademic papers, books, game industry articles, and web sitesdevoted to AI are helping to put advanced AI techniques within the grasp of every game developer, not just those professionals who devote their careers to AI.

With that said, wading through volumes of technical papers, text books, and web sites can be a daunting task for upcoming game AI developers. This book pulls together the information novices need so that they can get a jump-start in the field of game AI development. We present relevant theory on a wide range of topics, which we support with code samples throughout the book.

Many general game development books cover AI to some extent, however their treatment of the technology tends to be limited. This is probably because such books have to cover a lot of different topics and cannot go into great depth on any particular one. Although several very good books do focus on game AI (we list many of them in the "Additional Resources" section of this Preface), most of them are geared toward experienced AI developers and they focus on relatively specific and advanced topics. Therefore, novices likely would require companion resources that cover some of the more fundamental aspects of game AI in more detail. Still other books cover some specific game AI techniques in great detail, but are restricted to covering just those techniques.

Our book covers a wide range of game AI topics at a level appropriate for novice developers. So, if you are new to game programming or if you are an experienced game programmer who needs to get up to speed quickly on AI techniques such as finite state machines, fuzzy logic, and neural networks, among others, this book is for you.

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

Assumptions This Book Makes

Because this book is targeted for beginner game AI developers, we don't assume you have any AI background. We do, however, assume you know how to program using C/C++. We also assume you have a working knowledge of the basic vector math used in games, but we have included a brief vector math refresher in the Appendix in case your skills are a little rusty.

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

About This Book

We didn't hope to (nor did we attempt to) cover every aspect of game AI in this book; far too many techniques and variations of techniques are used for an even larger variety of game types, specific game architectures, and in-game scenarios. Instead, we present a mix of both deterministic (traditional) and nondeterministic (newer) AI techniques aimed squarely at beginner AI developers. Here's a summary of what we cover:

Chapter 1, *Introduction to Game AI*

Here, we define game AI and discuss the current state of the art as well as the future of this technology.

Chapter 2, *Chasing and Evading*

We cover basic techniques for chasing and evading as well as more advanced techniques for intercepting. We also cover techniques applicable to both tile-based and continuous game environments.

Chapter 3, *Pattern Movement*

Pattern movement techniques are common to many video games and developers have been using them since the early days of gaming. You can use these techniques to preprogram certain behaviors such as the patrolling of a guard or the swooping in of a spacecraft.

Chapter 4, *Flocking*

The flocking method we examine in this chapter is an example of an A-life algorithm. In addition to creating cool-looking flocking behavior, A-life algorithms form the basis of more advanced group movement.

Chapter 5, *Potential Function Based Movement*

Potential-based movement is relatively new in game AI applications. The cool thing about this method is that it can handle chasing, evading, swarming, and collision avoidance simultaneously.

Chapter 6, *Basic Pathfinding and Waypoints*

Game developers use many techniques to find paths in and around game environments. In this chapter, we cover several of these methods, including waypoints.

Chapter 7, *A* Pathfinding*

No treatment of pathfinding is complete without addressing the workhorse algorithm of pathfinding; therefore, we devote this whole chapter to the A* algorithm.

Chapter 8, *Scripted AI and Scripting Engines*

Programmers today often write scripting engines and hand off the tools to level designers who are responsible for creating the content and defining the AI. In this chapter, we explore some of the techniques developers use to apply a scripting system in their games, and the benefits they receive.

Chapter 9, *Finite State Machines*

Finite state machines are the nuts and bolts of game AI. This chapter discusses the fundamentals of finite state machines and how to implement them.

Chapter 10, *Fuzzy Logic*

Developers use fuzzy logic in conjunction with or as a replacement for finite state machines. In this chapter, you'll learn the advantages fuzzy techniques offer over traditional logic techniques.

Chapter 11, *Rule-Based AI*

Technically, fuzzy logic and finite state machines fall under the general heading of rules-based methods. In this chapter, we cover these methods as well as other variants.

Chapter 12, *Basic Probability*

Game developers commonly use basic probability to make their games less predictable. Such cheap unpredictability enables developers to maintain substantial control over their games. Here, we cover

basic probability for this purpose as well as lay the groundwork for more advanced methods.

Chapter 13, *Decisions Under UncertaintyBayesian Techniques*

Bayesian techniques are probabilistic techniques, and in this chapter we show how you can use them for decision making and for adaptation in games.

Chapter 14, *Neural Networks*

Game developers use neural networks for learning and adaptation in gamesin fact, for anything from making decisions to predicting the behavior of players. We cover the most widely used neural network architecture, in detail.

Chapter 15, *Genetic Algorithms*

Genetic algorithms offer opportunities for evolving game AI. Although developers don't often use genetic algorithms in games, their potential for specific applications is promising, particularly if they are combined with other methods.

Appendix, *Vector Operations*

This appendix shows you how to implement a C++ class that captures all of the vector operations that you'll need when writing 2D or 3D simulations.

All the chapters in this book are fairly independent of each other. Therefore, you generally can read the chapters in any order you want, without worrying about missing material in earlier chapters. The only exception to this rule is Chapter 12, on basic probability. If you don't have a background in probability, you should read this chapter before reading Chapter 13, on Bayesian methods.

Also, we encourage you to try these algorithms for yourself in your own programs. If you're just getting started in game AI, which we assume you are if you're reading this book, you might want to begin by applying some of the techniques we present in simple arcade-style or board games. You also might consider programming a *bot* using extensible AI tools that are increasingly becoming standard for first-person shooter games. This approach will give you the opportunity to try out your AI ideas without having to program all the other non-AI aspects of your game.

Top

All Online Books

Table of Contents
View as Frames

Previous Next

Conventions Used in This Book

The following typographical conventions are used in this book:

Plain text

Indicates menu titles, menu options, menu buttons, and keyboard accelerators (such as Alt and Ctrl).

Italic

Indicates new terms, URLs, email addresses, filenames, file extensions, pathnames, directories, and Unix utilities.

`Constant width`

Indicates commands, options, switches, variables, attributes, keys, functions, types, classes, namespaces, methods, modules, properties, parameters, values, objects, events, event handlers, XML tags, HTML tags, macros, the contents of files, or the output from commands.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values.

Bold

Variables shown in bold are vectors as opposed to scalar variables, which are shown in regular print.

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

Additional Resources

Although we attempt to cover a wide range of AI techniques in this book, we realize we can't compress within these pages everything there is to know about AI in game development. Therefore, we've compiled a short list of useful AI web and print resources for you to explore should you decide to pursue game AI further.

Here are some popular web sites related to game development and AI that we find helpful:

- The Game AI Page at http://www.gameai.com
- AI Guru at http://www.aiguru.com
- Gamasutra at http://www.gamasutra.com
- GameDev.net at http://www.gamedev.net
- AI Depot at http://ai-depot.com
- Generation5 at http://www.generation5.org
- The American Association for Artificial Intelligence at http://www.aaai.org

Each web site contains information relevant to game AI as well as additional links to other sources of information on AI.

Here are several print resources that we find helpful (note that these resources include both game and nongame AI books):

- *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference* by Judea Pearl (Morgan Kaufmann Publishers, Inc.)
- *Bayesian Artificial Intelligence* by Kevin Korb and Ann Nicholson (Chapman & Hall/CRC)
- *Bayesian Inference and Decision*, Second Edition by Robert Winkler (Probabilistic Publishing)
- *AI Game Programming Wisdom* by Steve Rabin, ed. (Charles River Media)
- *AI Techniques for Game Programming* by Mat Buckland (Premier Press)
- *Practical Neural Network Recipes in C++* by Timothy Masters (Academic Press)

- *Neural Networks for Pattern Recognition* by Christopher Bishop (Oxford University Press)
- *AI Application Programming* by M. Tim Jones (Charles River Media)

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

Using Code Examples

This book is designed to help you get your job done. In general, you can use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*AI for Game Developers*, by David M. Bourg and Glenn Seemann. Copyright 2004 O'Reilly Media, Inc., 0-596-00555-5."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at permissions@oreilly.com.

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA, 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international or local)
(707) 829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/ai

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

Acknowledgments

We'd like to thank our editors, Nathan Torkington and Tatiana Diaz, for their skill, insight, and patience. We'd also like to express our appreciation to O'Reilly for giving us the opportunity to write this book. Further, special thanks go to all the production and technical staff at O'Reilly, and to the technical reviewers, chromatic, Graham Evans, and Mike Keith, for their thoughtful and expert comments and suggestions. Finally, we'd like to thank our respective wives, Helena and Michelle, for their support and encouragement and for putting up with our repeatedly saying, "I have to work on the book this weekend." We can't forget to thank our respective children, who always remind us that it's OK to play. On that note, remember that the most important goal of good AI in games is to make games more fun. Play on!

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

Chapter 1. Introduction to Game AI

In the broadest sense, most games incorporate some form of artificial intelligence (AI). For instance, developers have used AI for years to give seemingly intelligent life to countless game characters, from the ghosts in the classic arcade game *Pac Man* to the bots in the first-person shooter *Unreal*, and many others in between. The huge variety of game genres and game characters necessitates a rather broad interpretation as to what is considered game AI. Indeed, this is true of AI in more traditional scientific applications as well.

Some developers consider tasks such as pathfinding as part of game AI. Steven Woodcock reported in his "2003 Game Developer's Conference AI Roundtable Moderator's Report' that some developers even consider collision detection to be part of game AI[*]. Clearly, some wide-ranging interpretations of game AI exist.

We're going to stick with a broad interpretation of game AI, which includes everything from simple chasing and evading, to pattern movement, to neural networks and genetic algorithms. Game AI probably best fits within the scope of *weak* AI (see the sidebar "Defining AI"). However, in a sense you can think of game AI in even broader terms.

In games, we aren't always interested in giving nonplayer characters human-level intellect. Perhaps we are writing code to control nonhuman creatures such as dragons, robots, or even rodents. Further, who says we always have to make nonplayer characters smart? Making some nonplayer characters dumb adds to the variety and richness of game content. Although it is true that game AI is often called upon to solve fairly complex problems, we can employ AI in attempts to give nonplayer characters the appearance of having different personalities, or of portraying emotions or various dispositionsfor example, scared, agitated, and so on.

Defining AI

The question "what is artificial intelligence?" is not easy to answer. If you look up *artificial intelligence* in a dictionary, you'll probably find a definition that reads something like this: "The ability of a computer or other machine to perform those activities that are normally thought to require intelligence." This definition comes from *The American Heritage Dictionary of the English Language, Fourth Edition* (Houghton Mifflin Company). Still other sources define artificial intelligence as the process or science of creating intelligent machines.

From another perspective it's appropriate to think of AI as the intelligent behavior exhibited by the machine that has been created, or perhaps the artificial brains behind that intelligent behavior. But even this interpretation is not complete. To some folks, the study of AI is not necessarily for the purpose of creating intelligent machines, but for the purpose of gaining better insight into the nature of human intelligence. Still others study AI methods to create machines that exhibit some limited form of intelligence.

This begs the question: "what is intelligence?" To some, the litmus test for AI is how close it is to human intelligence. Others argue that additional requirements must be met for a machine to be considered intelligent. Some people say intelligence requires a conscience and that emotions are integrally tied to intelligence, while others say the ability to solve a problem requiring intelligence if it were to be solved by a human is not enough; AI must also learn and adapt to be considered intelligent.

AI that satisfies all these requirements is considered *strong AI*. Unlike strong AI, *weak AI* involves a broader range of purposes and technologies to give machines specialized intelligent qualities. Game AI falls into the category of weak AI.

The bottom line is that the definition of game AI is rather broad and flexible. Anything that gives the illusion of intelligence to an appropriate level, thus making the game more immersive, challenging, and, most importantly, fun, can be considered game AI. Just like the use of real physics in games, good AI adds to the immersiveness of the game, drawing players in and suspending their reality for a time.

[*]

[*] Steven Woodcock maintains an excellent Web site devoted to game AI at http://www.gameai.com.

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

1.1 Deterministic Versus Nondeterministic AI

Game AI techniques generally come in two flavors: *deterministic* and *nondeterministic.*

Deterministic

> Deterministic behavior or performance is specified and predictable. There's no uncertainty. An example of deterministic behavior is a simple chasing algorithm. You can explicitly code a nonplayer character to move toward some target point by advancing along the x and y coordinate axes until the character's x and y coordinates coincide with the target location.

Nondeterministic

> Nondeterministic behavior is the opposite of deterministic behavior. Behavior has a degree of uncertainty and is somewhat unpredictable (the degree of uncertainty depends on the AI method employed and how well that method is understood). An example of nondeterministic behavior is a nonplayer character learning to adapt to the fighting tactics of a player. Such learning could use a neural network, a Bayesian technique, or a genetic algorithm.

Deterministic AI techniques are the bread and butter of game AI. These techniques are predictable, fast, and easy to implement, understand, test, and debug. Although they have a lot going for them, deterministic methods place the burden of anticipating all scenarios and coding all behavior explicitly on the developers' shoulders. Further, deterministic methods do not facilitate learning or evolving. And after a little gameplay, deterministic behaviors tend to become predictable. This limits a game's play-life, so to speak.

Nondeterministic methods facilitate learning and unpredictable gameplay. Further, developers don't have to explicitly code all behaviors in anticipation of all possible scenarios. Nondeterministic methods also can learn and extrapolate on their own, and they can promote so-called *emergent* behavior, or behavior that emerges without explicit instructions. The flocking and neural network algorithms we'll consider in this book are good

examples of emergent behavior.

Developers traditionally have been a bit wary of AI that is nondeterministic, although this is changing. Unpredictability is difficult to test and debughow can you test all possible variations of player action to make sure the game doesn't do something silly in some cases? Game developers face an ever-shortening development cycle that makes developing and testing new technology to production-ready standards extremely difficult. Such short development periods make it difficult for developers to understand cutting-edge AI technologies fully and to see their implications in a mass-market commercial game.

At least until recently, another factor that has limited game AI development is the fact that developers have been focusing most of their attention on graphics quality. As it turns out, such focus on developing better and faster graphics techniques, including hardware acceleration, might now afford more resources to be allocated toward developing better, more sophisticated AI. This fact, along with the pressure to produce the next hit game, is encouraging game developers to more thoroughly explore nondeterministic techniques. We'll come back to this point a little later.

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

1.2 Established Game AI

Perhaps the most widely used AI technique in games is *cheating*. For example, in a war simulation game the computer team can have access to all information on its human opponentslocation of their base; the types, number, and location of units, etc.without having to send out scouts to gather such intelligence the way a human player must. Cheating in this manner is common and helps give the computer an edge against intelligent human players. However, cheating can be bad. If it is obvious to the player that the computer is cheating, the player likely will assume his efforts are futile and lose interest in the game. Also, unbalanced cheating can give computer opponents too much power, making it impossible for the player to beat the computer. Here again, the player is likely to lose interest if he sees his efforts are futile. Cheating must be balanced to create just enough of a challenge for the player to keep the game interesting and fun.

Of course, cheating isn't the only well-established AI technique. *Finite state machines* are a ubiquitous game AI technique. We cover them in detail in Chapter 9, but basically the idea is to enumerate a bunch of actions or states for computer-controlled characters and execute them or transition between them using if-then conditionals that check various conditions and criteria.

Developers commonly use *fuzzy logic* in fuzzy state machines to make the resulting actions somewhat less predictable and to reduce the burden of having to enumerate huge numbers of if-then rules. Rather than have a rule that states *if distance = 10 and health = 100 then attack*, as you might in a finite state machine, fuzzy logic enables you to craft rules using less precise conditions, such as *if close and healthy then attack aggressively*. We cover fuzzy logic in Chapter 10.

Effective and efficient pathfinding is a fundamental task that nonplayer characters must accomplish in all sorts of games. Nonplayer character units in a war simulation must be able to navigate over terrain and avoid barriers to reach the enemy. Creatures in a first-person shooter must be able to navigate through dungeons or buildings to reach or escape from the player. The scenarios are endless, and it's no wonder that AI developers give pathfinding tremendous attention. We cover general pathfinding techniques in Chapter 6 and the venerable A* algorithm in Chapter 7.

These are only a few of the established game AI techniques; others include scripting, rules-based systems, and some artificial life (A-life) techniques, to name a few. A-life techniques are common in robotic applications, and developers have adapted and used them with great success in video games. Basically, an A-life system is a synthetic system that exhibits natural behaviors. These behaviors are emergent and develop as a result of the combined effect of lower-level algorithms. We'll see examples of A-life as well as other techniques throughout this book.

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

1.3 The Future of Game AI

The next big thing in game AI is learning. Rather than have all nonplayer character behavior be predestined by the time a game ships, the game should evolve, learn, and adapt the more it's played. This results in a game that grows with the player and is harder for the player to predict, thus extending the play-life of the game. It is precisely this unpredictable nature of learning and evolving games that has traditionally made AI developers approach learning techniques with a healthy dose of trepidation.

The techniques for learning and reacting to character behavior fall under the nondeterministic AI we talked about earlier, and its difficulties apply here too. Specifically, such nondeterministic, learning AI techniques take longer to develop and test. Further, it's more difficult to really understand what the AI is doing, which makes debugging more difficult. These factors have proven to be serious barriers for widespread use of learning AI techniques. All this is changing, though.

Several mainstream games, such as *Creatures*, *Black & White*, *Battlecruiser 3000AD*, *Dirt Track Racing*, *Fields of Battle*, and *Heavy Gear*, used nondeterministic AI methods. Their success sparked a renewed interest in learning AI methods such as decision trees, neural networks, genetic algorithms, and probabilistic methods.

These successful games use nondeterministic methods in conjunction with more traditional deterministic methods, and use them only where they are needed and only for problems for which they are best suited. A neural network is not a magic pill that will solve all AI problems in a game; however, you can use it with impressive results for very specific AI tasks within a hybrid AI system. This is the approach we advocate for using these nondeterministic methods. In this way, you can at least isolate the parts of your AI that are unpredictable and more difficult to develop, test, and debug, while ideally keeping the majority of your AI system in traditional form.

Throughout this book we cover both traditional game AI techniques as well as relatively new, up-and-coming AI techniques. We want to arm you with a thorough understanding of what has worked and continues to work for game AI. We also want you to learn several promising new techniques to give you a head start toward the

future of game AI.

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

Chapter 2. Chasing and Evading

In this chapter we focus on the ubiquitous problem of chasing and evading. Whether you're developing a spaceship shooter, a strategy simulation, or a role-playing game, chances are you will be faced with trying to make your game's nonplayer characters either chase down or run from your player character. In an action or arcade game the situation might involve having enemy spaceships track and engage the player's ship. In an adventure role-playing game it might involve having a troll or some other lovely creature chase down your player's character. In first-person shooters and flight simulations you might have to make guided missiles track and strike the player or his aircraft. In any case, you need some logic that enables nonplayer character predators to chase, and their prey to run.

The chasing/evading problem consists of two parts. The first part involves the decision to initiate a chase or to evade. The second part involves effecting the chase or evasionthat is, getting your predator to the prey, or having the prey get as far from the predator as possible without getting caught. In a sense, one could argue that the chasing/evading problem contains a third element: obstacle avoidance. Having to avoid obstacles while chasing or evading definitely complicates matters, making the algorithms more difficult to program. Although we don't cover obstacle avoidance in this chapter, we will come back to it in Chapters 5 and 6. In this chapter we focus on the second part of the problem: effecting the chase or evasion. We'll discuss the first part of the problemdecision makingin later chapters, when we explore such topics as state machines and neural networks, among others.

The simplest, easiest-to-program, and most common method you can use to make a predator chase its prey involves updating the predator's coordinates through each game loop such that the difference between the predator's coordinates and the prey's coordinates gets increasingly small. This algorithm pays no attention to the predator and prey's respective headings (the direction in which they're traveling) or their speeds. Although this method is relentlessly effective in that the predator constantly moves toward its prey unless it's impeded by an obstacle, it does have its limitations, as we'll discuss shortly.

In addition to this very basic method, other methods are available to you that might better serve your needs,

depending on your game's requirements. For example, in games that incorporate real-time physics engines you can employ methods that consider the positions and velocities of both the predator and its prey so that the predator can try to intercept its prey instead of relentlessly chasing it. In this case the relative position and velocity information can be used as input to an algorithm that will determine appropriate force actuationsteering forces, for exampleto guide the predator to the target. Yet another method involves using potential functions to influence the behavior of the predator in a manner that makes it chase its prey, or more specifically, makes the prey attract the predator. Similarly, you can use such potential functions to cause the prey to run from or repel a predator. We cover potential functions in Chapter 5.

In this chapter we explore several chase and evade methods, starting with the most basic method. We also give you example code that implements these methods in the context of tile-based and continuous-movement environments.

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

2.1 Basic Chasing and Evading

As we said earlier, the simplest chase algorithm involves correcting the predator's coordinates based on the prey's coordinates so as to reduce the distance between their positions. This is a very common method for implementing basic chasing and evading. (In this method, evading is virtually the opposite of chasing, whereby instead of trying to decrease the distance between the predator and prey coordinates, you try to increase it.) In code, the method looks something like that shown in Example 2-1.

Example 2-1. Basic chase algorithm

```
if (predatorX > preyX)
     predatorX--;
else if (predatorX < preyX)
     predatorX++;
if (predatorY > preyY)
     predatorY--;
else if (predatorY < preyY)
     predatorY++;
```

In this example, the prey is located at coordinates *preyX* and *preyY*, while the predator is located at coordinates *predatorX* and *predatorY*. During each cycle through the game loop the predator's coordinates are checked against the prey's. If the predator's x-coordinate is greater than the prey's x-coordinate, the predator's x-coordinate is decremented, moving it closer to the prey's x-position. Conversely, if the predator's x-coordinate is less than the prey's, the predator's x-coordinate is incremented. Similar logic applies to the predator's y-coordinate based on the prey's y-coordinate. The end result is that the predator will move closer and closer to the prey each cycle through the game loop.

Using this same methodology, we can implement evading by simply reversing the logic, as we illustrate in

Example 2-2.

Example 2-2. Basic evade algorithm

```
if (preyX > predatorX)
     preyX++;
else if (preyX < predatorX)
     preyX--?>;
if (preyY > predatorY)
     preyY++;
else if (preyY < predatorY)
     preyY--;
```

In *tile-based games* the game domain is divided into discrete tilessquares, hexagons, etc.and the player's position is fixed to a discrete tile. Movement goes tile by tile, and the number of directions in which the player can make headway is limited. In a *continuous environment*, position is represented by floating-point coordinates, which can represent any location in the game domain. The player also is free to head in any direction.

You can apply the approach illustrated in these two examples whether your game incorporates tile-based or continuous movement. In tile-based games, the *x*s and *y*s can represent columns and rows in a grid that encompasses the game domain. In this case, the *x*s and *y*s would be integers. In a continuous environment, the *x*s and *y*sand *z*s if yours is a 3D gamewould be real numbers representing the coordinates in a Cartesian coordinate system encompassing the game domain.

There's no doubt that although it's simple, this method works. The predator will chase his prey with unrelenting determination. The sample program *AIDemo2-1*, available for download from this book's web site (http://www.oreilly.com/BOOK"), implements the basic chase algorithm in a tile-based environment. The relevant code is shown in Example 2-3.

Example 2-3. Basic tile-based chase example

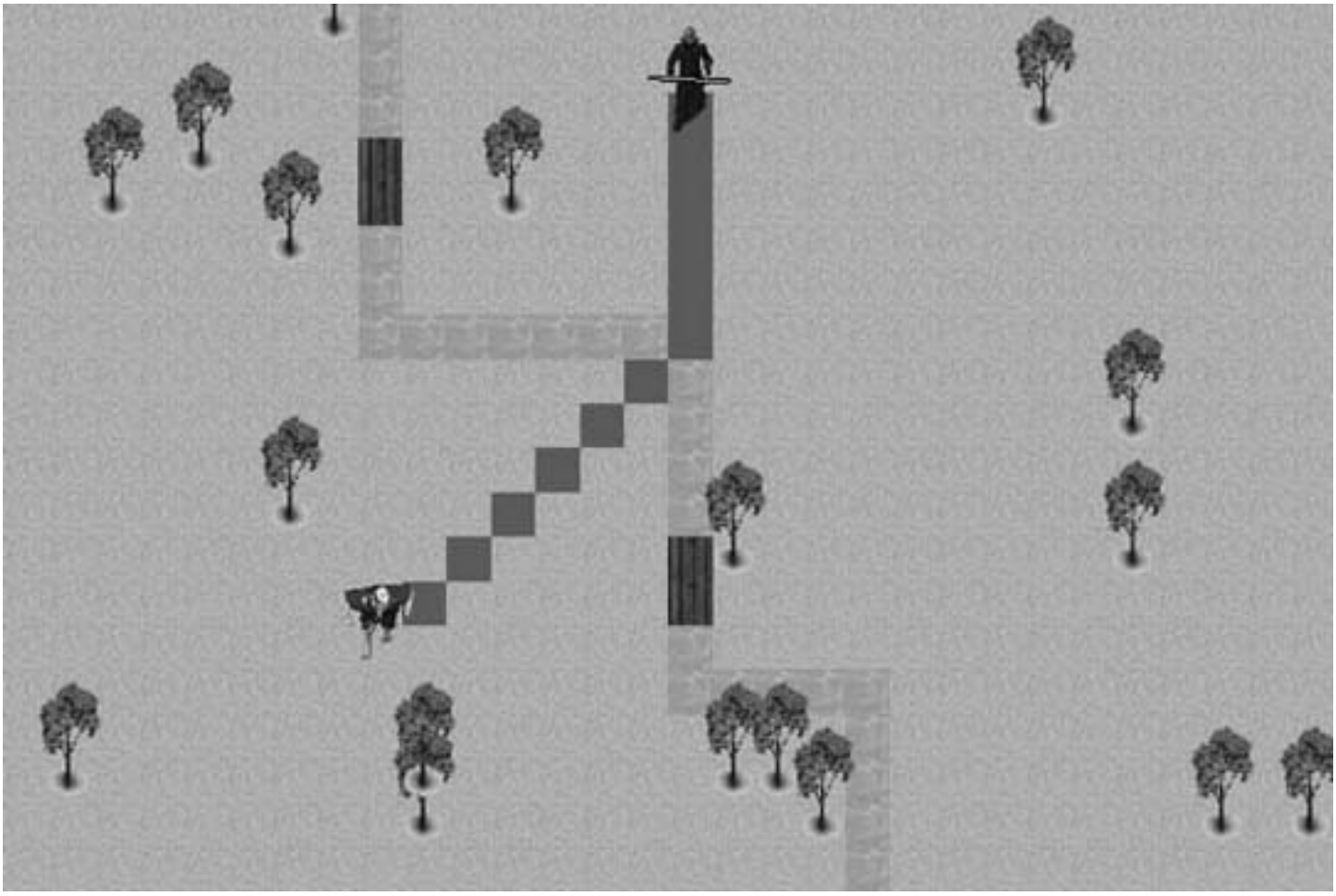
```
if (predatorCol > preyCol)
     predatorCol--;
else if (predatorCol < preyCol)
     predatorCol++;
if (predatorRow> preyRow)
     predatorRow--;
else if (predatorRow<preyRow)
```

```
    predatorRow++;
```

Notice the similarities in Examples 2-3 and 2-1. The only difference is that in Example 2-3 rows and columns are used instead of floating-point *x*s and *y*s.

The trouble with this basic method is that often the chasing or evading seems almost too mechanical. Figure 2-1 illustrates the path the troll in the sample program takes as he pursues the player.

Figure 2-1. Basic tile-based chase



As you can see, the troll first moves diagonally toward the player until one of the coordinates, the horizontal in this case, equals that of the player's.[*] Then the troll advances toward the player straight along the other coordinate axis, the vertical in this case. Clearly this does not look very natural. A better approach is to have the troll move directly toward the player in a straight line. You can implement such an algorithm without too much difficulty, as we discuss in the next section.

[*]

[*] In square tile-based games, characters appear to move faster when moving along a diagonal path. This is because the length of the diagonal of a square is SQRT(2) times longer than its sides. Thus, for every diagonal step, the character appears to move SQRT(2) times faster than when it moves horizontally or vertically.

Previous Next
Top

All Online Books

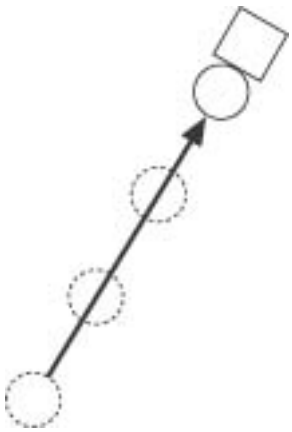
Table of Contents
View as Frames

Previous Next

2.2 Line-of-Sight Chasing

In this section we explain a few chasing and evading algorithms that use a line-of-sight approach. The gist of the line-of-sight approach is to have the predator take a straight-line path toward the prey; the predator always moves directly toward the prey's current position. If the prey is standing still, the predator will take a straight line path. However, if the prey is moving, the path will not necessarily be a straight line. The predator still will attempt to move directly toward the current position of the prey, but by the time he catches up with the moving prey, the path he would have taken might be curved, as illustrated in Figure 2-2.

Figure 2-2. Line-of-sight chasing



In Figure 2-2, the circles represent the predator and the diamonds represent the prey. The dashed lines and shapes indicate starting and intermediate positions. In the scenario on the left, the prey is sitting still; thus the predator makes a straight-line dash toward the prey. In the scenario on the right, the prey is moving along some arbitrary path over time. At each time step, or cycle through the game loop, the predator moves toward the current position of the prey. As the prey moves, the predator traces out a curved path from its starting point.

The results illustrated here look more natural than those resulting from the basic-chase algorithm. Over the remainder of this section, we'll show you two algorithms that implement line-of-sight chasing. One algorithm is

specifically for tiled environments, while the other applies to continuous environments.

Previous Next
Top

All Online Books

Table of Contents
View as Frames

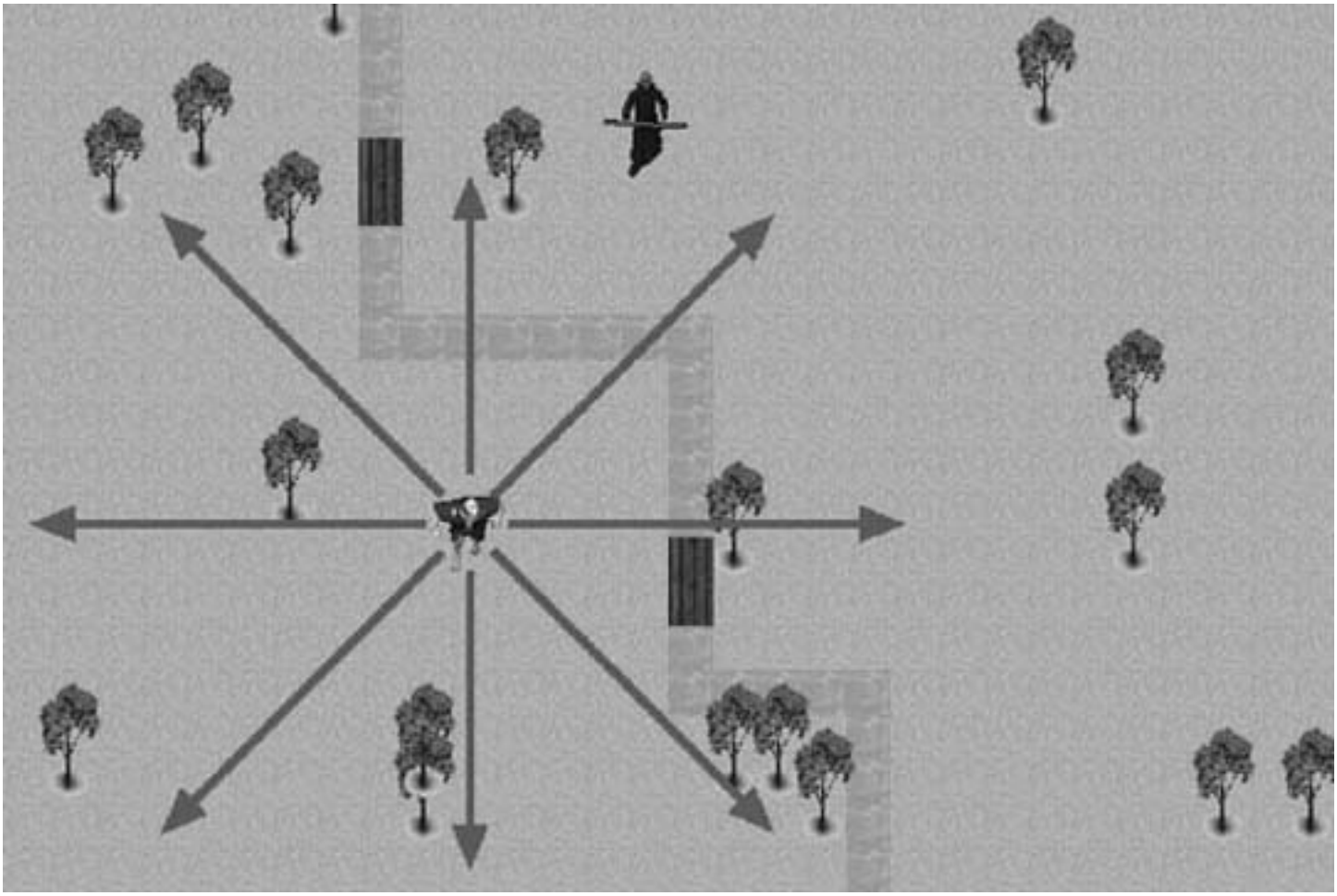
Previous Next

2.3 Line-of-Sight Chasing in Tiled Environments

As we stated earlier, the environment in a tile-based game is divided into discrete tiles. This places certain limitations on movement that don't necessarily apply in a continuous environment. In a continuous environment, positions usually are represented using floating-point variables. Those positions are then mapped to the nearest screen pixel. When changing positions in a continuous environment, you don't always have to limit movement to adjacent screen pixels. Screen pixels typically are small enough so that a small number of them can be skipped between each screen redraw without sacrificing motion fluidity.

In tile-based games, however, changing positions is more restrictive. By its very nature, tile-based movement can appear jaggy because each tile is not mapped to a screen pixel. To minimize the jaggy and sometimes jumpy appearance in tile-based games, it's important to move only to adjacent tiles when changing positions. For games that use square tiles, such as the example game, this offers only eight possible directions of movement. This limitation leads to an interesting problem when a predator, such as the troll in the example, is chasing its target. The troll is limited to only eight possible directions, but mathematically speaking, none of those directions can accurately represent the true direction of the target. This dilemma is illustrated in Figure 2-3.

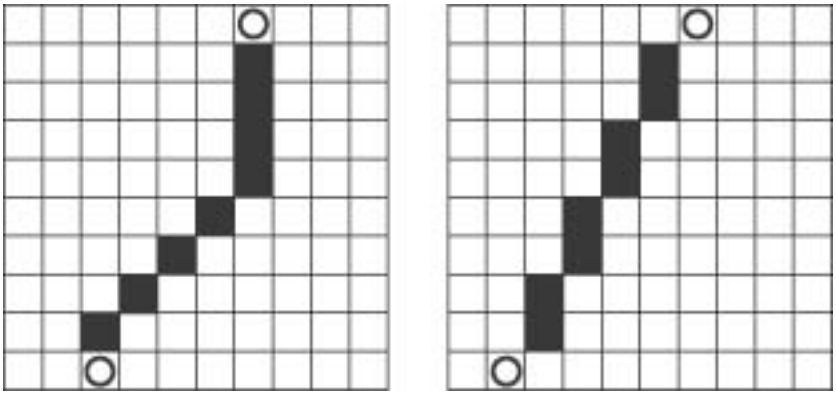
Figure 2-3. Tile-based eight-way movement



As you can see in Figure 2-3, none of the eight possible directions leads directly to the target. What we need is a way to determine which of the eight adjacent tiles to move to so that the troll appears to be moving toward the player in a straight line.

As we showed you earlier, you can use the simple chasing algorithm to make the troll relentlessly chase the player. It will even calculate the shortest possible path to the player. So, what's the disadvantage? One concerns aesthetics. When viewed in a tile-based environment, the simple chase method doesn't always appear to produce a visually straight line. Figure 2-4 illustrates this point.

Figure 2-4. Simple chase versus line-of-sight chase



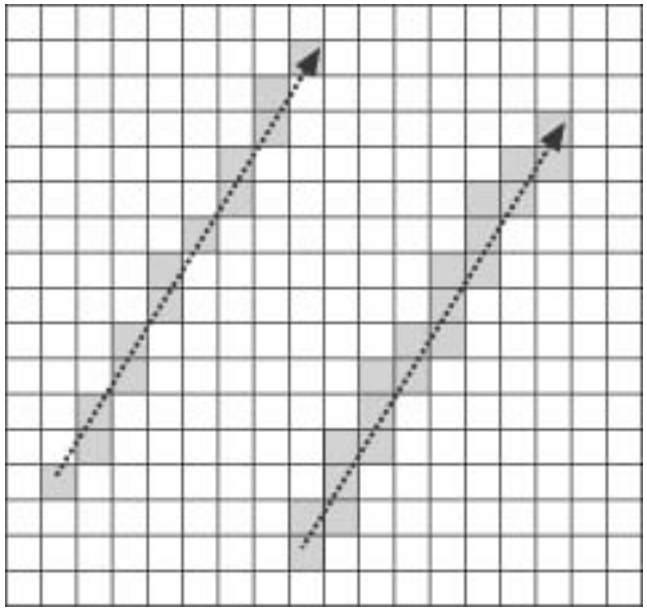
Another reason to avoid the simple chase method is that it can have undesirable side effects when a group of predators, such as a pack of angry trolls, are converging on the player. Using the simple method, they would all walk diagonally to the nearest axis of their target and then walk along that axis to the target. This could lead to them walking single file to launch their attack. A more sophisticated approach is to have them walk directly toward the target from different directions.

It's interesting to note that both paths shown in Figure 2-4 are the same distance. The line-of-sight method, however, appears more natural and direct, which in turn makes the troll seem more intelligent. So, the objective for the line-of-sight approach is to calculate a path so that the troll appears to be walking in a straight line toward the player.

The approach we'll take to solve this problem involves using a standard line algorithm that is typically used to draw lines in a pixel environment. We're essentially going to treat the tile-based environment as though each tile was in fact a giant screen pixel. However, instead of coloring the pixels to draw a line on the screen, the line algorithm is going to tell us which tiles the troll should follow so that it will walk in a straight line to its target.

Although you can calculate the points of a line in several ways, in this example we're going to use Bresenham's line algorithm. Bresenham's algorithm is one of the more efficient methods for drawing a line in a pixel-based environment, but that's not the only reason it's useful for pathfinding calculations. Bresenham's algorithm also is attractive because unlike some other line-drawing algorithms, it will never draw two adjacent pixels along a line's shortest axis. For our pathfinding needs, this means the troll will walk along the shortest possible path between the starting and ending points. Figure 2-5 shows how Bresenham's algorithm, on the left, might compare to other line algorithms that can sometimes draw multiple pixels along the shortest axis. If an algorithm that generated a line such as the one shown on the right is used, the troll would take unnecessary steps. It still would still reach its target, but not in the shortest and most efficient way.

Figure 2-5. Bresenham versus alternate line algorithm



As Figure 2-5 shows, a standard algorithm such as the one shown on the right would mark every tile for pathfinding that mathematically intersected the line between the starting and ending points. This is not desirable for a pathfinding application because it won't generate the shortest possible path. In this case, Bresenham's algorithm produces a much more desirable result.

The Bresenham algorithm used to calculate the direction of the troll's movement takes the starting point, which is the row and column of the troll's position, and the ending point, which is the row and column of the player's position, and calculates a series of steps the troll will have to take so that it will walk in a straight line to the player. Keep in mind that this function needs to be called each time the troll's target, in this case the player, changes position. Once the target moves, the precalculated path becomes obsolete, and therefore it becomes necessary to calculate it again. Examples 2-4 through 2-7 show how you can use the Bresenham algorithm to build a path to the troll's target.

Example 2-4. BuildPathToTarget function

```
void ai_Entity::BuildPathToTarget (void)
{
     int nextCol=col;
     int nextRow=row;
     int deltaRow=endRow-row;
     int deltaCol=endCol-col;
     int stepCol, stepRow;
     int currentStep, fraction;
```

As Example 2-4 shows, this function uses values stored in the *ai_Entity* class to establish the starting and ending

points for the path. The values in *col* and *row* are the starting points of the path. In the case of the sample program, *col* and *row* contain the current position of the troll. The values in *endRow* and *endCol* contain the position of the troll's desired location, which in this case is the player's position.

Example 2-5. Path initialization

```
for (currentStep=0;currentStep<kMaxPathLength; currentStep++)
{
    pathRow[currentStep]=-1;
        pathCol[currentStep]=-1;
}
currentStep=0;
pathRowTarget=endRow;
pathColTarget=endCol;
```

In Example 2-5 you can see the row and column path arrays being initialized. This function is called each time the player's position changes, so it's necessary to clear the old path before the new one is calculated.

Upon this function's exit, the two arrays, *pathRow* and *pathCol*, will contain the row and column positions along each point in the troll's path to its target. Updating the troll's position then becomes a simple matter of traversing these arrays and assigning their values to the troll's row and column variables each time the troll is ready to take another step.

Had this been an actual line-drawing function, the points stored in the path arrays would be the coordinates of the pixels that make up the line.

The code in Example 2-6 determines the direction of the path by using the previously calculated *deltaRow* and *deltaCol* values.

Example 2-6. Path direction calculation

```
    if (deltaRow <  0) stepRow=-1;  else stepRow=1;
    if (deltaCol <  0) stepCol=-1;  else stepCol=1;
    deltaRow=abs(deltaRow*2);
    deltaCol=abs(deltaCol*2);
    pathRow[currentStep]=nextRow;
    pathCol[currentStep]=nextCol;
    currentStep++;
```

It also sets the first values in the path arrays, which in this case is the row and column position of the troll.

Example 2-7 shows the meat of the Bresenham algorithm.

Example 2-7. Bresenham algorithm

```
if (deltaCol >deltaRow)
      {
       fraction = deltaRow *2-deltaCol;
       while (nextCol != endCol)
        {
         if (fraction >=0)
          {
           nextRow =nextRow +stepRow;
           fraction =fraction -deltaCol;
          }
         nextCol=nextCol+stepCol;
         fraction=fraction +deltaRow;
         pathRow[currentStep]=nextRow;
         pathCol[currentStep]=nextCol;
         currentStep++;
        }
      }
     else
      {
       fraction =deltaCol *2-deltaRow;
       while (nextRow !=endRow)
        {
         if (fraction >=0)
          {
           nextCol=nextCol+stepCol;
           fraction=fraction -deltaRow;
          }
         nextRow =nextRow +stepRow;
         fraction=fraction +deltaCol;
         pathRow[currentStep]=nextRow;
         pathCol[currentStep]=nextCol;
         currentStep++;
        }
```

```
    }
  }
```

The initial *if* conditional uses the values in *deltaCol* and *deltaRow* to determine which axis is the longest. The first block of code after the *if* statement will be executed if the column axis is the longest. The *else* part will be executed if the row axis is the longest. The algorithm will then traverse the longest axis, calculating each point of the line along the way. Figure 2-6 shows an example of the path the troll would follow using the Bresenham line-of-sight algorithm. In this case, the row axis is the longest, so the *else* part of the main *if* conditional would be executed.

Figure 2-6. Bresenham tile-based chase

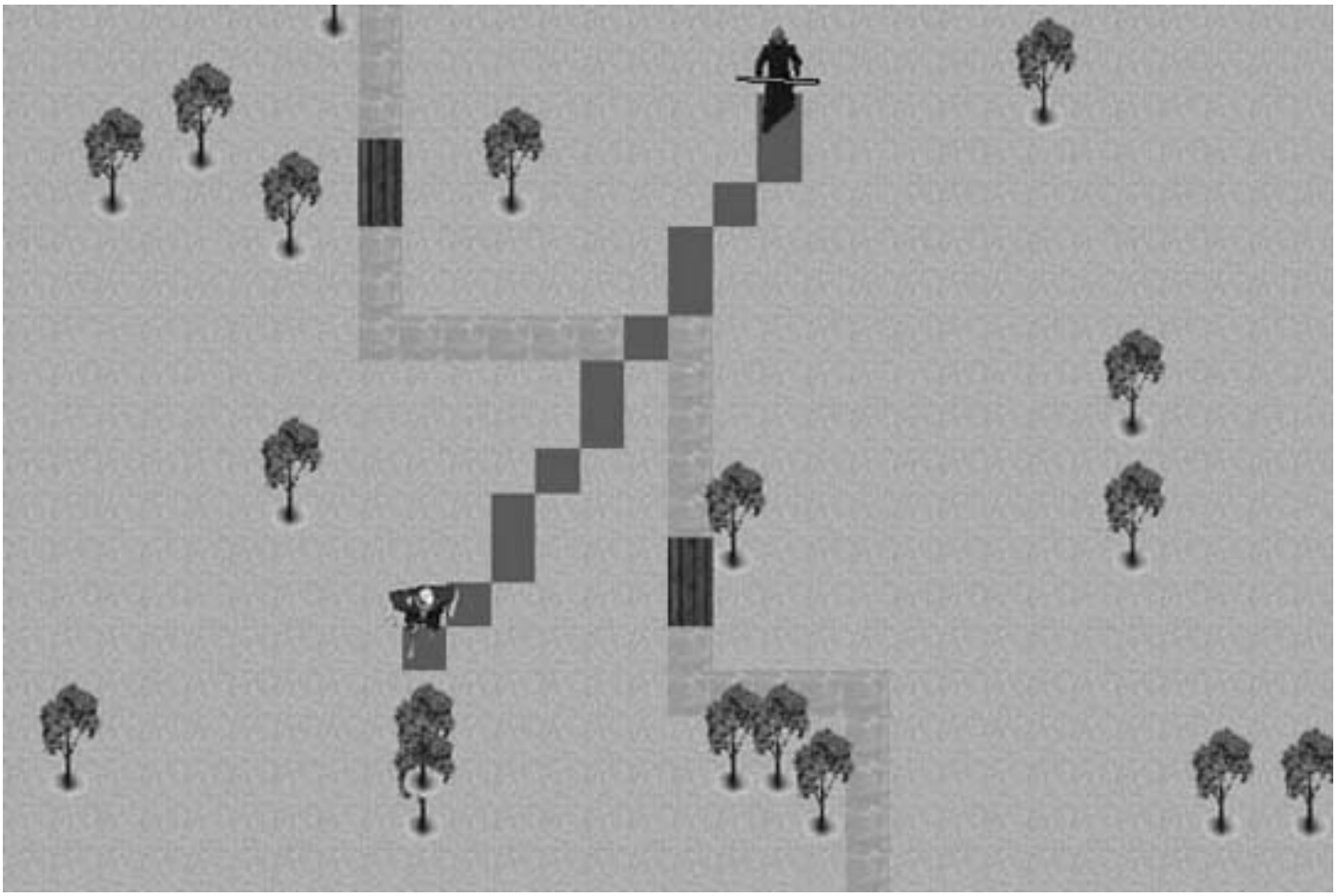


Figure 2-6 shows the troll's path, but of course this function doesn't actually draw the path. Instead of drawing the line points, this function stores each row and column coordinate in the *pathRow* and *pathCol* arrays. These stored values are then used by an outside function to guide the troll along a path that leads to the player.

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

2.4 Line-of-Sight Chasing in Continuous Environments

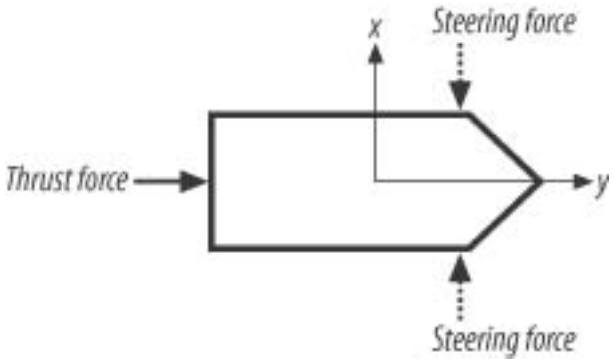
The Bresenham algorithm is an effective method for tiled environments. In this section we discuss a line-of-sight chase algorithm in the context of continuous environments. Specifically, we will show you how to implement a simple chase algorithm that you can use for games that incorporate physics engines where the game entitiesairplanes, spaceships, hovercraft, etc.are driven by applied forces and torques.

The example we'll discuss in this section uses a simple two-dimensional, rigid-body physics engine to calculate the motion of the predator and prey vehicles. You can download the complete source code for this sample program, *AIDemo2-2*, from this book's web site. We'll cover as much rigid-body physics as necessary to get through the example, but we won't go into great detail. For thorough coverage of this subject, refer to *Physics for Game Developers* (O'Reilly).

Here's the scenario. The player controls his vehicle by applying thrust for forward motion and steering forces for turning. The computer-controlled vehicle uses the same mechanics as the player's vehicle does, but the computer controls the thrust and steering forces for its vehicle. We want the computer to chase the player wherever he moves. The player will be the prey while the computer will be the predator. We're assuming that the only information the predator has about the prey is the prey's current position. Knowing this, along with the predator's current position, enables us to construct a line of sight from the predator to the prey. We will use this line of sight to decide how to steer the predator toward the prey. (In the next section we'll show you another method that assumes knowledge of the player's position and velocity expressed as a vector.)

Before getting to the chase algorithm, we want to explain how the predator and prey vehicles will move. The vehicles are identical in that both are pushed about by some applied thrust force and both can turn via activation of steering forces. To turn right, a steering force is applied that will push the nose of the vehicle to the right. Likewise, to turn left, a steering force is applied that will push the nose of the vehicle to the left. For the purposes of this example, we assume that the steering forces are bow thrustersfor example, they could be little jets located on the front end of the vehicle that push the front end to either side. These forces are illustrated in Figure 2-7.

Figure 2-7. Vehicle forces



The line-of-sight algorithm will control activation of the steering forces for the predator so as to keep it heading toward the prey at all times. The maximum speed and turning rate of each vehicle are limited due to development of linear and angular drag forces (forces that oppose motion) that are calculated and applied to each vehicle. Also, the vehicles can't always turn on a dime. Their turning radius is a function of their linear speedthe higher their linear speed, the larger the turning radius. This makes the paths taken by each vehicle look smoother and more natural. To turn on a dime, they have to be traveling fairly slowly.

Example 2-8 shows the function that controls the steering for the predator. It gets called every time step through the simulationthat is, every cycle through the physics engine loop. What's happening here is that the predator constantly calculates the prey's location relative to itself and then adjusts its steering to keep itself pointed directly toward the prey.

Example 2-8. Line-of-sight chase function

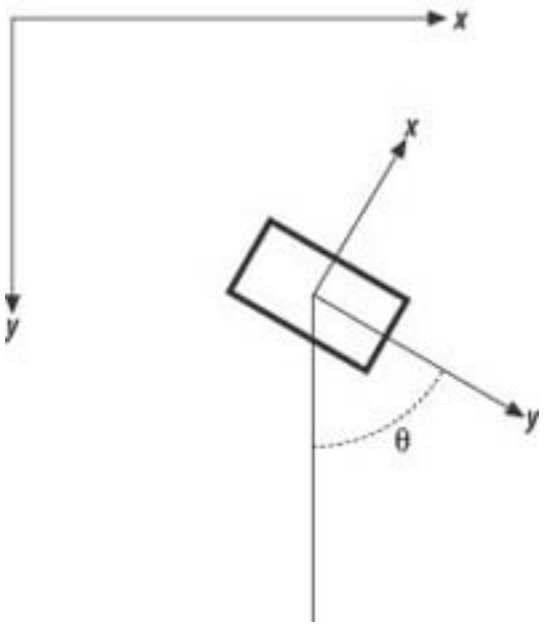
```
void    DoLineOfSightChase (void)
{
     Vector   u, v;
     bool     left = false;
     bool     right = false;
     u = VRotate2D(-Predator.fOrientation,
                       (Prey.vPosition - Predator.vPosition));
     u.Normalize();
     if (u.x <  -_TOL)
          left = true;
     else if (u.x > _TOL)
          right = true;
     Predator.SetThrusters(left, right);
}
```

As you can see, the algorithm in Example 2-8 is fairly simple. Upon entering the function, four local variables are defined. *u* and *v* are *Vector* types, where the *Vector* class is a custom class (see Appendix A) that handles all the basic vector math such as vector addition, subtraction, dot products, and cross products, among other operations. The two remaining local variables are a couple of boolean variables, *left* and *right*. These are flags that indicate which steering force to turn on; both are set to *false* initially.

The next line of code after the local variable definitions calculates the line of sight from the predator to the prey. Actually, this line does more than calculate the line of site. It also calculates the relative position vector between the predator and prey in global, earth-fixed coordinates via the code *(Prey.vPosition - Predator.vPosition)*, and then it passes the resulting vector to the function *VRotate2D* to convert it to the predator's local, body-fixed coordinates. *VRotate2D* performs a standard coordinate-system transform given the body-fixed coordinate system's orientation with respect to the earth-fixed system (see the sidebar "Global & Local Coordinate Systems"). The result is stored in *u*, and then *u* is normalizedthat is, it is converted to a vector of unit length.

Global & Local Coordinate Systems

A global (earth-fixed) coordinate system is fixed and does not move, whereas a local (body-fixed) coordinate system is locked onto objects that move around within the global, fixed coordinate system. A local coordinate system also rotates with the object to which it is attached.



You can use the following equations to convert coordinates expressed in terms of global coordinates to an object's local coordinate system given the object's orientation relative to the global coordinate system:

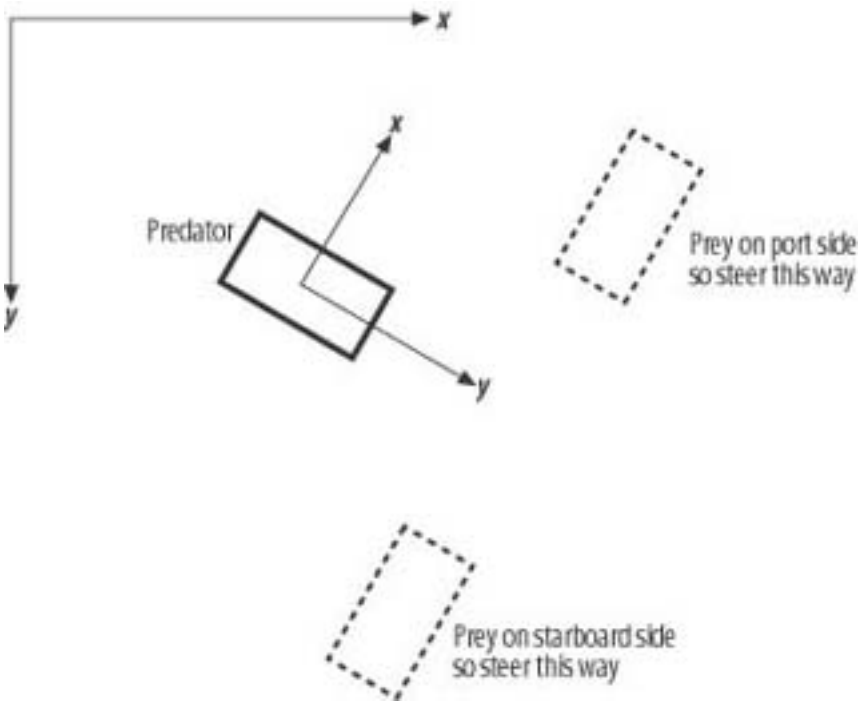
$$x = X\cos\theta + Y\sin\theta$$
$$y = -X\sin\theta + Y\cos\theta$$

Here, (x, y) are the local, body-fixed coordinates of the global point (X, Y).

What we have now is a unit vector, *u*, pointing from the predator directly toward the prey. With this vector the next few lines of code determine whether the prey is to the port side, the starboard side, or directly in front of the predator, and steering adjustments are made accordingly. The local y-axis fixed to the predator points in the positive direction from the back to the front of the vehiclethat is, the vehicle always heads along the positive local y-axis.

So, if the x-position of the prey, in terms of the predator's local coordinate system, is negative, the prey is somewhere to the starboard side of the predator and the port steering force should be activated to correct the predator's heading so that it again points directly toward the prey. Similarly, if the prey's x-coordinate is positive, it is somewhere on the port side of the predator and the starboard steering force should be activated to correct the predator's heading. Figure 2-8 illustrates this test to determine which bow thruster to activate. If the prey's x-coordinate is zero, no steering action should be taken.

Figure 2-8. Steering force test



The last line of code calls the *SetThrusters* member function of the *rigid body* class for the predator to apply the steering force for the current iteration through the simulation loop. In this example we assume a constant steering force, which can be tuned as desired.

The results of this algorithm are illustrated in Figure 2-9.

Figure 2-9. Line-of-sight chase in continuous environment

Figure 2-9 shows the paths taken by both the predator and the prey. At the start of the simulation, the predator was located in the lower left corner of the window while the prey was located in the lower right. Over time, the prey traveled in a straight line toward the upper left of the window. The predator's path curved as it continuously adjusted its heading to keep pointing toward the moving prey.

Just like the basic algorithm we discussed earlier in this chapter, this line-of-sight algorithm is relentless. The predator will always head directly toward the prey and most likely end up right behind it, unless it is moving so fast that it overshoots the prey, in which case it will loop around and head toward the prey again. You can prevent overshooting the prey by implementing some sort of speed control logic to allow the predator to slow down as it gets closer to the prey. You can do this by simply calculating the distance between the two vehicles, and if that distance is less than some predefined distance, reduce the forward thrust on the predator. You can calculate the distance between the two vehicles by taking the magnitude of the difference between their position vectors.

If you want the computer-controlled vehicle to evade the player rather than chase him, all you have to do is reverse the greater-than and less-than signs in Example 2-8.

All Online Books

Table of Contents
View as Frames

Previous Next

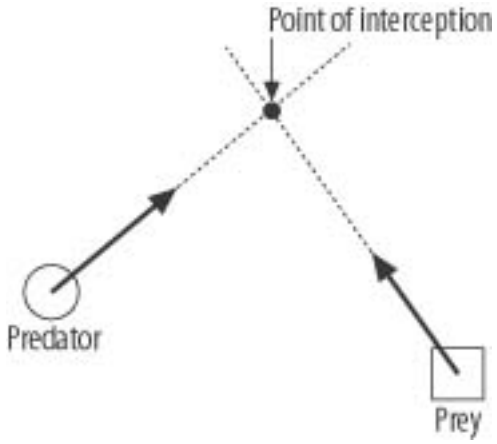
2.5 Intercepting

The line-of-sight chase algorithm we discussed in the previous section in the context of continuous movement is effective in that the predator always will head directly toward the prey. The drawback to this algorithm is that heading directly toward the prey is not always the shortest path in terms of range to target or, perhaps, time to target. Further, the line-of-sight algorithm usually ends up with the predator following directly behind the prey unless the predator is faster, in which case it will overshoot the prey. A more desirable solution in many casesfor example, a missile being shot at an aircraftis to have the predator intercept the prey at some point along the prey's trajectory. This allows the predator to take a path that is potentially shorter in terms of range or time. Further, such an algorithm could potentially allow slower predators to intercept faster prey.

To explain how the intercept algorithm works, we'll use as a basis the physics-based game scenario we described earlier. In fact, all that's required to transform the basic- chase algorithm into an intercept algorithm is the addition of a few lines of code within the chase function. Before getting to the code, though, we want to explain how the intercept algorithm works in principle. (You can apply the same algorithm, building on the line-of-sight example we discussed earlier, in tile-based games too.)

The basic idea of the intercept algorithm is to be able to predict some future position of the prey and to move toward that position so as to reach it at the same time as the prey. This is illustrated in Figure 2-10.

Figure 2-10. Interception



At first glance it might appear that the predicted interception point is simply the point along the trajectory of the prey that is closest to the location of the predator. This is the shortest-distance-to-the-line problem, whereby the shortest distance from a point to the line is along a line segment that is perpendicular to the line. This is not necessarily the interception point because the shortest-distance problem does not consider the relative velocities between the predator and the prey. It might be that the predator will reach the shortest-distance point on the prey's trajectory before the prey arrives. In this case the predator will have to stop and wait for the prey to arrive if it is to be intercepted. This obviously won't work if the predator is a projectile being fired at a moving target such as an aircraft. If the scenario is a role-playing game, as soon as the player sees that the predator is in his path, he'll simply turn away.

To find the point where the predator and prey will meet at the same time, you must consider their relative velocities. So, instead of just knowing the prey's current position, the predator also must know the prey's current velocitythat is, its speed and heading. This information will be used to predict where the prey will be at some time in the future. Then, that predicted position will become the target toward which the predator will head to make the interception. The predator must then continuously monitor the prey's position and velocity, along with its own, and update the predicted interception point accordingly. This facilitates the predator changing course to adapt to any evasive maneuvers the prey might make. This, of course, assumes that the predator has some sort of steering capability.

At this point you should be asking how far ahead in time you should try to predict the prey's position. The answer is that it depends on the relative positions and velocities of both the predator and the prey. Let's consider the calculations involved one step at a time.

The first thing the predator must do is to calculate the relative velocity between itself and the prey. This is called the *closing velocity* and is simply the vector difference between the prey's velocity and the predator's:

$$\mathbf{V}_r = \mathbf{V}_{prey} - \mathbf{V}_{predator}$$

Here the relative, or closing, velocity vector is denoted $\mathbf{V}_r$. The second step involves calculating the *range to close*. That's the relative distance between the predator and the prey, which is equal to the vector difference

between the prey's current position and the predator's current position:

$$\mathbf{S}_r = \mathbf{S}_{prey} - \mathbf{S}_{predator}$$

Here the relative distance, or range, between the predator and prey is denoted $\mathbf{S}_r$. Now there's enough information to facilitate calculating the *time to close*.

The time to close is the average time it will take to travel a distance equal to the range to close while traveling at a speed equal to the closing speed, which is the magnitude of the closing velocity, or the relative velocity between the predator and prey. The time to close is calculated as follows:

$$t_c = |\mathbf{S}_r|/|\mathbf{V}_r|$$

The time to close, t_c, is equal to the magnitude of the range vector, $\mathbf{S}_r$, divided by the magnitude of the closing velocity vector, $\mathbf{V}_r$.

Now, knowing the time to close, you can predict where the prey will be t_c in the future. The current position of the prey is $\mathbf{S}_{prey}$ and it is traveling at $\mathbf{V}_{prey}$. Because speed multiplied by time yields average distance traveled, you can calculate how far the prey will travel over a time interval t_c traveling at $\mathbf{V}_{prey}$ and add it to the current position to yield the predicted position, as follows:

$$\mathbf{S}_t = \mathbf{S}_{prey} + (\mathbf{V}_{prey})(t_c)$$

Here, $\mathbf{S}_t$ is the predicted position of the prey t_c in the future. It's this predicted position, $\mathbf{S}_t$, that now becomes the target, or aim, point for the predator. To make the interception, the predator should head toward this point in much the same way as it headed toward the prey using the line-of-sight chase algorithm. In fact, all you need to do is to add a few lines of code to Example 2-8, the line-of-sight chase function, to convert it to an intercepting function. Example 2-9 shows the new function.

Example 2-9. Intercept function

```
void     DoIntercept(void)
{
     Vector   u, v;
     Bool     left = false;
     Bool     right = false;
     Vector   Vr, Sr, St;   // added this line
     Double   tc             // added this line
     // added these lines:
```

```
    Vr = Prey.vVelocity - Predator.vVelocity;
    Sr = Prey.vPosition - Predator.vPosition;
    tc = Sr.Magnitude() / Vr.Magnitude();
    St = Prey.vPosition + (Prey.vVelocity * tc);
    // changed this line to use St instead of Prey.vPosition:
    u = VRotate2D(-Predator.fOrientation,
                      (St - Predator.vPosition));
    // The remainder of this function is identical to the line-of-
    // sight chase function:
    u.Normalize();
    if (u.x <  -_TOL)
         left = true;
    else if (u.x > _TOL)
         right = true;
    Predator.SetThrusters(left, right);
}
```

The code in Example 2-9 is commented to highlight where we made changes to adapt the line-of-sight chase function shown in Example 2-8 to an intercept function. As you can see, we added a few lines of code to calculate the closing velocity, range, time to close, and predicted position of the prey, as discussed earlier. We also modified the line of code that calculates the target point in the predator's local coordinates to use the predicted position of the prey rather than its current position.

That's all there is to it. This function should be called every time through the game loop or physics engine loop so that the predator constantly updates the predicted interception point and its own trajectory.

The results of this algorithm as incorporated into example *AIDemo2-2* are illustrated in Figures 2-11 through 2-14.

Figure 2-11. Intercept scenario 1initial trajectories

figs/ch02_fig11.jpg

Figure 2-12. Intercept scenario 1interception

figs/ch02_fig12.jpg

Figure 2-13. Intercept scenario 2corrective action

Figure 2-14. Intercept scenario 2interception

Figure 2-11 illustrates a scenario in which the predator and prey start out from the lower left and right corners of the window, respectively. The prey moves at constant velocity from the lower right to the upper left of the window. At the same time the predator calculates the predicted interception point and heads toward it, continuously updating the predicted interception point and its heading accordingly. The predicted interception point is illustrated in this figure as the streak of dots ahead of the prey. Initially, the interception point varies as the predator turns toward the prey; however, things settle down and the interception point becomes fixed because the prey is moving at constant velocity.

After a moment the predator intercepts the prey, as shown in Figure 2-12.

Notice the difference between the path taken by the predator using the intercept algorithm versus that shown in Figure 2-9 using the line-of-sight algorithm. Clearly, this approach yields a shorter path and actually allows the predator and prey to cross the same point in space at the same time. In the line-of-sight algorithm, the predator chases the prey in a roundabout manner, ending up behind it. If the predator was not fast enough to keep up, it would never hit the prey and might get left behind.

Figure 2-13 shows how robust the algorithm is when the prey makes some evasive maneuvers.

Here you can see that the initial predicted intercept point, as illustrated by the trail of dots ahead of the prey, is identical to that shown in Figure 2-11. However, after the prey makes an evasive move to the right, the predicted intercept point is immediately updated and the predator takes corrective action so as to head toward the new intercept point. Figure 2-14 shows the resulting interception.

The interception algorithm we discussed here is quite robust in that it allows the predator to continuously update its trajectory to effect an interception. After experimenting with the demo, you'll see that an interception is made almost all the time.

Sometimes interceptions are not possible, however, and you should modify the algorithm we discussed here to deal with these cases. For example, if the predator is slower than the prey and if the predator somehow ends up behind the prey, it will be impossible for the predator to make an interception. It will never be able to catch up to the prey or get ahead of it to intercept it, unless the prey makes a maneuver so that the predator is no longer behind it. Even then, depending on the proximity, the prey still might not have enough speed to effect an

interception.

In another example, if the predator somehow gets ahead of the prey and is moving at the same speed or faster than the prey, it will predict an interception point ahead of both the prey and the predator such that neither will reach the interception pointthe interception point will constantly move away from both of them. In this case, the best thing to do is to detect when the predator is ahead of the prey and have the predator loop around or take some other action so as to get a better angle on the prey. You can detect whether the prey is behind the predator by checking the position of the prey relative to the predator in the predator's local coordinate system in a manner similar to that shown in Examples 2-8 and 2-9. Instead of checking the x-coordinate, you check the y-coordinate, and if it is negative, the prey is behind the predator and the preditor needs to turn around. An easy way to make the predator turn around is to have it go back to the line-of-sight algorithm instead of the intercept algorithm. This will make the predator turn right around and head back directly toward the prey, at which point the intercept algorithm can kick back in to effect an interception.

Earlier we told you that chasing and evading involves two, potentially three, distinct problems: deciding to chase or evade, actually effecting the chase or evasion, and obstacle avoidance. In this chapter we discussed the second problem of effecting the chase or evasion from a few different perspectives. These included basic chasing, line-of-sight chasing, and intercepting in both tiled and continuous environments. The methods we examined here are effective and give an illusion of intelligence. However, you can greatly enhance the illusions by combining these methods with other algorithms that can deal with the other parts of the problemnamely, deciding when and if to chase or evade, and avoiding obstacles while in pursuit or on the run. We'll explore several such algorithms in upcoming chapters.

Also, note that other algorithms are available for you to use to effect chasing or evading. One such method is based on the use of potential functions, which we discuss in Chapter 5.

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

Chapter 3. Pattern Movement

This chapter covers the subject of pattern movement. Pattern movement is a simple way to give the illusion of intelligent behavior. Basically, the computer-controlled characters move according to some predefined pattern that makes it appear as though they are performing complex, thought-out maneuvers. If you're old enough to remember the classic arcade game Galaga, you'll immediately know what pattern movement is all about. Recall the alien ships swooping down from the top and in from the sides of the screen, performing loops and turns, all the while shooting at your ship. The aliens, depending on their type and the level you were on, were capable of different maneuvers. These maneuvers were achieved using pattern movement algorithms.

Although Galaga is a classic example of using pattern movement, even modern video games use some form of pattern movement. For example, in a role-playing game or first-person shooter game, enemy monsters might patrol areas within the game world according to some predefined pattern. In a flight combat simulation, the enemy aircraft might perform evasive maneuvers according to some predefined patterns. Secondary creatures and nonplayer characters in different genres can be programmed to move in some predefined pattern to give the impression that they are wandering, feeding, or performing some task.

The standard method for implementing pattern movement takes the desired pattern and encodes the control data into an array or set of arrays. The control data consists of specific movement instructions, such as move forward and turn, that force the computer-controlled object or character to move according to the desired pattern. Using these algorithms, you can create a circle, square, zigzag, curveany type of pattern that you can encode into a concise set of movement instructions.

In this chapter, we'll go over the standard pattern movement algorithm in generic terms before moving on to two examples that implement variations of the standard algorithm. The first example is tile-based, while the second shows how to implement pattern movement in physically simulated environments in which you must consider certain caveats specific to such environments.

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

3.1 Standard Algorithm

The standard pattern movement algorithm uses lists or arrays of encoded instructions, or control instructions, that tell the computer-controlled character how to move each step through the game loop. The array is indexed each time through the loop so that a new set of movement instructions gets processed each time through.

Example 3-1 shows a typical set of control instructions.

Example 3-1. Control instructions data structure

```
ControlData {
    double turnRight;
    double turnLeft;
    double stepForward;
    double stepBackward;
    };
```

In this example, turnRight and turnLeft would contain the number of degrees by which to turn right or left. If this were a tile-based game in which the number of directions in which a character could head is limited, turnRight and turnLeft could mean turn right or left by one increment. stepForward and stepBackward would contain the number of distance units, or tiles, by which to step forward or backward.

This control structure also could include other instructions, such as fire weapon, drop bomb, release chaff, do nothing, speed up, and slow down, among many other actions appropriate to your game.

Typically you set up a global array or set of arrays of the control structure type to store the pattern data. The data used to initialize these pattern arrays can be loaded in from a data file or can be hardcoded

within the game; it really depends on your coding style and on your game's requirements.

Initialization of a pattern array, one that was hardcoded, might look something such as that shown in Example 3-2.

Example 3-2. Pattern initialization

```
Pattern[0].turnRight = 0;
Pattern[0].turnLeft = 0;
Pattern[0].stepForward = 2;
Pattern[0].stepBackward = 0;
Pattern[1].turnRight = 0;
Pattern[1].turnLeft = 0;
Pattern[1].stepForward = 2;
Pattern[1].stepBackward = 0;
Pattern[2].turnRight = 10;
Pattern[2].turnLeft = 0;
Pattern[2].stepForward = 0;
Pattern[2].stepBackward = 0;
Pattern[3].turnRight = 10;
Pattern[3].turnLeft = 0;
Pattern[3].stepForward = 0;
Pattern[3].stepBackward = 0;
Pattern[4].turnRight = 0;
Pattern[4].turnLeft = 0;
Pattern[4].stepForward = 2;
Pattern[4].stepBackward = 0;
Pattern[5].turnRight = 0;
Pattern[5].turnLeft = 0;
Pattern[5].stepForward = 2;
Pattern[5].stepBackward = 0;
Pattern[6].turnRight = 0;
Pattern[6].turnLeft = 10;
Pattern[6].stepForward = 0;
Pattern[6].stepBackward = 0;
                  .
                  .
                  .
```

In this example, the pattern instructs the computer-controlled character to move forward 2 distance units, move forward again 2 distance units, turn right 10 degrees, turn right again 10 degrees, move forward 2 distance units, move forward again 2 distance units, and turn left 10 degrees. This specific

pattern causes the computer-controlled character to move in a zigzag pattern.

To process this pattern, you need to maintain and increment an index to the pattern array each time through the game loop. Further, each time through the loop, the control instructions corresponding to the current index in the pattern array must be read and executed. Example 3-3 shows how such steps might look in code.

Example 3-3. Processing the pattern array

```
void     GameLoop(void)
{
     .
     .
     .
     Object.orientation + = Pattern[CurrentIndex].turnRight;
     Object.orientation -- = Pattern[CurrentIndex].turnLeft;
     Object.x + = Pattern[CurrentIndex].stepForward;
     Object.x -- = Pattern[CurrentIndex].stepBackward;
     CurrentIndex++;
     .
     .
     .
}
```

As you can see, the basic algorithm is fairly simple. Of course, implementation details will vary depending on the structure of your game.

It's also common practice to encode several different patterns in different arrays and have the computer select a pattern to execute at random or via some other decision logic within the game. Such techniques enhance the illusion of intelligence and lend more variety to the computer-controlled character's behavior.

All Online Books

Table of Contents
View as Frames

Previous Next

3.2 Pattern Movement in Tiled Environments

The approach we're going to use for tile-based pattern movement is similar to the method we used for tile-based line-of-sight chasing in Chapter 2. In the line-of-sight example, we used Bresenham's line algorithm to precalculate a path between a starting point and an ending point. In this chapter, we're going to use Bresenham's line algorithm to calculate various patterns of movement. As we did in Chapter 2, we'll store the row and column positions in a set of arrays. These arrays can then be traversed to move the computer-controlled character, a troll in this example, in various patterns.

In this chapter, paths will be more complex than just a starting point and an ending point. Paths will be made up of line segments. Each new segment will begin where the previous one ended. You need to make sure the last segment ends where the first one begins to make the troll moves in a repeating pattern. This method is particularly useful when the troll is in a guarding or patrolling mode. For example, you could have the troll continuously walk around the perimeter of a campsite and then break free of the pattern only if an enemy enters the vicinity. In this case, you could use a simple rectangular pattern.

You can accomplish this rectangular pattern movement by simply calculating four line segments. In Chapter 2, the line-of-sight function cleared the contents of the row and column path arrays each time it was executed. In this case, however, each line is only a segment of the overall pattern. Therefore, we don't want to initialize the path arrays each time a segment is calculated, but rather, append each new line path to the previous one. In this example, we're going to initialize the row and column arrays before the pattern is calculated. Example 3-4 shows the function that we used to initialize the row and column path arrays.

Example 3-4. Initialize path arrays

```
void    InitializePathArrays(void)
{
     int i;
```

```
     for (i=0;i<kMaxPathLength;i++)
           {
                 pathRow[i] = --1;
                 pathCol[i] = --1;
          }
}
```

As Example 3-4 shows, we initialize each element of both arrays to a value of -1. We use -1 because it's not a valid coordinate in the tile-based environment. Typically, in most tile-based environments, the upper leftmost coordinate is (0,0). From that point, the row and column coordinates increase to the size of the tile map. Setting unused elements in the path arrays to -1 is a simple way to indicate which elements in the path arrays are not used. This is useful when appending one line segment to the next in the path arrays. Example 3-5 shows the modified Bresenham line-of-sight algorithm that is used to calculate line segments.

Example 3-5. Calculate line segment

```
void ai_Entity::BuildPathSegment(void)
{
     int i;
     int nextCol=col;
     int nextRow=row;
     int deltaRow=endRow-row;
     int deltaCol=endCol-col;
     int stepCol;
     int stepRow;
     int currentStep;
     int fraction;
     int i;
     for (i=0;i<kMaxPathLength;i++)
           if ((pathRow[i]==-1) && (pathCol[i]==1))
                {
                      currentStep=i;
                      break;
               }
     if (deltaRow < 0) stepRow=-1; else stepRow=1;
     if (deltaCol < 0) stepCol=-1; else stepCol=1;
     deltaRow=abs(deltaRow*2);
     deltaCol=abs(deltaCol*2);
     pathRow[currentStep]=nextRow;
     pathCol[currentStep]=nextCol;
     currentStep++;
```

```
    if (currentStep>=kMaxPathLength)
        return;
    if (deltaCol > deltaRow)
        {
            fraction = deltaRow * 2 - deltaCol;
            while (nextCol != endCol)
                {
                    if (fraction >= 0)
                        {
                            nextRow += stepRow;
                            fraction = fraction - deltaCol;
                        }
                    nextCol = nextCol + stepCol;
                    fraction = fraction + deltaRow;
                    pathRow[currentStep]=nextRow;
                    pathCol[currentStep]=nextCol;
                    currentStep++;
                    if (currentStep>=kMaxPathLength)
                        return;
                }
        }
    else
        {
            fraction = deltaCol * 2 - deltaRow;
            while (nextRow != endRow)
                {
                    if (fraction >= 0)
                        {
                            nextCol = nextCol + stepCol;
                            fraction = fraction - deltaRow;
                        }
                    nextRow = nextRow + stepRow;
                    fraction = fraction + deltaCol;
                    pathRow[currentStep]=nextRow;
                    pathCol[currentStep]=nextCol;
                    currentStep++;
                    if (currentStep>=kMaxPathLength)
                        return;
                }
        }
}
```

For the most part, this algorithm is very similar to the line-of-sight movement algorithm shown in Example 2-7 from Chapter 2. The major difference is that we replaced the section of code that initializes the path arrays with a new section of code. In this case, we want each new line segment to be appended to the previous one, so we don't want to initialize the path arrays each time this function is called. The new section of code determines where to begin appending the line segment. This is where we rely on the fact that we used a value of -1 to initialize the path arrays. All you need to do is simply traverse the arrays and check for the first occurrence of the value -1. This is where the new line segment begins. Using the function in Example 3-6, we're now ready to calculate the first pattern. Here, we're going to use a simple rectangular patrolling pattern. Figure 3-1 shows the desired pattern.

Figure 3-1. Rectangular pattern movement

As you can see in Figure 3-1, we highlighted the vertex coordinates of the rectangular pattern, along with the desired direction of movement. Using this information, we can establish the troll's pattern using the BuildPathSegment function from Example 3-5. Example 3-6 shows how the rectangular pattern is initialized.

Example 3-6. Rectangular pattern

```
entityList[1].InitializePathArrays();
entityList[1].BuildPathSegment(10, 3, 18, 3);
entityList[1].BuildPathSegment(18, 3, 18, 12);
entityList[1].BuildPathSegment(18, 12, 10, 12);
entityList[1].BuildPathSegment(10, 12, 10, 3);
entityList[1].NormalizePattern();
entityList[1].patternRowOffset = 5;
entityList[1].patternColOffset = 2;
```

As you can see in Example 3-6, you first initialize the path arrays by calling the function InitializePathArrays. Then you use the coordinates shown in Figure 3-1 to calculate the four line segments that make up the rectangular pattern. After each line segment is calculated and stored in the path arrays, we make a call to NormalizePattern to adjust the resulting pattern so that it is represented in terms of relative coordinates instead of absolute coordinates. We do this so that the pattern is not tied to any specific starting position in the game world. Once the pattern is built and normalized, we can execute it from anywhere. Example 3-7 shows the NormalizePattern function.

Example 3-7. NormalizePattern function

```
void ai_Entity::NormalizePattern(void)
{
     int i;
     int rowOrigin=pathRow[0];
     int colOrigin=pathCol[0];
     for (i=0;i<kMaxPathLength;i++)
           if ((pathRow[i]==-1) && (pathCol[i]==-1))
                       {
                            pathSize=i-1;
                            break;
                            }

     for (i=0;i<=pathSize;i++)
           {
                 pathRow[i]=pathRow[i]-rowOrigin;
                 pathCol[i]=pathCol[i]-colOrigin;
           }
}
```

As you can see, all we do to normalize a pattern is subtract the starting position from all the positions stored in the pattern arrays. This yields a pattern in terms of relative coordinates so that we can execute it from anywhere in the game world.

Now that the pattern has been constructed we can traverse the arrays to make the troll walk in the rectangular pattern. You'll notice that the last two coordinates in the final segment are equal to the first two coordinates of the first line segment. This ensures that the troll walks in a repeating pattern.

You can construct any number of patterns using the BuildPathSegment function. You simply need to determine the vertex coordinates of the desired pattern and then calculate each line segment. Of course, you can use as few as two line segments or as many line segments as the program resources allow to create a movement pattern. Example 3-8 shows how you can use just two line segments to create a simple back-and-forth patrolling pattern.

Example 3-8. Simple patrolling pattern

```
     entityList[1].InitializePathArrays();
     entityList[1].BuildPathSegment(10, 3, 18, 3);
     entityList[1].BuildPathSegment(18, 3, 10, 3);
     entityList[1].NormalizePattern();
     entityList[1].patternRowOffset = 5;
     entityList[1].patternColOffset = 2;
```

Using the line segments shown in Example 3-8, the troll simply walks back and forth between coordinates (10,3) and (18,3). This could be useful for such tasks as patrolling near the front gate of a castle or protecting an area near a bridge. The troll could continuously repeat the pattern until an enemy comes within sight. The troll could then switch to a chasing or attacking state.

Of course, there is no real limit to how many line segments you can use to generate a movement pattern. You can use large and complex patterns for such tasks as patrolling the perimeter of a castle or continuously marching along a shoreline to guard against invaders. Example 3-9 shows a more complex pattern. This example creates a pattern made up of eight line segments.

Example 3-9. Complex patrolling pattern

```
entityList[1].BuildPathSegment(4, 2, 4, 11);
entityList[1].BuildPathSegment(4, 11, 2, 24);
entityList[1].BuildPathSegment(2, 24, 13, 27);
entityList[1].BuildPathSegment(13, 27, 16, 24);
entityList[1].BuildPathSegment(16, 24, 13, 17);
entityList[1].BuildPathSegment(13, 17, 13, 13);
entityList[1].BuildPathSegment(13, 13, 17, 5);
entityList[1].BuildPathSegment(17, 5, 4, 2);
entityList[1].NormalizePattern();
entityList[1].patternRowOffset = 5;
entityList[1].patternColOffset = 2;
```

Example 3-9 sets up a complex pattern that takes terrain elements into consideration. The troll starts on the west bank of the river, crosses the north bridge, patrols to the south, crosses the south bridge, and then returns to its starting point to the north. The troll then repeats the pattern. Figure 3-2 shows the pattern, along with the vertex points used to construct it.

Figure 3-2. Complex tile pattern movement

figs/ch03_fig02.jpg

As Figure 3-2 shows, this method of pattern movement allows for very long and complex patterns. This can be particularly useful when setting up long patrols around various terrain elements.

Although the pattern method used in Figure 3-2 can produce long and complex patterns, these patterns can appear rather repetitive and predictable. The next method we'll look at adds a random factor, while

still maintaining a movement pattern. In a tile-based environment, the game world typically is represented by a two-dimensional array. The elements in the array indicate which object is located at each row and column coordinate. For this next pattern movement method, we'll use a second two-dimensional array. This pattern matrix guides the troll along a predefined track. Each element of the pattern array contains either a 0 or a 1. The troll is allowed to move to a row and column coordinate only if the corresponding element in the pattern array contains a 1.

The first thing you must do to implement this type of pattern movement is to set up a pattern matrix. As Example 3-10 shows, you start by initializing the pattern matrix to all 0s.

Example 3-10. Initialize pattern matrix

```
for (i=0;i<kMaxRows;i++)
     for (j=0;j<kMaxCols;j++)
          pattern[i][j]=0;
```

After the entire pattern matrix is set to 0, you can begin setting the coordinates of the desired movement pattern to 1s. We're going to create a pattern by using another variation of the Bresenham line-of-sight algorithm that we used in Chapter 2. In this case, however, we're not going to save the row and column coordinates in path arrays. We're going to set the pattern matrix to 1 at each row and column coordinate along the line. We can then make multiple calls to the pattern line function to create complex patterns. Example 3-11 shows a way to set up one such pattern.

Example 3-11. Pattern Setup

```
BuildPatternSegment(3, 2, 16, 2);
BuildPatternSegment(16, 2, 16, 11);
BuildPatternSegment(16, 11, 9, 11);
BuildPatternSegment(9, 11, 9, 2);
BuildPatternSegment(9, 6, 3, 6);
BuildPatternSegment(3, 6, 3, 2);
```

Each call to the BuildPatternSegment function uses the Bresenham line algorithm to draw a new line segment to the pattern matrix. The first two function parameters are the row and column of the starting point, while the last two are the row and column of the ending point. Each point in the line becomes a 1 in the pattern matrix. This pattern is illustrated in Figure 3-3.

Figure 3-3. Track pattern movement

Figure 3-3 highlights each point where the pattern matrix contains a 1. These are the locations where the troll is allowed to move. You'll notice, however, that at certain points along the track there is more than one valid direction for the troll. We're going to rely on this fact to make the troll move in a less repetitive and predictable fashion.

Whenever it's time to update the troll's position, we'll check each of the eight surrounding elements in the pattern array to determine which are valid moves. This is demonstrated in Example 3-12.

Example 3-12. Follow pattern matrix

```
void ai_Entity::FollowPattern(void)
{
     int i,j;
     int possibleRowPath[8]={0,0,0,0,0,0,0,0};
     int possibleColPath[8]={0,0,0,0,0,0,0,0};
     int rowOffset[8]={-1,-1,-1, 0, 0, 1, 1, 1};
     int colOffset[8]={-1, 0, 1,-1, 1,-1, 0, 1};
     j=0;
     for (i=0;i<8;i++)
          if (pattern[row+rowOffset[i]][col+colOffset[i]]==1)
               if (!(((row+rowOffset[i])==previousRow) &&
                    ((col+colOffset[i])==previousCol)))
                    {
                         possibleRowPath[j]=row+rowOffset[i];
                         possibleColPath[j]=col+colOffset[i];
                         j++;
                    }
     i=Rnd(0,j-1);
     previousRow=row;
     previousCol=col;
     row=possibleRowPath[i];
     col=possibleColPath[i];
```

You start by checking the pattern matrix at each of the eight points around the troll's current position. Whenever you find a value of 1, you save that coordinate to the possibleRowPath and possibleColPath arrays. After each point is checked, you randomly select a new coordinate from the array of valid points found. The end result is that the troll won't always turn in the same direction when it reaches an intersection in the pattern matrix.

Note that the purpose of the rowOffset and colOffset variables shown in Example 3-12 is to avoid having to write eight conditional statements. Using a loop and adding these values to the row and column position traverses the eight adjacent tiles. For example, the first two elements, when added to the current row and column, are the tiles to the upper left of the current position.

You have to consider one other point when moving the troll. The troll's previous location always will be in the array of valid moves. Selecting that point can lead to an abrupt back-and-forth movement when updating the troll's position. Therefore, you should always track the troll's previous location using the previousRow and previousCol variables. Then you can exclude that position when building the array of valid moves.

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

3.3 Pattern Movement in Physically Simulated Environments

So far in this chapter we discussed how to implement patterned movement in environments where you can instruct your game characters to take discrete steps or turns; but what about physically simulated environments? Surely you can take advantage of the utility of pattern movement in physically simulated environments as well. The trouble is that the benefit of using physically simulated environmentsnamely, letting the physics take control of movementisn't conducive to forcing a physically simulated object to follow a specific pattern of movement. Forcing a physically simulated aircraft, for example, to a specific position or orientation each time through the game loop defeats the purpose of the underlying physical simulation. In physically simulated environments, you don't specify an object's position explicitly during the simulation. So, to implement pattern movement in this case, you need to revise the algorithms we discussed earlier. Specifically, rather than use patterns that set an object's position and orientation each time through the game loop, you need to apply appropriate control forces to the object to coax it, or essentially drive it, to where you want it to go.

You saw in the second example in Chapter 2 how we used steering forces to make the predator chase his prey. You can use these same steering forces to drive your physically simulated objects so that they follow some pattern. This is, in fact, an approximation to simulating an intelligent creature at the helm of such a physically simulated vehicle. By applying control forces, you essentially are mimicking the behavior of the driver piloting his vehicle in some pattern. The pattern can be anythingevasive maneuvers, patrol paths, stuntsso long as you can apply the appropriate control forces, such as thrust and steering forces.

Keep in mind, however, that you don't have absolute control in this case. The physics engine and the model that governs the capabilitiessuch as speed and turning radius, among othersof the object being simulated still control the object's overall behavior. Your input in the form of steering forces and thrust modulation, for example, is processed by the physics engine, and the resulting behavior is a function of all inputs to the physics engine, not just yours. By letting the physics engine maintain control, we can give the computer-controlled object some sense of intelligence without forcing the object to do something the physics model does not allow. If you violate the physics model, you run the risk of ruining the immersive experience realistic physics create. Remember, our goal is to enhance that immersiveness

with the addition of intelligence.

To demonstrate how to implement path movement in the context of physically simulated environments, we're going to use as a basis the scenario we described in Chapter 2 for chasing and intercepting in continuous environments. Recall that this scenario included two vehicles that we simulated using a simple two-dimensional rigid-body simulation. The computer controlled one vehicle, while the player controlled the other. In this chapter, we're going to modify that example, demonstration program AIDemo2-2, such that the computer-controlled vehicle moves according to predefined patterns. The resulting demonstration program is entitled AIDemo3-2, and you can download it from this book's Web site.

The approach we'll take in this example is very similar to the algorithms we discussed earlier. We'll use an array to store the pattern information and then step through this array, giving the appropriate pattern instructions to the computer-controlled vehicle. There are some key differences between the algorithms we discussed earlier and the one required for physics-based simulations. In the earlier algorithms, the pattern arrays stored discrete movement informationtake a step left, move a step forward, turn right, turn left, and so on. Further, each time through the game loop the pattern array index was advanced so that the next move's instructions could be fetched. In physics-based simulations, you must take a different approach, which we discuss in detail in the next section.

3.2.1 Control Structures

As we mentioned earlier, in physics-based simulations you can't force the computer-controlled vehicle to make a discrete step forward or backward. Nor can you tell it explicitly to turn left or right. Instead, you have to feed the physics engine control force information that, in effect, pilots the computer-controlled vehicle in the pattern you desire. Further, when a control forcesay, a steering forceis applied in physics-based simulations, it does not instantaneously change the motion of the object being simulated. These control forces have to act over time to effect the desired motion. This means you don't have a direct correspondence between the pattern array indices and game loop cycles; you wouldn't want that anyway. If you have a pattern array that contains a specific set of instructions to be executed each time you step through the simulation, the pattern arrays would be huge because the time steps typically taken in a physics-based simulation are very small.

To get around this, the control information contained in the pattern arrays we use here also contains information so that the computer knows how long, so to speak, each set of instructions should be applied. The algorithm works like this: the computer selects the first set of instructions from the pattern array and applies them to the vehicle being controlled. The physics engine processes those instructions each time step through the simulation until the conditions specified in the given set of instructions are met. At that point the next set of instructions from the pattern array are selected and applied. This process repeats all the way through the pattern array, or until the pattern is aborted for some other reason.

The code in Example 3-13 shows the pattern control data structure we set up for this example.

Example 3-13. Pattern movement control data structure

```
struct    ControlData {
     bool     PThrusterActive;
     bool     SThrusterActive;
     double   dHeadingLimit;
     double   dPositionLimit;
     bool     LimitHeadingChange;
     bool     LimitPositionChange;
};
```

Be aware that this control data will vary depending on what you are simulating in your game and how its underlying physics model works. In this case, the vehicle we're controlling is steered only by bow thruster forces. Thus, these are the only two control forces at our disposal with which we can implement some sort of pattern movement.

Therefore, the data structure shown in Example 3-13 contains two boolean members, PThrusterActive and SThrusterActive, which indicate whether each thruster should be activated. The next two members, dHeadingLimit and dPositionLimit, are used to determine how long each set of controls should be applied. For example, dHeadingLimit specifies a desired change in the vehicle's heading. If you want a particular instruction to turn the vehicle 45 degrees, you set this dHeadingLimit to 45. Note that this is a relative change in heading and not an absolute orientation. If the flag LimitHeadingChange is set to true, dHeadingLimit is checked each time through the simulation loop while the given pattern instruction is being applied. If the vehicle's heading has changed sufficiently relative to its last heading before this instruction was applied, the next instruction should be fetched.

Similar logic applies to dPositionLimit. This member stores the desired change in positionthat is, distance traveled relative to the position of the vehicle before the given set of instructions was applied. If LimitPositionChange is set to true, each time through the simulation loop the relative position change of the vehicle is checked against dPositionChange to determine if the next set of instructions should be fetched from the pattern array.

Before proceeding further, let us stress that the pattern movement algorithm we're showing you here works with relative changes in heading and position. The pattern instructions will be something such as move forward 100 ft, then turn 45 degrees to the left, then move forward another 100 ft, then turn 45 degrees to the right, and so on. The instructions will be absolute: move forward until you reach position $(x, y)_0$, then turn until you are facing southeast, then move until you reach position $(x, y)_1$, then turn until you are facing southwest, and so on.

Using relative changes in position and heading enables you to execute the stored pattern regardless of the location or initial orientation of the object being controlled. If you were to use absolute coordinates and compass directions, the patterns you use would be restricted near those coordinates. For example, you

could patrol a specific area on a map using some form of pattern, but you would not be able to patrol any area on a map with the specific pattern. The latter approach, using absolute coordinates, is consistent with the algorithm we showed you in the previous tile-based example. Further, such an approach is in line with waypoint navigation, which has its own merits, as we discuss in later chapters.

Because we're using relative changes in position and heading here, you also need some means of tracking these changes from one set of pattern instructions to the next. To this end, we defined another structure that stores the changes in state of the vehicle from one set of pattern instructions to the next. Example 3-14 shows the structure.

Example 3-14. State change tracking structure

```
struct    StateChangeData {
     Vector     InitialHeading;
     Vector     InitialPosition;
     double     dHeading;
     double     dPosition;
     int        CurrentControlID;
};
```

The first two members, InitialHeading and InitialPosition, are vectors that store the heading and position of the vehicle being controlled at the moment a set of pattern instructions is selected from the pattern array. Every time the pattern array index is advanced and a new set of instructions is fetched, these two members must be updated. The next two members, dHeading and dPosition, store the changes in position and heading as the current set of pattern instructions is being applied during the simulation. Finally, CurrentControlID stores the current index in the pattern array, which indicates the current set of pattern control instructions being executed.

3.2.2 Pattern Definition

Now, to define some patterns, you have to fill in an array of ControlData structures with appropriate steering control instructions corresponding to the desired movement pattern. For this example, we set up three patterns. The first is a square pattern, while the second is a zigzag pattern. In an actual game, you could use the square pattern to have the vehicle patrol an area bounded by the square. You could use the zigzag pattern to have the vehicle make evasive maneuvers, such as when Navy ships zigzag through the ocean to make it more difficult for enemy submarines to attack them with torpedoes. You can define control inputs for virtually any pattern you want to simulate; you can define circles, triangles, or any arbitrary path using this method. In fact, the third pattern we included in this example is an arbitrarily shaped pattern.

For the square and zigzag patterns, we set up two global arrays called PatrolPattern and ZigZagPattern, as shown in Example 3-15.

Example 3-15. Pattern array declarations

```
#define_PATROL_ARRAY_SIZE          8
#define _ZIGZAG_ARRAY_SIZE          4
ControlData          PatrolPattern[_PATROL_ARRAY_SIZE];
ControlData          ZigZagPattern[_ZIGZAG_ARRAY_SIZE];
StateChangeData      PatternTracking;
```

As you can see, we also defined a global variable called PatternTracking that tracks changes in position and heading as these patterns get executed.

Examples 3-16 and 3-17 show how each of these two patterns is initialized with the appropriate control data. We hardcoded the pattern initialization in this demo; however, in an actual game you might prefer to load in the pattern data from a data file. Further, you can optimize the data structure using a more concise encoding, as opposed to the structure we used here for the sake of clarity.

Example 3-16. Square patrol pattern initialization

```
    PatrolPattern[0].LimitPositionChange = true;
    PatrolPattern[0].LimitHeadingChange = false;
    PatrolPattern[0].dHeadingLimit = 0;
    PatrolPattern[0].dPositionLimit = 200;
    PatrolPattern[0].PThrusterActive = false;
    PatrolPattern[0].SThrusterActive = false;
    PatrolPattern[1].LimitPositionChange = false;
    PatrolPattern[1].LimitHeadingChange = true;
    PatrolPattern[1].dHeadingLimit = 90;
    PatrolPattern[1].dPositionLimit = 0;
    PatrolPattern[1].PThrusterActive = true;
    PatrolPattern[1].SThrusterActive = false;
    PatrolPattern[2].LimitPositionChange = true;
    PatrolPattern[2].LimitHeadingChange = false;
    PatrolPattern[2].dHeadingLimit = 0;
    PatrolPattern[2].dPositionLimit = 200;
    PatrolPattern[2].PThrusterActive = false;
    PatrolPattern[2].SThrusterActive = false;
    PatrolPattern[3].LimitPositionChange = false;
    PatrolPattern[3].LimitHeadingChange = true;
    PatrolPattern[3].dHeadingLimit = 90;
    PatrolPattern[3].dPositionLimit = 0;
    PatrolPattern[3].PThrusterActive = true;
```

```
PatrolPattern[3].SThrusterActive = false;
PatrolPattern[4].LimitPositionChange = true;
PatrolPattern[4].LimitHeadingChange = false;
PatrolPattern[4].dHeadingLimit = 0;
PatrolPattern[4].dPositionLimit = 200;
PatrolPattern[4].PThrusterActive = false;
PatrolPattern[4].SThrusterActive = false;
PatrolPattern[5].LimitPositionChange = false;
PatrolPattern[5].LimitHeadingChange = true;
PatrolPattern[5].dHeadingLimit = 90;
PatrolPattern[5].dPositionLimit = 0;
PatrolPattern[5].PThrusterActive = true;
PatrolPattern[5].SThrusterActive = false;
PatrolPattern[6].LimitPositionChange = true;
PatrolPattern[6].LimitHeadingChange = false;
PatrolPattern[6].dHeadingLimit = 0;
PatrolPattern[6].dPositionLimit = 200;
PatrolPattern[6].PThrusterActive = false;
PatrolPattern[6].SThrusterActive = false;
PatrolPattern[7].LimitPositionChange = false;
PatrolPattern[7].LimitHeadingChange = true;
PatrolPattern[7].dHeadingLimit = 90;
PatrolPattern[7].dPositionLimit = 0;
PatrolPattern[7].PThrusterActive = true;
PatrolPattern[7].SThrusterActive = false;
```

Example 3-17. Zigzag pattern initialization

```
ZigZagPattern[0].LimitPositionChange = true;
ZigZagPattern[0].LimitHeadingChange = false;
ZigZagPattern[0].dHeadingLimit = 0;
ZigZagPattern[0].dPositionLimit = 100;
ZigZagPattern[0].PThrusterActive = false;
ZigZagPattern[0].SThrusterActive = false;
ZigZagPattern[1].LimitPositionChange = false;
ZigZagPattern[1].LimitHeadingChange = true;
ZigZagPattern[1].dHeadingLimit = 60;
ZigZagPattern[1].dPositionLimit = 0;
ZigZagPattern[1].PThrusterActive = true;
ZigZagPattern[1].SThrusterActive = false;
ZigZagPattern[2].LimitPositionChange = true;
ZigZagPattern[2].LimitHeadingChange = false;
ZigZagPattern[2].dHeadingLimit = 0;
```

```
ZigZagPattern[2].dPositionLimit = 100;
ZigZagPattern[2].PThrusterActive = false;
ZigZagPattern[2].SThrusterActive = false;
ZigZagPattern[3].LimitPositionChange = false;
ZigZagPattern[3].LimitHeadingChange = true;
ZigZagPattern[3].dHeadingLimit = 60;
ZigZagPattern[3].dPositionLimit = 0;
ZigZagPattern[3].PThrusterActive = false;
ZigZagPattern[3].SThrusterActive = true;
```

The square pattern control inputs are fairly simple. The first set of instructions corresponding to array element [0] tells the vehicle to move forward by 200 distance units. In this case no steering forces are applied. Note here that the forward thrust acting on the vehicle already is activated and held constant. You could include thrust in the control structure for more complex patterns that include steering and speed changes.

The next set of pattern instructions, array element [1], tells the vehicle to turn right by activating the port bow thruster until the vehicle's heading has changed 90 degrees. The instructions in element [2] are identical to those in element [0] and they tell the vehicle to continue straight for 200 distance units. The remaining elements are simply a repeat of the first threeelement [3] makes another 90-degree right turn, element [4] heads straight for 200 distance units, and so on. The end result is eight sets of instructions in the array that pilot the vehicle in a square pattern.

In practice you could get away with only two sets of instructions, the first two shown in Example 3-16, and still achieve a square pattern. The only difference is that you'd have to repeat those two sets of instructions four times to form a square.

The zigzag controls are similar to the square controls in that the vehicle first moves forward a bit, then turns, then moves forward some more, and then turns again. However, this time the turns alternate from right to left, and the angle through which the vehicle turns is limited to 60 degrees rather than 90. The end result is that the vehicle moves in a zigzag fashion.

3.2.3 Executing the Patterns

In this example, we initialize the patterns in an Initialize function that gets called when the program first starts. Within that function, we also go ahead and initialize the PatternTracking structure by making a call to a function called InitializePatternTracking, which is shown in Example 3-18.

Example 3-18. InitializePatternTracking function

```
void InitializePatternTracking(void)
{
```

```
     PatternTracking.CurrentControlID = 0;
     PatternTracking.dPosition = 0;
     PatternTracking.dHeading = 0;
     PatternTracking.InitialPosition = Craft2.vPosition;
     PatternTracking.InitialHeading = Craft2.vVelocity;
     PatternTracking.InitialHeading.Normalize();
}
```

Whenever InitializePatternTracking is called, it copies the current position and velocity vectors for Craft2, the computer-controlled vehicle, and stores them in the state change data structure. The CurrentControlID, which is the index to the current element in the given pattern array, is set to 0, indicating the first element. Further, changes in position and heading are initialized to 0.

Of course, nothing happens if you don't have a function that actually processes these instructions. So, to that end, we defined a function called DoPattern, which takes a pointer to a pattern array and the number of elements in the array as parameters. This function must be called every time through the simulation loop to apply the pattern controls and step through the pattern array. In this example, we make the call to DoPattern within the UpdateSimulation function as illustrated in Example 3-19.

Example 3-19. UpdateSimulation function

```
void UpdateSimulation(void)
{
     .
     .
     .
     if(Patrol)
     {
          if(!DoPattern(PatrolPattern, _PATROL_ARRAY_SIZE))
               InitializePatternTracking();
     }
     if(ZigZag)
     {
          if(!DoPattern(ZigZagPattern, _ZIGZAG_ARRAY_SIZE))
               InitializePatternTracking();
     }
     .
     .
     .
     Craft2.UpdateBodyEuler(dt);
     .
     .
```

```
    .
}
```

In this case, we have two global variables, boolean flags, that indicate which pattern to execute. If Patrol is set to true, the square pattern is processed; whereas if ZigZag is set to true, the zigzag pattern is processed. These flags are mutually exclusive in this example.

Using such flags enables you to abort a pattern if required. For example, if in the midst of executing the patrol pattern, other logic in the game detects an enemy vehicle in the patrol area, you can set the Patrol flag to false and a Chase flag to true. This would make the computer-controlled craft stop patrolling and begin chasing the enemy.

The DoPattern function must be called before the physics engine processes all the forces and torques acting on the vehicles; otherwise, the pattern instructions will not get included in the force and torque calculations. In this case, that happens when the Craft2.UpdateBodyEuler (dt) call is made.

As you can see here in the if statements, DoPattern returns a boolean value. If the return value of DoPattern is set to false, it means the given pattern has been fully stepped through. In that case, the pattern is reinitialized so that the vehicle continues in that pattern. In a real game, you would probably have some other control logic to test for other conditions before deciding that the patrol pattern should be repeated. Detecting the presence of an enemy is a good check to make. Also, checking fuel levels might be appropriate depending on your game. You really can check anything here, it just depends on your game's requirements. This, by the way, ties into finite state machines, which we cover later.

3.2.4 DoPattern Function

Now, let's take a close look at the DoPattern function shown in Example 3-20.

Example 3-20. DoPattern function

```
bool     DoPattern(ControlData *pPattern, int size)
{
     int     i = PatternTracking.CurrentControlID;
     Vector     u;
     // Check to see if the next set of instructions in the pattern
     // array needs to be fetched.
     if(      (pPattern[i].LimitPositionChange &&
          (PatternTracking.dPosition >= pPattern[i].dPositionLimit)) ||
          (pPattern[i].LimitHeadingChange &&
          (PatternTracking.dHeading >= pPattern[i].dHeadingLimit)) )
     {
          InitializePatternTracking();
```

```
            i++;
            PatternTracking.CurrentControlID = i;
            if(PatternTracking.CurrentControlID >= size)
                return false;
        }
        // Calculate the change in heading since the time
        // this set of instructions was initialized.
        u = Craft2.vVelocity;
        u.Normalize();
        double P;
        P = PatternTracking.InitialHeading * u;
        PatternTracking.dHeading = fabs(acos(P) * 180 / pi);
        // Calculate the change in position since the time
        // this set of instructions was initialized.
        u = Craft2.vPosition - PatternTracking.InitialPosition;
        PatternTracking.dPosition = u.Magnitude();
        // Determine the steering force factor.
        double f;
        if(pPattern[i].LimitHeadingChange)
            f = 1 - PatternTracking.dHeading /
                pPattern[i].dHeadingLimit;
        else
            f = 1;
        if(f < 0.05) f = 0.05;
        // Apply steering forces in accordance with the current set
        // of instructions.
        Craft2.SetThrusters( pPattern[i].PThrusterActive,
                        pPattern[i].SThrusterActive, f);
        return true;
}
```

The first thing DoPattern does is copy the CurrentControlID, the current index to the pattern array, to a temporary variable, i, for use later.

Next, the function checks to see if either the change in position or change in heading limits have been reached for the current set of control instructions. If so, the tracking structure is reinitialized so that the next set of instructions can be tracked. Further, the index to the pattern array is incremented and tested to see if the end of the given pattern has been reached. If so, the function simply returns false at this point; otherwise, it continues to process the pattern.

The next block of code calculates the change in the vehicle's heading since the time the current set of instructions was initialized. The vehicle's heading is obtained from its velocity vector. To calculate the change in heading as an angle, you copy the velocity vector to a temporary vector, u in this case, and

normalize it. (Refer to the Appendix for a review of basic vector operations.) This gives the current heading as a unit vector. Then you take the vector dot product of the initial heading stored in the pattern-tracking structure with the unit vector, u, representing the current heading. The result is stored in the scalar variable, P. Next, using the definition of the vector dot product and noting that both vectors involved here are of unit length, you can calculate the angle between these two vectors by taking the inverse cosine of P. This yields the angle in radians, and you must multiply it by 180 and divide by pi to get degrees. Note that we also take the absolute value of the resulting angle because all we're interested in is the change in the heading angle.

The next block of code calculates the change in position of the vehicle since the time the current set of instructions was initialized. You find the change in position by taking the vector difference between the vehicle's current position and the initial position stored in the pattern tracking structure. The magnitude of the resulting vector yields the change in distance.

Next, the function determines an appropriate steering force factor to apply to the maximum available steering thruster force for the vehicle defined by the underlying physics model. You find the thrust factor by subtracting 1 from the ratio of the change in heading to the desired change in heading, which is the heading limit we are shooting for given the current set of control instructions. This factor is then passed to the SetThrusters function for the rigid-body object, Craft2, which multiplies the maximum available steering force by the given factor and applies the thrust to either the port or starboard side of the vehicle.

We clip the minimum steering force factor to a value of 0.05 so that some amount of steering force always is available. Because this is a physically simulated vehicle, it's inappropriate to just override the underlying physics and force the vehicle to a specific heading. You could do this, of course, but it would defeat the purpose of having the physics model in the first place. So, because we are applying steering forces, which act over time to steer the vehicle, and because the vehicle is not fixed to a guide rail, a certain amount of lag exists between the time we turn the steering forces on or off and the response of the vehicle. This means that if we steer hard all the way through the turn, we'll overshoot the desired change in heading. If we were targeting a 90-degree turn, we'd overshoot it a few degrees depending on the underlying physics model. Therefore, to avoid overshooting, we want to start our turn with full force but then gradually reduce the steering force as we get closer to our heading change target. This way, we turn smoothly through the desired change in heading, gradually reaching our goal without overshooting.

Compare this to turning a car. If you're going to make a right turn in your car, you initially turn the wheel all the way to the right, and as you progress through the turn you start to let the wheel go back the other way, gradually straightening your tires. You wouldn't turn the wheel hard over and keep it there until you turned 90 degrees and then suddenly release the wheel, trying not to overshoot.

Now, the reason we clip the minimum steering force is so that we actually reach our change in heading target. Using the "1 minus the change in heading ratio" formula means that the force factor goes to 0 in the limit as the actual change in heading goes to the desired change in heading. This means our change in heading would asymptote to the desired change in heading but never actually get there because the steering force would be too small or 0. The 0.05 factor is just a number we tuned for this particular

model. You'll have to tune your own physics models appropriately for what you are modeling.

3.2.5 Results

Figures 3-4 and 3-5 show the results of this algorithm for both the square and zigzag patterns. We took these screenshots directly from the example program available for download.

In Figure 3-4 you can see that a square pattern is indeed traced out by the computer-controlled vehicle. You should notice that the corners of the square are filleted nicelythat is, they are not hard right angles. The turning radius illustrated here is a function of the physics model for this vehicle and the steering force thrust modulation we discussed a moment ago. It will be different for your specific model, and you'll have to tune the simulation as always to get satisfactory results.

Figure 3-4. Square path

Figure 3-5 shows the zigzag pattern taken from the same example program. Again, notice the smooth turns. This gives the path a rather natural look. If this were an aircraft being simulated, one would also expect to see smooth turns.

Figure 3-5. Zigzag path

The pattern shown in Figure 3-6 consists of 10 instructions that tell the computer-controlled vehicle to go straight, turn 135 degrees right, go straight some more, turn 135 degrees left, and so on, until the pattern shown here is achieved.

Figure 3-6. Arbitrary pattern

Just for fun, we included an arbitrary pattern in this example to show you that this algorithm does not restrict you to simple patterns such as squares and zigzags. You can encode any pattern you can imagine into a series of instructions in the same manner, enabling you to achieve seemingly intelligent movement.

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

Chapter 4. Flocking

Often in video games, nonplayer characters must move in cohesive groups rather than independently. Let's consider some examples. Say you're writing an online role-playing game, and just outside the main town is a meadow of sheep. Your sheep would appear more realistic if they were grazing in a flock rather than walking around aimlessly. Perhaps in this same role-playing game is a flock of birds that prey on the game's human inhabitants. Here again, birds that hunt in flocks rather than independently would seem more realistic and pose the challenge to the player of dealing with somewhat cooperating groups of predators. It's not a huge leap of faith to see that you could apply such flocking behavior to giant ants, bees, rats, or sea creatures as well.

These examples of local fauna moving, grazing, or attacking in herds or flocks might seem like obvious ways in which you can use flocking behavior in games. With that said, you do not need to limit such flocking behavior to fauna and can, in fact, extend it to other nonplayer characters. For example, in a real-time strategy simulation, you can use group movement behavior for nonplayer unit movement. These units can be computer-controlled humans, trolls, orcs, or mechanized vehicles of all sorts. In a combat flight simulation, you can apply such group movement to computer-controlled squadrons of aircraft. In a first-person shooter, computer-controlled enemy or friendly squads can employ such group movement. You even can use variations on basic flocking behavior to simulate crowds of people loitering around a town square, for example.

In all these examples, the idea is to have the nonplayer characters move cohesively with the illusion of having purpose. This is as opposed to a bunch of units that move about, each with their own agenda and with no semblance of coordinated group movement whatsoever.

At the heart of such group behavior lie basic flocking algorithms such as the one presented by Craig Reynolds in his 1987 SIGGRAPH paper, "Flocks, Herds, and Schools: A Distributed Behavioral Model." You can apply the algorithm Reynolds presented in its original form to simulate flocks of birds, fish, or other creatures, or in modified versions to simulate group movement of units, squads, or air squadrons. In this chapter we're going to take a close look at a basic flocking algorithm and show how you can modify it to handle such situations as obstacle avoidance. For generality, we'll use the term *units* to refer to the individual entities comprising the

groupfor example, birds, sheep, aircraft, humans, and so onthroughout the remainder of this chapter.

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

4.1 Classic Flocking

Craig Reynolds coined the term *boids* when referring to his simulated flocks. The behavior he generated very closely resembles shoals of fish or flocks of birds. All the boids can be moving in one direction at one moment, and then the next moment the tip of the flock formation can turn and the rest of the flock will follow as a wave of turning boids propagates through the flock. Reynolds' implementation is leaderless in that no one boid actually leads the flock; in a sense they all sort of follow the group, which seems to have a mind of its own. The motion Reynolds' flocking algorithm generated is quite impressive. Even more impressive is the fact that this behavior is the result of three elegantly simple rules. These rules are summarized as follows:

Cohesion

Have each unit steer toward the average position of its neighbors.

Alignment

Have each unit steer so as to align itself to the average heading of its neighbors.

Separation

Have each unit steer to avoid hitting its neighbors.

It's clear from these three rule statements that each unit must be able to steer, for example, by application of steering forces. Further, each unit must be aware of its local surroundingsit has to know where its neighbors are located, where they're headed, and how close they are to it.

In physically simulated, continuous environments, you can steer by applying steering forces on the units being simulated. Here you can apply the same technique we used in the chasing, evading, and pattern movement

examples earlier in the book. (Refer to Chapter 2 and, specifically, to Figure 2-7 and surrounding discussion to see how you can handle steering.) We should point out that many flocking algorithms you'll find in other printed material or on the Web use particles to represent the units, whereas here we're going to use rigid bodies such as those we covered in Chapters 2 and 3. Although particles are easier to handle in that you don't have to worry about rotation, it's very likely that in your games the units won't be particles. Instead, they'll be units with a definite volume and a well defined front and back, which makes it important to track their orientations so that while moving around, they rotate to face the direction in which they are heading. Treating the units as rigid bodies enables you to take care of orientation.

For tiled environments, you can employ the line-of-sight methods we used in the tile-based chasing and evading examples to have the units steer, or rather, head toward a specific point. For example, in the case of the cohesion rule, you'd have the unit head toward the average location, expressed as a tile coordinate, of its neighbors. (Refer to the section "Line-of-Sight Chasing" in Chapter 2.)

To what extent is each unit aware of its neighbors? Basically, each unit is aware of its local surroundingsthat is, it knows the average location, heading, and separation between it and the other units in the group in its immediate vicinity. The unit does not necessarily know what the entire group is doing at any given time. Figure 4-1 illustrates a unit's local visibility.

Figure 4-1. Unit visibility

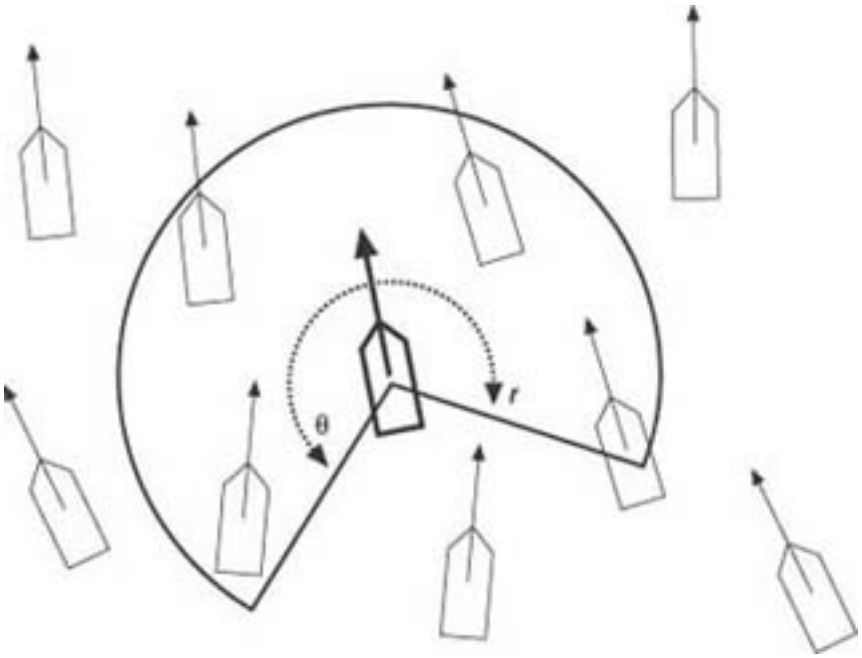
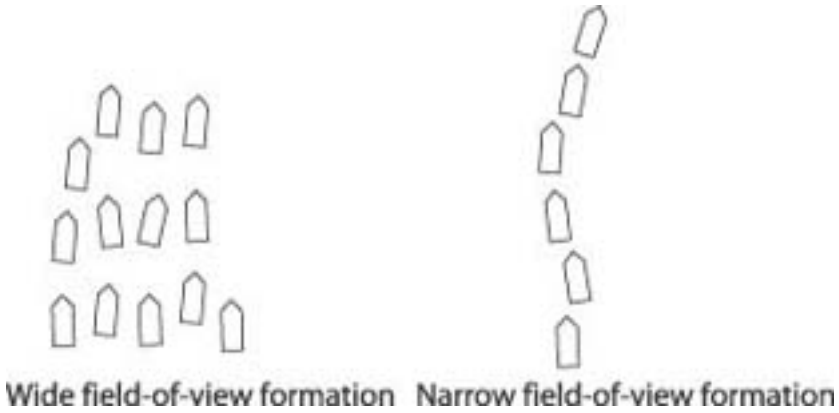


Figure 4-1 illustrates a unit (the bold one in the middle of the figure) with a visibility arc of radius *r* around it. The unit can see all other units that fall within that arc. The visible units are used when applying the flocking rules; all the other units are ignored. The visibility arc is defined by two parametersthe arc radius, *r*, and the angle, θ. Both parameters affect the resulting flocking motion, and you can tune them to your needs.

In general, a large radius will allow the unit to see more of the group, which results in a more cohesive flock. That is, the flock tends to splinter into smaller flocks less often because each unit can see where most or all of the units are and steer accordingly. On the other hand, a smaller radius tends to increase the likelihood of the flock to splinter, forming smaller flocks. A flock might splinter if some units temporarily lose sight of their neighbors, which can be their link to the larger flock. When this occurs, the detached units will splinter off into a smaller flock and could perhaps rejoin the others if they happen to come within sight again. Navigating around obstacles also can cause a flock to break up. In this case, a larger radius will help the flock to rejoin the group.

The other parameter, θ, measures the field of view, so to speak, of each unit. The widest field of view is, of course, 360 degrees. Some flocking algorithms use a 360-degree field of view because it is easier to implement; however, the resulting flocking behavior might be somewhat unrealistic. A more common field of view is similar to that illustrated in Figure 4-1, where there is a distinct blind spot behind each unit. Here, again, you can tune this parameter to your liking. In general, a wide field of view, such as the one illustrated in Figure 4-2 in which the view angle is approximately 270 degrees, results in well formed flocks. A narrow field of view, such as the one illustrated in Figure 4-2 in which the view angle is a narrow 45 degrees, results in flocks that tend to look more like a line of ants walking along a path.

Figure 4-2. Wide versus narrow field-of-view flock formations



Both results have their uses. For example, if you were simulating a squadron of fighter jets, you might use a wide field of view. If you were simulating a squad of army units sneaking up on someone, you might use a narrow field of view so that they follow each other in a line and, therefore, do not present a wide target as they make their approach. If you combine this latter case with obstacle avoidance, your units would appear to follow the point man as they sneak around obstacles.

Later, we'll build on the three flocking rules of cohesion, alignment, and separation to facilitate obstacle avoidance and leaders. But first, let's go through some example code that implements these three rules.

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

4.2 Flocking Example

The example we're going to look at involves simulating several units in a continuous environment. Here, we'll use the same rigid-body simulation algorithms we used in the chasing and pattern movement examples we discussed earlier. This example is named *AIDemo4*, and it's available for download from the book's Web site (http://www.oreilly.com/catalog/ai).

Basically, we're going to simulate about 20 units that will move around in flocks and interact with the environment and with a player. For this simple demonstration, interaction with the environment consists of avoiding circular objects. The flocking units interact with the player by chasing him.

4.2.1 Steering Model

For this example, we'll implement a steering model that is more or less identical to the one we used in the physics-based demo in Chapter 2. You can refer to Figure 2-8 and the surrounding discussion to refresh your memory on the steering model. Basically, we're going to treat each unit as a rigid body and apply a net steering force at the front end of the unit. This net steering force will point in either the starboard or port direction relative to the unit and will be the accumulation of steering forces determined by application of each flocking rule. This approach enables us to implement any number or combination of flocking ruleseach rule makes a small contribution to the total steering force and the net result is applied to the unit once all the rules are considered.

We should caution you that this approach does require some tuning to make sure no single rule dominates. That is, you don't want the steering force contribution from a given rule to be so strong that it always overpowers the contributions from other rules. For example, if we make the steering force contribution from the cohesion rule overpower the others, and say we implement an obstacle avoidance rule so that units try to steer away from objects, if the cohesion rule dominates, the units might stay together. Therefore, they will be unable to steer around objects and might run into or through them. To mitigate this sort of unbalance, we're going to do two things: first, we're going to modulate the steering force contribution from each rule; and second, we're going to

tune the steering model to make sure everything is balanced, at least most of the time.

Tuning will require trial and error. Modulating the steering forces will require that we write the steering force contribution from each rule in the form of an equation or response curve so that the contribution is not constant. Instead, we want the steering force to be a function of some key parameter important to the given rule.

Consider the avoidance rule for a moment. In this case, we're trying to prevent the units from running into each other, while at the same time enabling the units to get close to each other based on the alignment and cohesion rules. We want the avoidance rule steering force contribution to be small when the units are far away from each other, but we want the avoidance rule steering force contribution to be relatively large when the units are dangerously close to each other. This way, when the units are far apart, the cohesion rule can work to get them together and form a flock without having to fight the avoidance rule. Further, once the units are in a flock, we want the avoidance rule to be strong enough to prevent the units from colliding in spite of their tendency to want to stay together due to the cohesion and alignment rules. It's clear in this example that separation distance between units is an important parameter. Therefore, we want to write the avoidance steering force as a function of separation distance. You can use an infinite number of functions to accomplish this task; however, in our experience, a simple inverse function works fine. In this case, the avoidance steering force is inversely proportional to the separation distance. Therefore, large separation distances yield small avoidance steering forces, while small separation distances yield larger avoidance steering forces.

We'll use a similar approach for the other rules. For example, for alignment we'll consider the angle between a given unit's current heading relative to the average heading of its neighbors. If that angle is small, we want to make only a small adjustment to its heading, whereas if the angle is large, a larger adjustment is required. To achieve such behavior, we'll make the alignment steering force contribution directly proportional to the angle between the unit's current heading and the average heading of its neighbors. In the following sections, we'll look at and discuss some code that implements this steering model.

4.2.2 Neighbors

As we discussed earlier, each unit in a flock must be aware of its neighbors. Exactly how many neighbors each unit is aware of is a function of the field-of-view and view radius parameters shown in Figure 4-1. Because the arrangement of the units in a flock will change constantly, each unit must update its view of the world each time through the game loop. This means we must cycle through all the units in the flock collecting the required data. Note that we have to do this for each unit to acquire each unit's unique perspective. This neighbor search can become computationally expensive as the number of units grows large. The sample code we discuss shortly is written for clarity and is a good place to make some optimizations.

The example program entitled *AIDemo4*, which you can download from the book's web site (http://www.oreilly.com/catalog/ai"), is set up similar to the examples we discussed earlier in this book. In this example, you'll find a function called *UpdateSimulation* that is called each time through the game, or simulation, loop. This function is responsible for updating the positions of each unit and for drawing each unit to the display buffer. Example 4-

1 shows the *UpdateSimulation* function for this example.

Example 4-1. UpdateSimulation function

```
void     UpdateSimulation(void)
{
     double     dt = _TIMESTEP;
     int          i;
     // Initialize the back buffer:
     if(FrameCounter >= _RENDER_FRAME_COUNT)
     {
          ClearBackBuffer();
          DrawObstacles();
     }
     // Update the player-controlled unit (Units[0]):
     Units[0].SetThrusters(false, false, 1);
     if (IsKeyDown(VK_RIGHT))
          Units[0].SetThrusters(true, false, 0.5);
     if (IsKeyDown(VK_LEFT))
           Units[0].SetThrusters(false, true, 0.5);
     Units[0].UpdateBodyEuler(dt);
     if(Units[0].vPosition.x > _WINWIDTH) Units[0].vPosition.x = 0;
     if(Units[0].vPosition.x < 0) Units[0].vPosition.x = _WINWIDTH;
     if(Units[0].vPosition.y > _WINHEIGHT) Units[0].vPosition.y = 0;
     if(Units[0].vPosition.y < 0) Units[0].vPosition.y = _WINHEIGHT;
     if(FrameCounter >= _RENDER_FRAME_COUNT)
             DrawCraft(Units[0], RGB(0, 255, 0));
     // Update the computer-controlled units:
     for(i=1; i<_MAX_NUM_UNITS; i++)
     {
          DoUnitAI(i);
          Units[i].UpdateBodyEuler(dt);
          if(Units[i].vPosition.x > _WINWIDTH)
               Units[i].vPosition.x = 0;
          if(Units[i].vPosition.x < 0)
               Units[i].vPosition.x = _WINWIDTH;
          if(Units[i].vPosition.y > _WINHEIGHT)
               Units[i].vPosition.y = 0;
          if(Units[i].vPosition.y < 0)
```

```
                Units[i].vPosition.y = _WINHEIGHT;
            if(FrameCounter >= _RENDER_FRAME_COUNT)
            {
                if(Units[i].Leader)
                    DrawCraft(Units[i], RGB(255,0,0));
                else {
                    if(Units[i].Interceptor)
                        DrawCraft(Units[i], RGB(255,0,255));
                    else
                        DrawCraft(Units[i], RGB(0,0,255));
                }
            }
        }
        // Copy the back buffer to the screen:
        if(FrameCounter >= _RENDER_FRAME_COUNT) {
            CopyBackBufferToWindow();
            FrameCounter = 0;
        }  else
            FrameCounter++;
}
```

UpdateSimulation performs the usual tasks. It clears the back buffer upon which the scene will be drawn; it handles any user interaction for the player-controlled unit; it updates the computer-controlled units; it draws everything to the back buffer; and it copies the back buffer to the screen when done. The interesting part for our purposes is where the computer-controlled units are updated. For this task, *UpdateSimulation* loops through an array of computer-controlled units and, for each one, calls another function named *DoUnitAI*. All the fun happens in *DoUnitAI*, so we'll spend the remainder of this chapter looking at this function.

DoUnitAI handles everything with regard to the computer-controlled unit's movement. All the flocking rules are implemented in this function. Before the rules are implemented, however, the function has to collect data on the given unit's neighbors. Notice here that the given unit, the one currently under consideration, is passed in as a parameter. More specifically, an array index to the current unit under consideration is passed in to *DoUnitAI* as the parameter *i*.

Example 4-2 shows a snippet of the very beginning of *DoUnitAI*. This snippet contains only the local variable list and initialization code. Normally, we just brush over this kind of code, but because this code contains a relatively large number of local variables and because they are used often in the flocking calculations, it's worthwhile to go through it and state exactly what each one represents.

Example 4-2. DoUnitAI initialization

```
void     DoUnitAI(int i)
{
          int         j;
          int         N;      // Number of neighbors
          Vector     Pave;  // Average position vector
          Vector     Vave;  // Average velocity vector
          Vector     Fs;     // Net steering force
          Vector     Pfs;    // Point of application of Fs
          Vector     d, u, v, w;
          double     m;
          bool        InView;
          bool        DoFlock = WideView||LimitedView||NarrowView;
          int         RadiusFactor;
          // Initialize:
          Fs.x = Fs.y = Fs.z = 0;
          Pave.x = Pave.y = Pave.z = 0;
          Vave.x = Vave.y = Vave.z = 0;
          N = 0;
          Pfs.x = 0;
          Pfs.y = Units[i].fLength / 2.0f;
          .
          .
          .
}
```

We've already mentioned that the parameter, *i*, represents the array index to the unit currently under consideration. This is the unit for which all the neighbor data will be collected and the flocking rules will be implemented. The variable, *j*, is used as the array index to all other units in the *Units* array. These are the potential neighbors to *Units[i]*. *N* represents the number of neighbors that are within view of the unit currently under consideration. *Pave* and *Vave* will hold the average position and velocity vectors, respectively, of the *N* neighbors. *Fs* represents the net steering force to be applied to the unit under consideration. **Pfs** represents the location in body-fixed coordinates at which the steering force will be applied. *d*, *u*, *v*, and *w* are used to store various vector quantities that are calculated throughout the function. Such quantities include relative position vectors and heading vectors in both global and local coordinates. *m* is a multiplier variable that always will be either *+1* or *-1*. It's used to make the steering forces point in the directions we needthat is, to either the starboard or port side of the unit under consideration. *InView* is a flag that indicates whether a particular unit is within view of the unit under consideration. *DoFlock* is simply a flag that indicates whether to apply the flocking rules.

In this demo, you can turn flocking on or off. Further, you can implement three different visibility models to see how the flock behaves. These visibility models are called *WideView*, *LimitedView*, and *NarrowView*. Finally, *RadiusFactor* represents the *r* parameter shown in Figure 4-1, which is different for each visibility model. Note the field-of-view angle is different for each model as well; we'll talk more about this in a moment.

After all the local variables are declared, several of them are initialized explicitly. As you can see in Example 4-2, they are, for the most part, initialized to *0*. The variables you see listed there are the ones that are used to accumulate some valuefor example, to accumulate the steering force contributions from each rule, or to accumulate the number of neighbors within view, and so on. The only one not initialized to *0* is the vector **Pfs**, which represents the point of application of the steering force vector on the unit under consideration. Here, **Pfs** is set to represent a point on the very front and centerline of the unit. This will make the steering force line of action offset from the unit's center of gravity so that when the steering force is applied, the unit will move in the appropriate direction as well as turn and face the appropriate direction.

Upon completing the initialization of local variables, *DoUnitAI* enters a loop to gather information about the current unit's neighbors, if there are any.

Example 4-3 contains a snippet from *DoUnitAI* that performs all the neighbor checks and data collection. To this end, a loop is entered, the *j* loop, whereby each unit in the *Units* arrayexcept for *Units[0]* (the player-controlled unit) and *Units[i]* (the unit for which neighbors are being sought)is tested to see if it is within view of the current unit. If it is, its data is collected.

Example 4-3. Neighbors

```
.
.
.
for(j=1; j<_MAX_NUM_UNITS; j++)
{
     if(i!=j)
     {
          InView = false;
          d = Units[j].vPosition - Units[i].vPosition;
          w = VRotate2D(-Units[i].fOrientation, d);
          if(WideView)
          {
               InView = ((w.y > 0) || ((w.y < 0) &&
                         (fabs(w.x) >
                         fabs(w.y)*
                        _BACK_VIEW_ANGLE_FACTOR)));
```

```
                    RadiusFactor = _WIDEVIEW_RADIUS_FACTOR;
               }
               if(LimitedView)
               {
                    InView = (w.y > 0);
                    RadiusFactor = _LIMITEDVIEW_RADIUS_FACTOR;
               }
               if(NarrowView)
               {
                    InView = (((w.y > 0) && (fabs(w.x) <
                                  fabs(w.y)*
                                 _FRONT_VIEW_ANGLE_FACTOR)));
                    RadiusFactor = _NARROWVIEW_RADIUS_FACTOR;
               }
               if(InView)
               {
                    if(d.Magnitude() <= (Units[i].fLength *
                                              RadiusFactor))
                    {
                         Pave += Units[j].vPosition;
                         Vave += Units[j].vVelocity;
                         N++;
                    }
               }
               .
               .
               .
          }
     }
     .
     .
     .
```

After checking to make sure that *i* is not equal to *j*that is, we aren't checking the current unit against itselfthe function calculates the distance vector between the current unit, *Units[i]*, and *Units[j]*, which is simply the difference in their position vectors. This result is stored in the local variable, *d*. Next, *d* is converted from global coordinates to local coordinates fixed to *Units[i]*. The result is stored in the vector **w**.

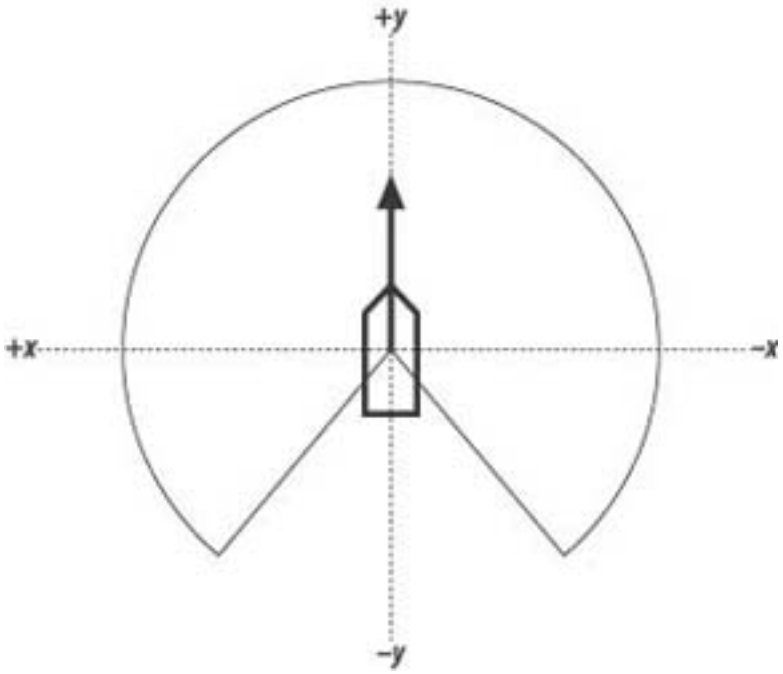
Next, the function goes on to check to see if *Units[j]* is within the field of view of *Units[i]*. This check is a

function of the field-of-view angle as illustrated in Figure 4-1; we'll check the radius value later, and only if the field-of-view check passes.

Now, because this example includes three different visibility models, three blocks of code perform field-of-view checks. These checks correspond to the wide-field-of-view, the limited-field-of-view, and the narrow-field-of-view models. As we discussed earlier, a unit's visibility influences the group's flocking behavior. You can toggle each model on or off in the example program to see their effect.

The wide-view model offers the greatest visibility and lends itself to easily formed and regrouped flocks. In this case, each unit can see directly in front of itself, to its sides, and behind itself, with the exception of a narrow blind spot directly behind itself. Figure 4-3 illustrates this field of view.

Figure 4-3. Wide field of view



The test to determine whether *Units[j]* falls within this field of view consists of two parts. First, if the relative position of *Units[j]* in terms of local coordinates fixed to the current unit, *Units[i]*, is such that its y-coordinate is positive, we know that *Units[j]* is within the field of view. Second, if the y-coordinate is negative, it could be either within the field of view or in the blind spot, so another check is required. This check looks at the x-coordinate to determine if *Units[j]* is located within the pie slice-shaped blind spot formed by the two straight lines that bound the visibility arc, as shown in Figure 4-3. If the absolute value of the x-coordinate of *Units[j]* is greater than some factor times the absolute value of the y-coordinate, we know *Units[j]* is located on the outside of the blind spotthat is, within the field of view. That factor times the absolute value of the y-coordinate calculation simply represents the straight lines bounding the field-of-view arc we mentioned earlier. The code that performs this check is shown in Example 4-3, but the key part is repeated here in Example 4-4 for convenience.

Example 4-4. Wide-field-of-view check

```
.
.
.
          if(WideView)
          {
               InView = ((w.y > 0) || ((w.y < 0) &&
                           (fabs(w.x) >
                        fabs(w.y)*
                       _BACK_VIEW_ANGLE_FACTOR)));
               RadiusFactor = _WIDEVIEW_RADIUS_FACTOR;
          }
.
.
.
```

In the code shown here, the *BACK_VIEW_ANGLE_FACTOR* represents a field-of-view angle factor. If it is set to a value of *1*, the field-of-view bounding lines will be 45 degrees from the x-axis. If the factor is greater than *1*, the lines will be closer to the x-axis, essentially creating a larger blind spot. Conversely, if the factor is less than *1*, the lines will be closer to the y-axis, creating a smaller blind spot.

You'll also notice here that the *RadiusFactor* is set to some predefined value, *_WIDEVIEW_RADIUS_FACTOR*. This factor controls the radius parameter shown in Figure 4-1. By the way, when tuning this example, this radius factor is one of the parameters that require adjustment to achieve the desired behavior.

The other two visibility model checks are very similar to the wide-view model; however, they each represent smaller and smaller fields of view. These two models are illustrated in Figures 4-4 and 4-5.

Figure 4-4. Limited field of view

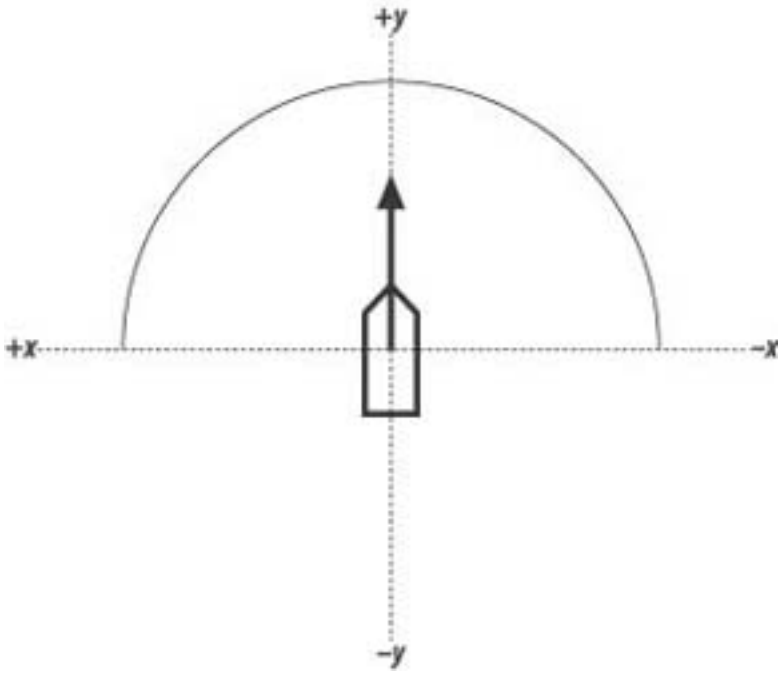
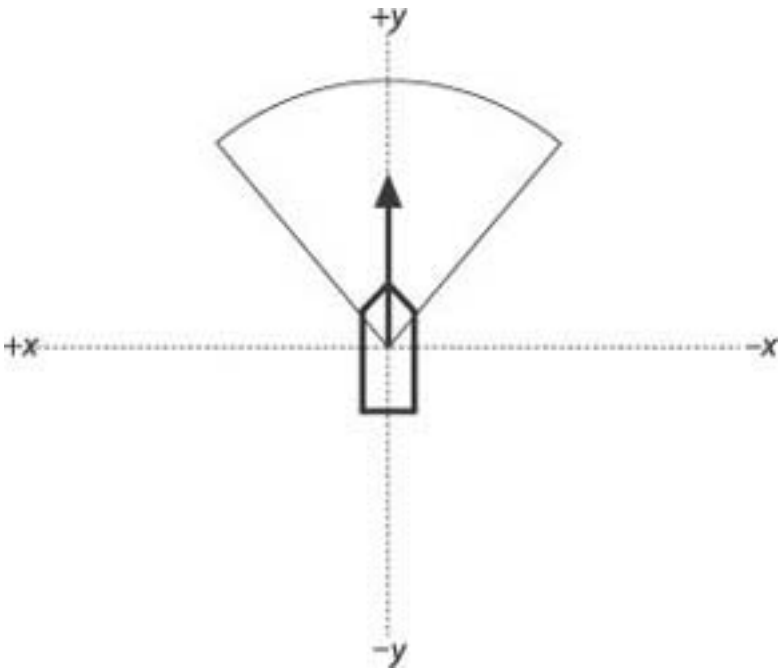


Figure 4-5. Narrow field of view



In the limited-view model, the visibility arc is restricted to the local positive y-axis of the unit. This means each unit cannot see anything behind itself. In this case, the test is relatively simple, as shown in Example 4-5, where all you need to determine is whether the *y*-coordinate of *Units[j]*, expressed in *Units[i]* local coordinates, is positive.

Example 4-5. Limited-field-of-view check

```
.
.
.
          if(LimitedView)
          {
               InView = (w.y > 0);
               RadiusFactor = _LIMITEDVIEW_RADIUS_FACTOR;
          }
.
.
.
```

The narrow-field-of-view model restricts each unit to seeing only what is directly in front of it, as illustrated in Figure 4-5.

The code check in this case is very similar to that for the wide-view case, where the visibility arc can be controlled by some factor. The calculations are shown in Example 4-6.

Example 4-6. Narrow-field-of-view check

```
.
.
.
          if(NarrowView)
          {
               InView = (((w.y > 0) && (fabs(w.x) <
                              fabs(w.y)*
                         _FRONT_VIEW_ANGLE_FACTOR)));
               RadiusFactor = _NARROWVIEW_RADIUS_FACTOR;
          }
.
.
.
```

In this case, the factor, *_FRONT_VIEW_ANGLE_FACTOR*, controls the field of view directly in front of the unit. If this factor is equal to *1*, the lines bounding the view cone are 45 degrees from the x-axis. If the factor is greater than *1*, the lines move closer to the x-axis, effectively increasing the field of view. If the factor is less than *1*, the lines move closer to the y-axis, effectively reducing the field of view.

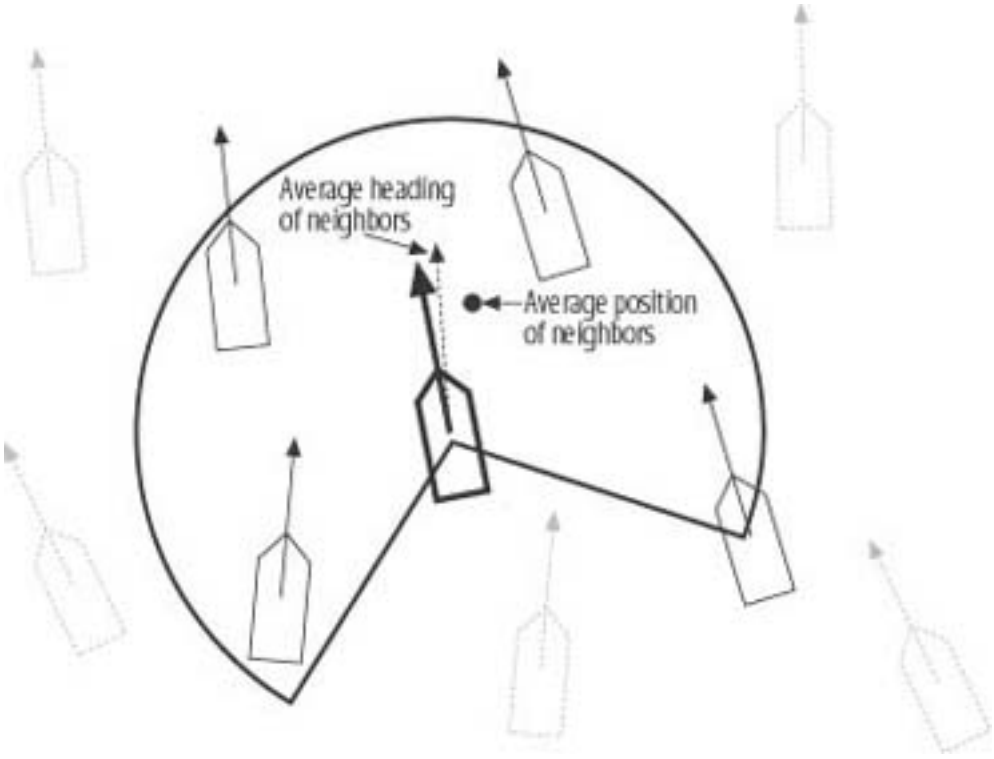
If any of these tests pass, depending on which view model you selected for this demo, another check is made to see if *Units[j]* is also within a specified distance from *Units[i]*. If *Units[j]* is within the field of view and within the specified distance, it is visible by *Units[i]* and will be considered a neighbor for subsequent calculations.

The last *if* block in Example 4-3 shows this distance test. If the magnitude of the **d** vector is less than the *Units [i]*'s length times the *RadiusFactor*, *Units[j]* is close enough to *Units[i]* to be considered a neighbor. Notice how this prescribed separation threshold is specified in terms of the unit's length times some factor. You can use any value here depending on your needs, though you'll have to tune it for your particular game; however, we like using the radius factor times the unit's length because it scales. If for some reason you decide to change the scale (the dimensions) of your game world, including the units in the game, their visibility will scale proportionately and you won't have to go back and tune some new visibility distance at the new scale.

4.2.3 Cohesion

Cohesion implies that we want all the units to stay together in a group; we don't want each unit breaking from the group and going its separate way. As we stated earlier, to satisfy this rule, each unit should steer toward the average position of its neighbors. Figure 4-6 illustrates a unit surrounded by several neighbors. The small dashed circle in the figure represents the average position of the four neighbors that are within view of the unit shown in bold lines with the visibility arc around itself.

Figure 4-6. Average position and heading of neighbors



The average position of neighbors is fairly easy to calculate. Once the neighbors have been identified, their

average position is the vector sum of their respective positions divided by the total number of neighbors (a scalar). The result is a vector representing their average position. Example 4-3 already shows where the positions of the neighbors are summed once they've been identified. The relevant code is repeated here in Example 4-7 for convenience.

Example 4-7. Neighbor position summation

```
.
.
.
if(InView)
{
     if(d.Magnitude() <= (Units[i].fLength *
                                RadiusFactor))
     {
          Pave += Units[j].vPosition;
          Vave += Units[j].vVelocity;
          N++;
     }
}
.
.
.
```

The line that reads *Pave += Units[j].vPosition;* sums the position vectors of all neighbors. Remember, *Pave* and *vPosition* are *Vector* types, and the overloaded operators take care of vector addition for us.

After *DoUnitAI* takes care of identifying and collecting information on neighbors, you can apply the flocking rules. The first one handled is the cohesion rule, and the code in Example 4-8 shows how to do this.

Example 4-8. Cohesion rule

```
.
.
.
// Cohesion Rule:
if(DoFlock && (N > 0))
{
     Pave = Pave / N;
```

```
            v = Units[i].vVelocity;
            v.Normalize();
            u = Pave - Units[i].vPosition;
            u.Normalize();
            w = VRotate2D(-Units[i].fOrientation, u);
            if(w.x < 0) m = -1;
            if(w.x > 0) m = 1;
            if(fabs(v*u) < 1)
                  Fs.x += m * _STEERINGFORCE * acos(v * u) / pi;
      }
      .
      .
      .
```

Notice that the first thing this block of code does is check to make sure the number of neighbors is greater than zero. If so, we can go ahead and calculate the average position of the neighbors. Do this by taking the vector sum of all neighbor positions, *Pave*, and dividing by the number of neighbors, *N*.

Next, the heading of the current unit under consideration, *Units[i]*, is stored in **v** and normalized. It will be used in subsequent calculations. Now the displacement between *Units[i]* and the average position of its neighbors is calculated by taking the vector difference between *Pave* and *Units[i]*'s position. The result is stored in **u** and normalized. **u** is then rotated from global coordinates to local coordinates fixed to *Units[i]* and the result is stored in *w*. This gives the location of the average position of *Units[i]*'s neighbors relative to *Units[i]*'s current position.

Next, the multiplier, *m*, for the steering force is determined. If the x-coordinate of *w* is greater than zero, the average position of the neighbors is located to the starboard side of *Units[i]* and it has to turn left (starboard). If the x-coordinate of *w* is less than zero, *Units[i]* must turn right (port side).

Finally, a quick check is made to see if the dot product between the unit vectors **v** and **u** is less than 1 and greater than minus -1.[*] This must be done because the dot product will be used when calculating the angle between these two vectors, and the arc cosine function takes an argument between +/-1.

[*] Refer to the Appendix for a review of the vector dot product operation.

The last line shown in Example 4-8 is the one that actually calculates the steering force satisfying the cohesion rule. In that line the steering force is accumulated in *Fs.x* and is equal to the direction factor, *m*, times the prescribed maximum steering force times the angle between the current unit's heading and the vector from it to the average position of its neighbors divided by *pi*. The angle between the current unit's heading and the vector

to the average position of its neighbors is found by taking the arc cosine of the dot product of vectors **v** and **u**. This comes from the definition of dot product. Note that the two vectors, **v** and **u**, are unit vectors. Dividing the resulting angle by *pi* yields a scale factor that gets applied to the maximum steering force. Basically, the steering force being accumulated in *Fs.x* is a linear function of the angle between the current unit's heading and the vector to the average position of its neighbors. This means that if the angle is large, the steering force will be relatively large, whereas if the angle is small, the steering force will be relatively small. This is exactly what we want. If the current unit is heading in a direction far from the average position of its neighbors, we want it to make a harder corrective turn. If it is heading in a direction not too far off from the average neighbor position, we want smaller corrections to its heading.

4.2.4 Alignment

Alignment implies that we want all the units in a flock to head in generally the same direction. To satisfy this rule, each unit should steer so as to try to assume a heading equal to the average heading of its neighbors. Referring to Figure 4-6, the bold unit in the center is moving along a given heading indicated by the bold arrow attached to it. The light, dashed vector also attached to it represents the average heading of its neighbors. Therefore, for this example, the bold unit needs to steer toward the right.

We can use each unit's velocity vector to determine its heading. Normalizing each unit's velocity vector yields its heading vector. Example 4-7 shows how the heading data for a unit's neighbors is collected. The line *Vave += Units[j].vVelocity;* accumulates each neighbor's velocity vector in *Vave* in a manner similar to how positions were accumulated in *Pave*.

Example 4-9 shows how the alignment steering force is determined for each unit. The code shown here is almost identical to that shown in Example 4-8 for the cohesion rule. Here, instead of dealing with the average position of neighbors, the average heading of the current unit's neighbors is first calculated by dividing *Vave* by the number of neighbors, *N*. The result is stored in **u** and then normalized, yielding the average heading vector.

Example 4-9. Alignment rule

```
.
.
.
// Alignment Rule:
if(DoFlock && (N > 0))
{
     Vave = Vave / N;
     u = Vave;
     u.Normalize();
     v = Units[i].vVelocity;
```

```
            v.Normalize();
            w = VRotate2D(-Units[i].fOrientation, u);
            if(w.x < 0) m = -1;
            if(w.x > 0) m = 1;
            if(fabs(v*u) < 1)
                  Fs.x += m * _STEERINGFORCE * acos(v * u) / pi;
      }
      .
      .
      .
```

Next, the heading of the current unit, *Units[i]*, is determined by taking its velocity vector and normalizing it. The result is stored in **v**. Now, the average heading of the current unit's neighbors is rotated from global coordinates to local coordinates fixed to *Units[i]* and stored in vector **w**. The steering direction factor, *m*, is then calculated in the same manner as before. And, as in the cohesion rule, the alignment steering force is accumulated in *Fs.x*.

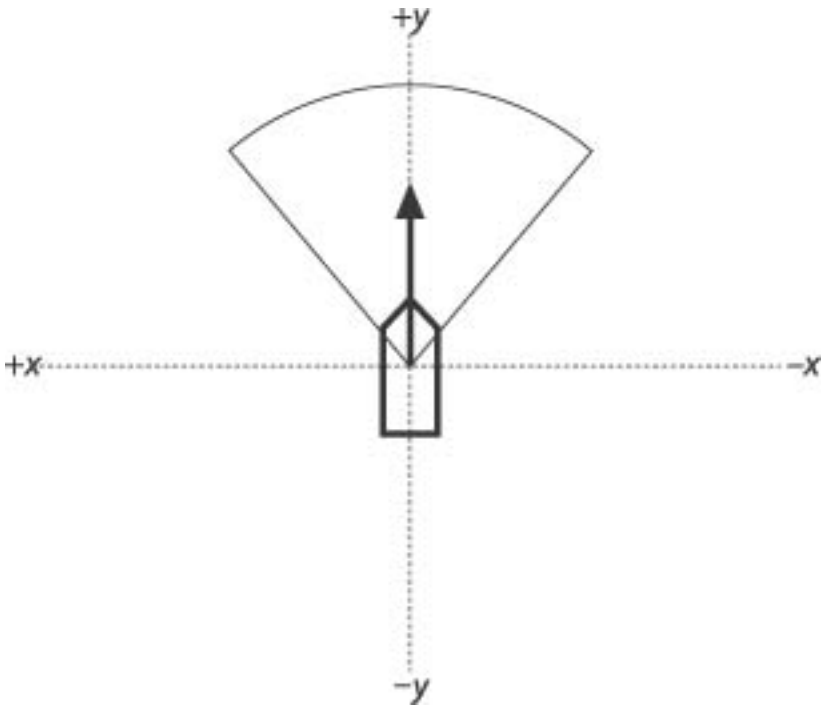
In this case, the steering force is a linear function of the angle between the current unit's heading and the average heading of its neighbors. Here again, we want small steering corrections to be made when the current unit is heading in a direction fairly close to the average of its neighbors, whereas we want large steering corrections to be made if the current unit is heading in a direction way off from its neighbors' average heading.

4.2.5 Separation

Separation implies that we want the units to maintain some minimum distance away from each other, even though they might be trying to get closer to each other as a result of the cohesion and alignment rules. We don't want the units running into each other or, worse yet, coalescing at a coincident position. Therefore, we'll enforce separation by requiring the units to steer away from any neighbor that is within view and within a prescribed minimum separation distance.

Figure 4-7 illustrates a unit that is too close to a given unit, the bold one. The outer arc centered on the bold unit is the visibility arc we've already discussed. The inner arc represents the minimum separation distance. Any unit that moves within this minimum separation arc will be steered clear of it by the bold unit.

Figure 4-7. Separation



The code to handle separation is just a little different from that for cohesion and alignment because for separation, we need to look at each individual neighbor when determining suitable steering corrections rather than some average property of all the neighbors. It is convenient to include the separation code within the same *j* loop shown in Example 4-3 where the neighbors are identified. The new *j* loop, complete with the separation rule implementation, is shown in Example 4-10.

Example 4-10. Neighbors and separation

```
.
.
.
for(j=1; j<_MAX_NUM_UNITS; j++)
{
  if(i!=j)
  {
       InView = false;
       d = Units[j].vPosition - Units[i].vPosition;
       w = VRotate2D(-Units[i].fOrientation, d);
       if(WideView)
       {
            InView = ((w.y > 0) || ((w.y < 0) &&
                  (fabs(w.x) >
                  fabs(w.y)*_BACK_VIEW_ANGLE_FACTOR)));
            RadiusFactor = _WIDEVIEW_RADIUS_FACTOR;
```

```
            }
            if(LimitedView)
            {
                 InView = (w.y > 0);
                 RadiusFactor = _LIMITEDVIEW_RADIUS_FACTOR;
            }
            if(NarrowView)
            {
                 InView = (((w.y > 0) && (fabs(w.x) <
                      fabs(w.y)*_FRONT_VIEW_ANGLE_FACTOR)));
                 RadiusFactor = _NARROWVIEW_RADIUS_FACTOR;
            }
            if(InView)
            {
                 if(d.Magnitude() <= (Units[i].fLength *
                                      RadiusFactor))
                 {
                      Pave += Units[j].vPosition;
                      Vave += Units[j].vVelocity;
                      N++;
                 }
            }
            if(InView)
            {
                 if(d.Magnitude() <=
                    (Units[i].fLength * _SEPARATION_FACTOR))
                 {
                      if(w.x < 0) m = 1;
                      if(w.x > 0) m = -1;
                      Fs.x += m * _STEERINGFORCE *
                              (Units[i].fLength *
                               _SEPARATION_FACTOR) /
                               d.Magnitude();
                 }
            }
       }
  }
  .
  .
```

The last *if* block contains the new separation rule code. Basically, if the *j* unit is in view and if it is within a distance of *Units[i].fLength *_SEPARATION_FACTOR* from the current unit, *Units[i]*, we calculate and apply a steering correction. Notice that *d* is the distance separating *Units[i]* and *Units[j]*, and was calculated at the beginning of the *j* loop.

Once it has been determined that *Units[j]* presents a potential collision, the code proceeds to calculate the corrective steering force. First, the direction factor, *m*, is determined so that the resulting steering force is of such a direction that the current unit, *Units[i]*, steers away from *Units[j]*. In this case, *m* takes on the opposite sense, as in the cohesion and alignment calculations.

As in the cases of cohesion and alignment, steering forces get accumulated in *Fs.x*. In this case, the corrective steering force is inversely proportional to the actual separation distance. This will make the steering correction force greater the closer *Units[j]* gets to the current unit. Notice here again that the minimum separation distance is scaled as a function of the unit's length and some prescribed separation factor. This occurs so that separation scales just like visibility, as we discussed earlier.

We also should mention that even though separation forces are calculated here, units won't always avoid each other with 100% certainty. Sometimes the sum of all steering forces is such that one unit is forced very close to or right over an adjacent unit. Tuning all the steering force parameters helps to mitigate, though not eliminate, this situation. You could set the separation steering force so high as to override any other forces, but you'll find that the units' behavior when in close proximity to each other appears very erratic. Further, it will make it difficult to keep flocks together. In the end, depending on your game's requirements, you still might have to implement some sort of collision detection and response algorithm similar to that discussed in *Physics for Game Developers* (O'Reilly) to handle cases in which two or more units run into each other.

You also should be aware that visibility has an important effect on separation. For example, while in the wide-view visibility model, the units maintain separation very effectively; however, in the narrow-view model the units fail to maintain side-to-side separation. This is because their views are so restricted, they are unaware of other units right alongside them. If you go with such a limited-view model in your games, you'll probably have to use a separate view model, such as the wide-view model, for the separation rule. You can easily change this example to use such a separate model by replacing the last *if* block's condition to match the logic for determining whether a unit is in view according to the wide-view model.

Once all the flocking rules are implemented and appropriate steering forces are calculated for the current unit, *DoUnitAI* stores the resulting steering forces and point of application in the current unit's member variables. This is shown in Example 4-11.

Example 4-11. Set Units[i] member variables

```
void DoUnitAI(int i)
        // Do all steering force calculations...
        .
        .
        .
        Units[i].Fa = Fs;
        Units[i].Pa = Pfs;
}
```

Once *DoUnitAI* returns, *UpdateSimulation* becomes responsible for applying the new steering forces and updating the positions of the units (see Example 4-1).

Previous Next
Top

All Online Books

Table of Contents
View as Frames

Previous Next

4.3 Obstacle Avoidance

The flocking rules we discussed so far yield impressive results. However, such flocking behavior would be far more realistic and useful in games if the units also could avoid running into objects in the game world as they move around in a flock. As it turns out, adding such obstacle avoidance behavior is a relatively simple matter. All we have to do is provide some mechanism for the units to see ahead of them and then apply appropriate steering forces to avoid obstacles in their paths.

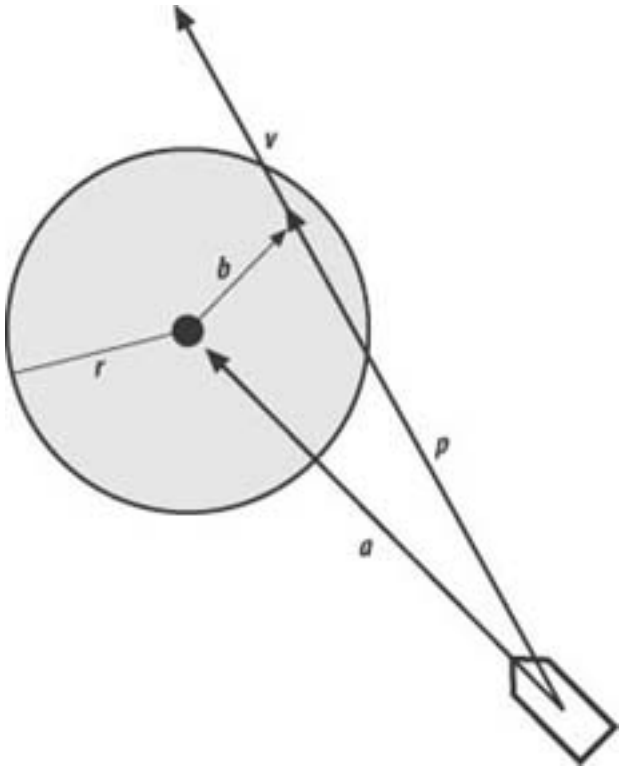
In this example, we'll consider a simple idealization of an obstaclewe'll consider them as circles. This need not be the case in your games; you can apply the same general approach we'll apply here for other obstacle shapes as well. The only differences will, of course, be geometry, and how you mathematically determine whether a unit is about to run into the obstacle.

To detect whether an obstacle is in the path of a unit, we'll borrow from robotics and outfit our units with virtual *feelers*. Basically, these feelers will stick out in front of the units, and if they hit something, this will be an indication to the units to turn. We'll assume that each unit can see obstacles to the extent that we can calculate to which side of the unit the obstacle is located. This will tell us whether to turn right or left.

The model we just described isn't the only one that will work. For example, you could outfit your units with more than one feelersay, three sticking out in three different directions to sense not only whether the obstacle is present, but also to which side of the unit it is located. Wide units might require more than one feeler so that you can be sure the unit won't sideswipe an obstacle. In 3D you could use a virtual volume that extends out in front of the unit. You then could test this volume against the game-world geometry to determine an impending collision with an obstacle. You can take many approaches.

Getting back to the approach we'll discuss, take a look at Figure 4-8 to see how our single virtual feeler will work in geometric terms. The vector, **v**, represents the feeler. It's of some prescribed finite length and is collinear with the unit's heading. The large shaded circle represents an obstacle. To determine whether the feeler intersects the obstacle at some point, we need to apply a little vector math.

Figure 4-8. Obstacle avoidance



First, we calculate the vector, **a**. This is simply the difference between the unit's and the obstacle's positions. Next, we project **a** onto **v** by taking their dot product. This yields vector **p**. Subtracting vector **p** from **a** yields vector **b**. Now to test whether **v** intersects the circle somewhere we need to test two conditions. First, the magnitude of **p** must be less than the magnitude of **v**. Second, the magnitude of **b** must be less than the radius of the obstacle, *r*. If both of these tests pass, corrective steering is required; otherwise, the unit can continue on its current heading.

The steering force to be applied in the event of an impending collision is calculated in a manner similar to the flocking rules we discussed earlier. Basically, the required force is calculated as inversely proportional to the distance from the unit to the center of the obstacle. More specifically, the steering force is a function of the prescribed maximum steering force times the ratio of the magnitude of **v** to the magnitude of a. This will make the steering correction greater the closer the unit is to the obstacle, where there's more urgency to get out of the way.

Example 4-12 shows the code that you must add to *DoUnitAI* to perform these avoidance calculations. You insert this code just after the code that handles the three flocking rules. Notice here that all the obstacles in the game world are looped through and checked to see if there's an impending collision. Here again, in practice you'll want to optimize this code. Also notice that the corrective steering force is accumulated in the same *Fs.x* member variable within which the other flocking rule steering forces were accumulated.

Example 4-12. Obstacle avoidance

```
.
.
.
Vector     a, p, b;
for(j=0; j<_NUM_OBSTACLES; j++)
{
     u = Units[i].vVelocity;
     u.Normalize();
     v = u * _COLLISION_VISIBILITY_FACTOR *
         Units[i].fLength;
     a = Obstacles[j] - Units[i].vPosition;
     p = (a * u) * u;
     b = p - a;
     if((b.Magnitude() < _OBSTACLE_RADIUS) &&
         (p.Magnitude() < v.Magnitude()))
     {
          // Impending collision...steer away
          w = VRotate2D(-Units[i].fOrientation, a);
          w.Normalize();
          if(w.x < 0) m = 1;
          if(w.x > 0) m = -1;
          Fs.x += m * _STEERINGFORCE *
                  (_COLLISION_VISIBILITY_FACTOR *
                  Units[i].fLength)/a.Magnitude();
     }
}
.
.
.
```

If you download and run this example, you'll see that even while the units form flocks, they'll still steer well clear of the randomly placed circular objects. It is interesting to experiment with the different visibility models to see how the flocks behave as they encounter obstacles. With the wide-visibility model the flock tends to split and go around the obstacles on either side. In some cases, they regroup quite readily while in others they don't. With the limited- and narrow-visibility models, the units tend to form single-file lines that flow smoothly around obstacles, without splitting.

We should point out that this obstacle avoidance algorithm will not necessarily guarantee zero collisions

between units and obstacles. A situation could arise such that a given unit receives conflicting steering instructions that might force it into an obstaclefor example, if a unit happens to get too close to a neighbor on one side while at the same time trying to avoid an obstacle on the other side. Depending on the relative distances from the neighbor and the obstacle, one steering force might dominate the other, causing a collision. Judicious tuning, again, can help mitigate this problem, but in practice you still might have to implement some sort of collision detection and response mechanism to properly handle these potential collisions.

Previous Next
Top