5.3 Artificial Intelligence: Agents, Architecture, and Techniques

In This Chapter

Overview

In many video games, the quality of the experience depends on whether the game presents a good challenge to the player. One way to present a good challenge is to offer computer opponents, or sometimes even allies, that are capable of intelligently playing the game. In most cases, this is not a trivial problem to solve, but fortunately, there is an entire field of study that can help us out—*artificial intelligence* (or AI for short).

AI describes the intelligence embodied in any manufactured device. If we design a character or opponent in a video game that acts on its own, it is generally accredited with possessing AI.

Human-level AI is the stuff of dreams and science fiction. How do you take the accumulated common sense and expertise of a human and distill it into a computer?

Unfortunately, this problem is currently unsolved and it will likely be decades before we get close to understanding what it truly entails. Since general human-level intelligence is currently impossible to re-create, researchers chip away from dozens of different angles by solving much simpler problems. By sufficiently narrowing down the domain of an AI problem, it becomes possible to create behavior that is reasonable and believable, especially in the realm of video games.

This chapter first discusses the unique properties of game AI, and how it differs from other AI fields. With believable characters being the centerpiece of most game AI, the next section introduces the concept of a game agent. Game agents perceive the world, react in intelligent ways, and potentially adapt to the player. As the most widely used architecture for game AI, various flavors of finite-state machines are then examined and compared. The chapter finishes with a survey of the most common and promising techniques in game AI today. Intelligent movement for game agents is covered in-depth in Chapter 5.4, "Artificial Intelligence: Pathfinding."

AI for Games

Video game AI is very distinct from most other AI applications, such as military defense, robotics, or data mining. The core distinction is in terms of goals. The goal of an AI programmer is to create both entertaining and challenging opponents while shipping the product on time. These goals have the following five implications:

1. The AI must be intelligent, yet purposely flawed.
   - Opponents must present a challenge.
   - Opponents must keep the game entertaining and fun.
   - Opponents must lose to the player in a challenging and fun manner.
2. The AI must have no unintended weaknesses.
   - There must be no "golden paths" to defeating the AI every time in the same way.
   - The AI must not fail miserably or look dumb.
3. The AI must perform within the CPU and memory constraints of the game.
   - Most games are real time and must have their AIs react in real time.
   - Game AI seldom receives more than 10 to 20 percent of the frame time.
4. The AI must be configurable by game designers/players.
   - Designers must be able to adjust the difficulty level and adjust the AI.
   - If the game is extensible, players can tweak or customize the AI.
5. The AI must not stop the game from shipping.
   - The AI techniques employed must not put the game at risk.
   - Experimental techniques must be proved early in the development cycle during preproduction.
   - If the AI is given latitude to evolve or change, it must be testable to guarantee that it doesn't deteriorate when released to millions of consumers.

These requirements color a game developer's perception of the field of AI. An important distinction is that game AI doesn't need to solve a problem perfectly, only

to the satisfaction of the player. For example, in pathfinding, the AI might need to calculate a path across a crowded room. Search algorithms exist to find the absolute shortest or cheapest path, but perfection is generally not a requirement for games. By relaxing the standards for many problems, shortcuts can be taken that make the problem tractable in real time or result in large computational savings.

Another consequence of game-specific AI is that the AI has access to perfect knowledge. For example, a given opponent doesn't need to sense the world the way a physical robot would need to. The game world is wholly inside the computer and the AI has the luxury of performing its analysis on these completely accurate representations. Much of robotics research concentrates on the problems of vision recognition and mechanical movement, both of which are rightfully ignored in games.

When designing game AI, considerable thought must be put into making the AI configurable by the game designers. Rather than making a perfect autonomous character or adversarial opponent, the goal is to make a highly customizable AI that can be adjusted for difficulty and individual attributes such as aggressiveness or accuracy. By creating a slightly more general AI that can be adjusted, the game can be balanced and tuned by game design experts to ensure that the game is enjoyable and fun.

Finally, an important consideration is that the product must ship on time. Experimental AI techniques are exciting and intriguing, but they have the potential to put the project unnecessarily at risk. Therefore, new AI techniques must be proven early in the development cycle. The promise that it will all come together three months before shipping is simply not acceptable.

Specialization

The last decade has seen a dramatic specialization of disciplines within game development. One of the more notable positions to fall out of this specialization is the role of the artificial intelligence developer. Once considered a side duty of general game programmers, AI has become complicated enough to warrant a deep understanding of the dozens of current and potential techniques. Even more interestingly, the skills of an AI programmer must vary dramatically between game genres. While strategy games require careful battlefield analysis and strategic planning, first-person shooter games require one-on-one tactical analysis and intelligent movement at the level of individual footsteps. There is no one-size-fits-all solution to game AI, which reinforces the tremendous specialization that takes place within this discipline.

Real-time strategy (RTS) games are perhaps the most demanding for an AI programmer, with current AAA titles typically requiring as many as three full-time AI developers. However, other titles like racing games, street fighting, or puzzle games might only need a part-time programmer for AI. Additionally, many companies are scripting more and more of their AI, which tends to push some of the AI work toward level designers.

Game Agents

With a firm grasp on the goals and purpose of game AI, let's now turn our attention to the *game agent*. In most games, the purpose of AI is to create an intelligent agent, sometimes referred to as a *nonplayer character* (NPC). This agent acts as an opponent, an ally, or as a neutral entity in the game world. Since the majority of game AI focuses around the agent, it is very helpful to study game AI from this perspective.

An agent has three key steps through which it continually loops. The steps are commonly known as the *sense-think-act* cycle. In addition to these three steps, there is an optional learning or remembering step that may also take place at the end of this loop. In practice, most game agents do not take this extra step, but this is slowly changing because of the added challenge and replayability that is leveraged as a result.

Sensing

The game agent must have information about the current state of the world to make good decisions and to act on those decisions. Since the game world is represented entirely inside the program, perfect information about the state of the world is always available. This means that there is no uncertainty about the world. The world offers accurate information to the game agent about the existence, location, and state of every opponent, barrier, or object. Unfortunately, while all of this rich information exists, it may be expensive or difficult to tease out useful and pertinent information.

At any time, the game agent can query the game world representation to locate the player or other enemies, but most players would consider this cheating. Therefore, it is necessary to endow the game agent with certain limitations in terms of what it can sense. For example, it might seem obvious, but game agents should typically not be able to see through walls.

Game agents are usually given human limitations. They are restricted to knowing only about events or entities they have seen, heard, or perhaps were told about by other agents. Therefore, it is necessary to create models for how an agent should be able to see into the world, hear the world, and communicate with other agents.

Vision

When modeling agent vision, it is important that the game engine provide fast methods for determining the visibility of objects. While game AI typically isn't very CPU intensive, visibility testing can be enormously expensive. Therefore, it is often limited to particular agents and performed only on a periodic basis.

In 3D games, vision usually starts with obtaining a list of pertinent game objects. For example, the agent might ask for a list of all enemies. Since agents are not concerned with most game objects that populate a world, it would be wasteful to consider every object in the game database. Once this pared-down list is constructed, a vector from the game agent to each game object is calculated. This `toObject` vector is then processed in the following ways to determine if the agent can see the game object. The order of these steps is important to minimize processing.

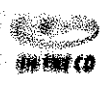
1. **Is the object within the viewing distance of the agent?** Check that the magnitude of the vector is less than or equal to the maximum viewing distance. Note that it is computationally faster to compare the distance squared, since that eliminates having to perform a square root operation.
2. **Is the object within the viewing angle of the agent?** Use a simple dot product between the `toObject` vector and the agent's forward vector to determine if the game object is within the agent's viewing angle. For example, if the dot product of the two normalized vectors is greater than or equal to 0.5, the object is located within an agent's 120-degree viewing angle.
3. **Is the object unobscured by the environment?** The ray defined by the agent's position and the `toObject` vector must be tested against the environment. This test is expensive, so it is purposely performed after all other tests.

The preceding three steps are a reasonable approximation of human vision. However, the unobstructed test is rather coarse and will not detect if just a portion of an object is visible. This can be improved by testing if the extents of the agent's bounding volume can be seen. However, this added accuracy comes at a high cost since multiple ray casting tests will need to be performed.

Depending on the game, it might be advantageous to model vision that is more sensitive to movement. As most people have experienced, it's easier to see a moving object than a perfectly still object. Of course, this effect is related to how distant the object is, since close stationary objects are easier to see than distant ones. Movement sensitive vision can be modeled by ignoring stationary objects that are beyond a particular threshold distance, or by varying the recognition reaction time of stationary objects based on their distance.

Beyond visually sensing the simple existence of objects, many more aspects of the environment might be of interest. For example, it may be important to recognize hiding spots or high-risk areas that should be avoided. This advanced recognition about the topology of the world is critical for particular games such as first-person shooters. The existence of these interesting spots can be flagged by hand, or algorithms can be devised to discover them from the world representation [Lidén02, Tozour03b]. Once these areas of interest are marked, an agent should be able to sense them like any other object.

Truly understanding the topology of the world is so important for some games that Criterion, a middleware company, advertises a dynamic solution as one of their core technologies in their *RenderWare A.I.* product. Included on the companion CD-ROM are four *RenderWare A.I.* demos that illustrate their 3D Topology Dynamic Analyzer technology. These demos are quite fun to play with, so don't miss exploring this aspect of agent sensing.



Hearing

An interesting twist on agent awareness is to allow an agent to sense through hearing. For example, if the player tip-toes past a sleeping enemy, the enemy might not notice. However if the player runs past the same enemy, the enemy might hear the noise and

wake up. Similarly, if the player starts wildly firing his gun, agents that can't see the player might rush to the scene because they heard the gunfire coming from that location.

Hearing is most commonly modeled through event-driven notifications. For example, if the player performs an action that makes a noise, the game will compute where that noise might travel to and inform any agents within that range. Rather than performing elaborate sound reflection calculations against the environment, this is usually accomplished through a simple distance calculation coupled with bounding areas. If a sound emanates inside area B and can be heard up to 10 meters away, all agents inside area B and within 10 meters are notified. This eliminates any computationally expensive sound modeling. See Chapter 5.5, "Audio Programming," for more details related to sound propagation.

Communication

Many types of agents are expected to communicate with each other, so it may be important to model the transfer of sensed knowledge between agents. Take, for example, guards. If a guard saw the player in a sensitive area, the guard could run away and alert others. The other guards can then use this information to make better decisions themselves, such as deciding to hunt down the player together, starting with the player's last known location.

Similar to the mechanism of hearing, information from communication will be event-driven in the form of notifications. When an agent has useful information and comes within a certain distance of other agents, the information will be sent directly to the other agents.

Reaction Times

When sensing the environment, it is important to build in artificial reaction times. Agents should not be able to see, hear, or communicate instantaneously. For example, it would look decidedly wrong to witness a guard take off running at the same instant the alarm is sounded.

Since agents do sense the world instantaneously, simple timers can be used to simulate reaction times. Typical reaction times for seeing and hearing might be on the order of a quarter to half a second. Communication reaction times would be longer to model speaking or gesturing between agents.

Thinking

Once an agent has gathered information about the world through its senses, the information can be evaluated and a decision can be made. This thinking step is the crux of what most people consider true AI, and can be as simple or elaborate as required.

Generally, there are two main ways in which agents make decision in games. The first is for the agent to rely on precoded expert knowledge, typically handcrafted through if-then rules, with randomness introduced to make agents less predictable. The second is for the agent to use a search algorithm to find a near-optimal solution.

Expert Knowledge

Many techniques exist for encoding expert knowledge. These include finite-state machines, production systems, decision trees, and even logical inference. By far, the most popular technique is the finite-state machine, to which a subsequent section is dedicated.

Encoding expert knowledge is appealing because it is simple and comes naturally to most people. It is quick and easy to write a series of if-then statements that ask "just the right questions," in order to make a good decision. For example, consider the rule, "If you see an enemy that is weaker than you, attack the enemy; otherwise, run away and get backup support." This simple rule embodies a great deal of common sense about knowing when to pick a fight. By accumulating knowledge in the form of if-then rules, elaborate decision-making processes can be modeled.

While expert knowledge can create a formidable AI, it is not a scalable solution. As the number of rules mounts, the system becomes brittle and bugs must be patched with more rules, which only exacerbate the inherent weakness in the system. Since expert knowledge is not a complete solution, it relies on game testers to uncover bugs so they can be repaired before the game ships. Since most agents only solve very narrow problem domains, limited expert knowledge is usually sufficient and the scalability problems generally aren't enough to cripple the technique.

Search

Search is another commonly used technique for making intelligent decisions. Search employs a search algorithm to discover a sequence of steps (a plan) that will lead to a solution or ideal state. Given possible moves and rules that govern moves, it is possible for an algorithm to explore the search space and find an optimal or near-optimal solution, if one exists.

In games, the most common use of search is in planning where the agent should move next. Game agent navigation is a tough problem that requires a great deal of programming effort in many games. As a result, a thorough discussion of pathfinding issues using search is detailed in Chapter 5.4.

Machine Learning

If imparting an agent with expert knowledge is not possible and search cannot efficiently tackle the problem, it is possible to use machine learning to discover systems for making good decisions. Potential machine learning algorithms include reinforcement learning, neural networks, and decision trees. These techniques show promise, but in practice they are almost entirely ignored by game developers. This may be due to their complexity, CPU requirements, effect on development time, the inexperience of developers, or simply the technique's inability to outperform various forms of expert systems.

Flip-Flopping

When decisions are made, there must be a mechanism to maintain that decision over some reasonable time period. If a decision is reevaluated every frame, it might flip-flop between two states and the agent will be paralyzed in a moment of indecisiveness.

Since agents should have reaction times built into their sensing and thinking, this should never happen at the scale of individual frames. However, flip-flopping might still occur every half-second and needs to be guarded against.

Acting

Until now, the game agent's sensing and thinking steps have been invisible to the player. Only in the acting step is the player able to witness the agent's intelligence. Therefore, this is a very important step in having the agent carry out its chosen decisions, and communicating its decisions to the player (if that enhances the game and the player's perception of the agent). In other words, if the agent is brilliant, and the player never realizes it, the effort making the agent intelligent was clearly wasted.

Depending on the game, there are numerous agent actions. Some common ones are to change locations, play an animation, play a sound effect, pick up an item, converse with the player, and fire a weapon. The adeptness and subtlety with which the agent carries out these actions will impact the player's opinion of the agent's intelligence. This places an enormous burden on the variety and aesthetic quality of the animations, sound effects, and dialogue created for the agent. In a very real sense, the agent can only express its intelligence in terms of the vocabulary afforded by these art assets.

In the early days of games, agents had very few animations with which to contend. Once 3D games emerged, the agent's repertoire expanded from several dozen animations to hundreds and thousands. This complexity resulted in a need to cope in a scalable manner with animation selection. Best practices in this area have moved the animation selection problem out of the code and directly into the hands of artists and game designers through data-driven design [Hargrove03a, Hargrove03b, Orkin02b].

As previously mentioned, the player is oblivious to any work the agent does during the sensing-thinking steps unless it is revealed during the acting step. Therefore, it is important to convey the hidden work to the player if it enhances gameplay. For example, if the agent has concluded that it's going to inevitably die in the near future, there might be nothing the agent can do to avoid this outcome. However, if the agent just sits there and dies, the agent will look dumb. A more entertaining outcome would be for the agent to use that information to either cower or shout "Oh no!" as it is about to die. This way, the players don't see a dumb agent getting killed—they instead see a smart agent who comprehends the situation. So, even though the outcome is the same, the agent and the game are greatly enhanced by exposing the intelligence.

Learning and Remembering

Learning and remembering together form an optional fourth step to the sense-think-act cycle. Without it, the agent will never get better, will never adapt to a particular player, and will never benefit from past events or information it witnessed or was told.

Interestingly, learning and remembering aren't necessarily important in many games, simply because agents might not live long enough to benefit from anything they might have learned. However, in games in which the agent is persistent for

longer than 30 seconds, a significant advantage can exist when learning and remembering is incorporated.

For game agents, learning is the process of remembering specific outcomes and using them to generalize and predict future outcomes. Most commonly, this can be modeled with a statistical approach. By gathering statistics about past events or outcomes, future decisions can leverage these probabilities. For example, if 80 percent of the time the player attacks from the left, the AI would be smart to expect and prepare for this likely event. Thus, the AI has adapted to the player's behavior.

Remembering can be as simple as noting the last place the player was seen to use that information during the think cycle. By keeping some bookkeeping information on observed states, objects, or players, the agent can leverage past observances at a later date. In order to not accumulate too much knowledge, these memories can fade with time depending on how important they are. Memory fading can be a way to model selective memory and forgetfulness.

It is important to note that past knowledge doesn't always need to be stored in the agent. Some types of knowledge can be stored directly in the world's data structures (this is related to smart terrain, as discussed later). For example, if agents consistently get slaughtered in a particular place, that area can be marked as more dangerous. You could almost conceptualize this as the smell of death in a particular spot. During the think cycle, path planning and tactical decisions can consider this information and prefer to avoid the area.

Making Agents Stupid

In many cases, it is actually very easy to create agents that will dominate and destroy the player. Simply make the agents faster, stronger, have more resources, or more accurate with their firing. Of course, that's not really the point of game AI. The point is generally to lose to the player in a fun and challenging way.

Dumbing down an agent can be accomplished by making it less accurate when shooting, having longer reaction times, engaging the player only one at a time, and unnecessarily making itself more vulnerable by changing positions often. These simple steps will bring agents down a notch and give the player ample time and opportunity to defeat them.

Agent Cheating

While agents can be made superior by making them faster, stronger, or omniscient, in many situations players consider this cheating. Ideally, agents don't need to cheat to make intelligent decisions or to represent a challenge, but there are situations in which it can be the best route to go. For example, in a real-time strategy game, it is often necessary to make the opponents cheat at the highest difficulty levels to provide a supreme challenge to the player. However, it is advisable to let the player know so he will not feel resentful of the AI. That way, the player is making an informed decision to play against an AI that has an unfair advantage.

The primary lesson with cheating is to be upfront with the players and never let them catch you cheating. If the players suspect that the AI is cheating, they will feel less compelled to continue playing, and it can ultimately hurt the success of the game.

Summary of Game Agents

The sense-think-act cycle of game agents is a simple conceptual framework for organizing intelligent behavior. It isn't intelligent in and of itself, but it provides a good foundation for creating intelligent and believable agent behavior. It's a guide that helps the programmer conceptualize what an agent needs to know and consider before the agent acts in the world. As we will see in subsequent sections, many more techniques need to be employed to achieve intelligent AI.

Finite-State Machines

Within game AI, it is generally recognized that *finite-state machines* (FSMs) are the most common software pattern. This kind of popularity doesn't happen by accident. Rather, FSMs are widely used because they possess some amazing qualities. They are simple to program, easy to comprehend, easy to debug, and completely general to any problem. They might not always provide the best solution, but they consistently get the job done with minimal risk to the project.

However, FSMs have a darker side as well. Many programmers look at them with distrust since they tend to be constructed ad hoc with no consistent structure. They also tend to grow uncontrollably as the development cycle churns on. This poor structure, coupled with unbounded growth, makes many FSM implementations difficult to maintain.

Yet with all of their warts, FSMs are still the most compelling way to structure most game AI implementations.

The Basic Finite-State Machine

Formally, a finite-state machine is an abstract model of computation that consists of a set of states, a starting state, an input vocabulary, and a transition function that maps inputs and current states to a next state. Computation begins with the starting state and transitions to new states as inputs are received. The FSM can perform work within a given state, known as a *Moore machine*, or on the transitions between states, known as a *Mealy machine*.

Game developers deviate from the strict FSM definition in many different ways. First, the states themselves are used to define behaviors that contain code specific to that state. For example, states can be behaviors such as wander, attack, or flee. Second, the single transition function is typically divided among the states so that each state knows exactly what will cause its transition to another state, which helps keep the relation between states and transitions easy to understand. Third, the line between Moore and Mealy machines is blurred, as work is often performed both inside of a state and during transitions. Fourth, leveraging probability and randomness is extremely

common when transitioning to a new state. For example, after being attacked, an agent might have a 10 percent chance of transitioning to the flee state. Fifth, extra state information not directly represented in the FSM, such as agent health, is often used as a deciding factor for some state transitions.

Since FSMs can elegantly capture the mental states or behaviors of an agent, they are a natural choice for defining character AI. Figure 5.3.1 demonstrates how an agent's behavioral FSM might be diagrammed using UML (Unified Modeling Language).

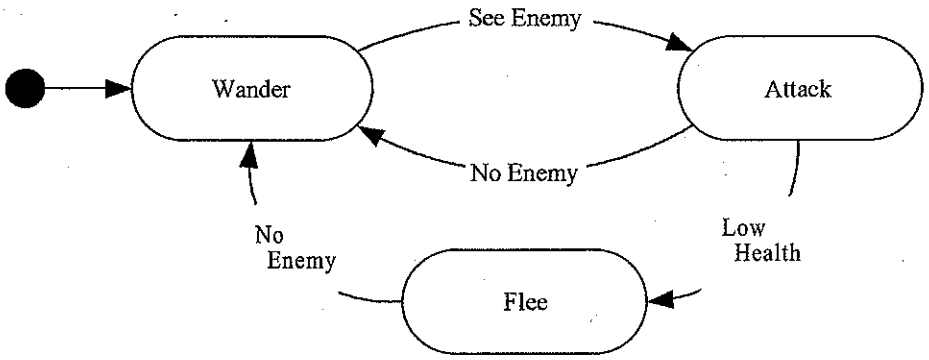


FIGURE 5.3.1 *An example of diagramming an agent's behavioral FSM in UML. The black dot points to the starting state.*

Defining an FSM

Having covered the basics, the next issue is how to define an FSM in the game. There are several different methods. The simplest and most straightforward approach is to directly code the FSM in the game's source language. Listing 5.3.1 shows an FSM defined in C/C++ using the `switch` statement construct. This is perhaps the simplest implementation of a finite-state machine.

Listing 5.3.1 An FSM coded directly in C/C++

```
void RunLogic( int * state )
{
    switch( *state )
    {
        case 0:  //Wander
            Wander();
            if( SeeEnemy() )    { *state = 1; }
            break;

        case 1:  //Attack
            Attack();
            if( LowOnHealth() ) { *state = 2; }
```

```
            if( NoEnemy() )      { *state = 0; }
            break;

        case 2:  //Flee
            Flee();
            if( NoEnemy() )      { *state = 0; }
            break;
    }
}
```

The FSM in Listing 5.3.1 consists of three states and four transitions, identical to the diagram in Figure 5.3.1. Presumably, `RunLogic` is called each frame during the game while the agent is alive. Depending on the agent's current state, a single action will be carried out on that frame. After each action executes, potential transitions to new states are checked and possibly taken. The logic is simple and easy to understand. Debugging is also quite trivial with this implementation.

Unfortunately, there are several problems with this simple `switch` statement structure:

- The code is ad hoc, in that the language doesn't enforce the structure. There is nothing preventing another programmer from adding catch-all code outside the `switch` statement that further modifies states or executes actions.
- All transitions result from polling, which can be inefficient. In practice, it is better to be able to transition to a different state when an event occurs, such as being attacked by an enemy, rather than checking every frame if an enemy has attacked.
- There is no easy way to know that a state was entered for the first time. For example, upon entering the attack state, the agent might need to unsheathe his sword. One solution is to create a "preattack" state that prepares the sword and then immediately transitions to the proper attack state, but this can lead to an explosion of states that complicates the structure.
- The FSM is defined directly in the code and can't be specified by game designers. If the FSM is data-driven outside of the game code, the possibility for more parallel work exists, which might be important for larger games.

An alternative to directly hard coding an FSM in the game's source language is to create a scripting language that encapsulates the best features of an FSM and enforces a consistent structure. Such a scripting language might resemble Listing 5.3.2.

Listing 5.3.2 A fictional FSM scripting language that abstracts and enforces a consistent structure

```
AgentFSM
{
    DeclareState( STATE_Wander )
        OnUpdate
            Execute( Wander )
```

```
                if( SeeEnemy )
                    ChangeState( STATE_Attack )
            OnEvent( AttackedByEnemy )
                ChangeState( Attack )

        DeclareState( STATE_Attack )
            OnEnter
                Execute( PrepareWeapon )
            OnUpdate
                Execute( Attack )
                if( LowOnHealth )
                    ChangeState( STATE_Flee )
                if( NoEnemy )
                    ChangeState( STATE_Wander )
            OnExit
                Execute( StoreWeapon )

        DeclareState( STATE_Flee )
            OnUpdate
                Execute( Flee )
                if( NoEnemy )
                    ChangeState( STATE_Wander )
}
```

The fictional FSM scripting language in Listing 5.3.2 demonstrates several improvements over the hard-coded version:

- The structure of the FSM is enforced by what will be accepted by the script compiler.
- Events can be handled (via the `OnEvent` convention), as well as polling.
- When a state is entered for the first time, the `OnEnter` construct can be used to execute any special initialization. Conversely, there is an `OnExit` construct to carry out any cleanup code, regardless of what triggered the transition. The `OnExit` construct makes the script more explicit and reduces redundant code.
- A scripted, data-driven FSM can be specified by game designers and artists who are not familiar with traditional programming languages.

The `Execute` and `if` constructs indicate that a particular function should be called by the specified name. This requires the name within the parentheses to be bound to an actual function name in the game code. In the case of the `if` statement, the function will return a Boolean so that the script knows whether to execute the next statement. Note that this particular scripting language lacks curly brackets and semicolons, which is simply a language design choice.

Unfortunately, it is not trivial to create an FSM scripting language, and the decision to implement one should not be taken lightly. Typically, it can require several months of engineering work to design and implement the language. A custom compiler must be written that converts the scripted FSM into bytecode that can be interpreted "on the fly" by the game engine during runtime. This presents several problems in terms of usability and debugging, since compile-time errors in the script must be reported back by the custom compiler, and debugging the scripts during runtime will

require extensive hooks and support. Unsurprisingly, it is not uncommon for a custom scripting language within a game company to be despised and hated by the people who must work with it on a daily basis. After all, it is extremely difficult and time consuming to create tools that approach the polish and robustness of commercial compilers and debuggers.

Since the difficulty is in the tools, at least one middleware company now offers FSM solutions that assist with creation (using visual diagramming) and debugging. However, since FSMs are trivial to implement directly in code and many companies already have proprietary scripting languages, it can be difficult to convince game developers that they can benefit from these outside solutions.

One possible solution to the dilemma is to develop a hybrid approach. Included on the companion CD-ROM is a "State Machine Language" that is implemented completely in C++. Through the use of several C-style macros and an FSM class, it is possible to achieve the abstraction and structure of many scriptable FSM languages. By existing entirely in the game's source language, namely C++, all of the compiling and debugging problems of scripting languages fall away. Listing 5.3.3 shows an example of the C++ State Machine Language.

Listing 5.3.3 A hybrid approach of coding an FSM directly in C++ using a supporting FSM class and C-Style macros

```
bool AgentFSM::States( StateMachineEvent event,
                       MSG_Object * msg,
                       int state, int substate )
{
    DeclareState( STATE_Wander )
        OnUpdate
            Wander();
            if( SeeEnemy() )
                ChangeState( STATE_Attack );
        OnMsg( MSG_Attacked )
            ChangeState( Attack )

    DeclareState( STATE_Attack )
        OnEnter
            PrepareWeapon();
        OnUpdate
            Attack();
            if( LowOnHealth() )
                ChangeState( STATE_Flee );
            if( NoEnemy() )
                ChangeState( STATE_Wander );
        OnExit
            StoreWeapon();

    DeclareState( STATE_Flee )
        OnUpdate
            Flee();
```

```
            if( NoEnemy )
                ChangeState( STATE_Wander );
    }
```

The C++ state machine language example in Listing 5.3.3 conceals a great deal of functionality. However, the primary point is that the structure promotes a consistent format, good readability, and straightforward debugging. It supports the `OnEnter` and `OnExit` concepts, and event-driven triggers in the form of messages that get pumped into the state machine (captured by the `OnMsg` construct). What isn't supported is defining the FSM in a way that designers or artists can author it from outside the source code. However, this is the tradeoff to avoid creating tools and instead leverage the existing compiler and debugger.

Extending the Basic FSM

So far, we have seen several ways to extend the basic FSM. These include extending states to offer *OnEnter*/*OnExit* blocks and allowing event notifications to trigger transitions. As mentioned earlier, it's common to allow randomness and probability to influence transitions, and it's also common to refer to additional state information, such as health, when making state transition decisions.

Beyond these, there are several other important ways that FSMs can be extended. First, FSMs can have a stack-based history that tracks past states. As transitions are followed, states are pushed on and popped off the history stack. This is extremely useful if the agent becomes interrupted and later needs to return to a previous state. For example, if an agent was repairing a building (repair state), but subsequently got caught up in a firefight (attack state), once the fight is over, the attack state can pop itself off the stack and the FSM would return to the repair state. Therefore, once a state completes, it can choose to resume the last behavior without having to reexamine the situation.

A related stack-based extension is to allow a state to transition to a completely new FSM, pushing it onto an FSM stack. This results in a *hierarchical* FSM that can lead to better encapsulation of behaviors and tasks, thus keeping any one FSM from becoming too large and cumbersome [Houlette01]. Hierarchical FSMs can also reduce code duplication, since common subbehaviors can be referenced by many other FSMs.

Similar to hierarchical FSMs, a single FSM can potentially have substates that exist within a given state. Depending on the situation, this can be an effective way to break down behavior without resorting to a completely new FSM for just a couple of related states.

Multiple FSMs

So far, we have considered a single agent owning a single FSM, but there is no reason why an agent couldn't own several concurrently running FSMs. One model is to have an agent run both a "brain" FSM and a "movement" FSM. Another model is known

as a *subsumption* architecture, where there are multiple levels of concurrently running FSMs [Brooks89]. The lowest level FSM takes care of rudimentary decisions such as obstacle avoidance, while the higher level FSMs focus on goal seeking and goal determination. A subsumption architecture remains robust because the lower levels must be satisfied before they allow the higher levels to influence the behavior.

Debugging FSMs

One of the chief benefits of working with FSMs is the ease with which they can be debugged. However, when there are dozens or hundreds of agents milling about, complex interactions can still be very difficult to debug. Therefore, it is prudent to build debugging facilities directly into your FSM architecture. At the very least, you should be able to log the states of each FSM over time. By capturing this data, particular logs can be examined after a bug has occurred and good clues will be available to help track down the cause.

Another way to facilitate debugging is to have agents display their current state above their heads. By being able to see the current state, you can quickly identify what each agent is "thinking," thus making it easy to visually spot errors. It might also help to display the last state as well so that it's clearer how the agent transitioned into the current state.

Summary of FSMs

Finite-state machines are general, simple, easy to understand, and easy to debug. They are much more useful than the formal definition might suggest, and can serve as the basis for almost any AI agent implementation. However, FSMs aren't capable of pathfinding, reasoning, or learning, so other techniques will most certainly have to be employed. Yet, it would be a mistake to not initially investigate whether FSMs could solve a portion of your AI needs.

Common AI Techniques

The following survey of common AI techniques is designed to provide an executive summary of the many tools that an AI programmer can wield. Since game AI is approached from so many diverse directions, a whirlwind tour of techniques is a good way to familiarize oneself with the diverse landscape of available solutions. The next section similarly provides a survey of promising AI techniques.

A* Pathfinding

A pathfinding* (pronounced A-star) is an algorithm for finding the cheapest path through an environment. Specifically, it is a directed search algorithm that exploits knowledge about the destination to intelligently guide the search. By doing so, the processing required to find a solution is minimized. Compared to other search algorithms, A* is the fastest at finding the absolute cheapest path. Note that if all movement has the same traversal cost, the cheapest path is also the shortest path.

Game Example

The environment must first be represented by a data structure that defines where movement is allowed [Tozour03a]. A path is requested by defining a start position and a goal position within that search space. When A* is run, it returns a list of points, like a trail of breadcrumbs, that defines the path. A character or vehicle can then use the points as guidelines to find its way to the goal.

A* can be optimized for speed [Cain02, Higgins02b, Rabin00a], for aesthetics [Rabin00b], and for general applicability to other tasks [Higgins02a]. Variations like D* attempt to make path replanning cheaper [Stentz94]. A* pathfinding is described in detail in Chapter 5.4.

Command Hierarchy

A *command hierarchy* is a strategy for dealing with AI decisions at different levels, from the general down to the foot soldier. Modeled after military hierarchies, the general directs the high-level strategy on the battlefield, while the foot soldier concentrates on individual combat. The levels in between deal with cooperation between various platoons and squads. The benefit of a command hierarchy is that decisions are separated at each level, thus making each decision more straightforward and abstracted from other levels [Kent03, Reynolds02].

Game Example

A command hierarchy is often used in real-time strategy or turn-based strategy games where there are typically three easily identifiable levels of decisions: overall strategy, squad tactics, and individual combat. A command hierarchy is also useful when a large number of agents must have an overall coherency.

Dead Reckoning

Dead reckoning is a method for predicting a player's future position based on that player's current position, velocity, and acceleration. This simple form of prediction works well since the movement of most objects resembles a straight line over short periods of time. More advanced forms of dead reckoning can also provide guidance for how far an object *could have moved* since it was last seen.

Game Example

In a first-person shooter (FPS) game, an effective method of controlling the difficulty level is to vary how accurate the computer is at "leading the target" when shooting projectiles. Since most weapons don't travel instantaneously, the computer must predict the future position of targets and aim the weapon at these predicted positions. Similarly, in a sports game, the computer player must anticipate the future positions of other players to effectively pass the ball or intercept a player [Laramée03, Stein02].

Emergent Behavior

Emergent behavior is behavior that wasn't explicitly programmed, but instead emerges from the interaction of several simpler behaviors. Many life forms use rather basic

behavior that, when viewed as a whole, can be perceived as being much more sophisticated. In games, emergent behavior generally manifests itself as low-level simple rules that interact to create interesting and complex behaviors. Some examples of rules are seek food, seek similar creatures, avoid walls, and move toward the light. While any one rule isn't interesting by itself, unanticipated individual or group behavior can emerge from the interaction of these rules.

Game Example

Flocking is a classical example of emergent behavior in games, which results in realistic movement of flocks of birds or schools of fish [Reynolds87, Reynolds01]. Other emergent behaviors involving racecar applications and general-purpose creatures have been well documented [Darby03, Porcino03].

Flocking

Flocking is a technique for moving groups of creatures in a natural and organic manner. It works well at simulating flocks of birds and schools of fish. Each creature follows three simple movement rules that result in complex group behavior. It is said that this group behavior *emerges* from the individual rules (emergent behavior). Flocking is a form of artificial life that was popularized by Craig Reynolds' work [Reynolds87, Reynolds01].

The three classic flocking rules devised by Reynolds are:

- **Separation:** Steer to avoid crowding local flock mates
- **Alignment:** Steer toward the average heading of local flock mates
- **Cohesion:** Steer toward the average position of local flock mates

Game Example

Games typically use flocking to control background creatures such as birds or fish. Since the path of any one creature is highly arbitrary, flocking is typically used for simple creatures that tend to wander with no particular destination. The result is that flocking techniques, as embodied by the three core rules, rarely get used for key enemies or creatures. However, the flocking rules have inspired several other movement algorithms, such as formations and swarming [Scutt02].

Formations

Formations are a group movement technique used to mimic military formations. Although it shares similarities to flocking, it is quite distinct in that each unit is guided toward a specific goal location and heading, based on its position in the formation [Dawson02].

Game Example

Formations can be used to organize the movement of ground troops, vehicles, or aircraft. Often, the formations must split or distort themselves to facilitate movement

through tight areas. The game *Age of Empires 2: Age of Kings* pioneered several key techniques for formations [Pottinger99a, Pottinger99b].

Influence Mapping

An *influence map* is a method for viewing the distribution of power within a game world. Typically, it is a two-dimensional (2D) grid that is superimposed onto the landscape. Within each grid cell, units that lie in the cell are summed into a single number representing the combined influence of the units. It is assumed that each unit also has an influence into neighboring cells that falls off either linearly or exponentially with distance. The result is a 2D grid of numbers that gives insight into the location and influence of differing forces [Woodcock02].

Game Example

Influence maps can be used offensively to plan attacks; for example, by finding neutral routes to flank the enemy. They can also be used defensively to identify areas or positions that need to be strengthened. If one faction is represented by positive values and the other faction is represented by negative values within the same influence map, any grid cells near zero are either unowned territory or the "front" of the battle (where the influence of each side cancels each other out) [Tozour01].

There are also nonviolent uses for influence maps. For example, the *Sim City* series offers real-time maps that show the influence of police and fire departments placed around the city. The player can then use this information to place future buildings to fill in the gaps in coverage. The game also uses the same information to help simulate the world.

Level-of-Detail AI

Level-of-detail (LOD) is a common optimization technique in 3D graphics where polygonal detail is only used when it can be noticed and appreciated by the human viewer. Close-up models use large numbers of polygons, while faraway models use fewer polygons. This results in faster graphics processing since fewer polygons are rendered, yet there is no noticeable degradation in visual quality. The same concept can be applied to AI, where computations are performed only if the player will notice or appreciate the result [Brockington02a].

Game Example

One approach is to vary an agent's update frequency based on its proximity to the player. Another technique is to calculate paths only for agents that the player can see; otherwise, use straight-line path approximations and estimate off-screen movement. This technique becomes important when there are more than several dozen agents in a game and collectively they use too much processing power. This often occurs with RPG, RTS, strategy, and simulation games.

Manager Task Assignment

When a group of agents tries to independently choose tasks to accomplish, such as selecting a target in battle, the performance of the group can be rather dismal. Interestingly, the problem can be turned around so that instead of the individuals choosing tasks, a manager has a list of required tasks and assigns agents based on who is best suited for the job [Rabin98]. Note that this is very different from having the manager run through the list of individuals and assign tasks. Task assignment considers the tasks themselves first and uses them as the basis for prioritizing. This avoids duplication of tasks, and the best candidate for a task is always chosen. This type of tactical planning is more deliberate than the emergent coordination that can be achieved with a blackboard architecture. However, the resulting plan might not be as optimal as performing an exhaustive planning search [Orkin03a].

Game Example

In a baseball game with no runners on base, it might be determined that the first priority is to field the ball, the second priority is to cover first base, the third priority is to back up the person fielding the ball, and the fourth priority is to cover second base. The manager can organize who covers each priority by examining the best person for the job for a given situation. On a soft hit between first and second base, the manager might assign the first baseman to field the ball, the pitcher to cover first base, the second baseman to back up the first baseman fielding the ball, and the shortstop to cover second base, which is the correct play. Without a manager to organize the task assignment, it can be significantly harder to get coherent cooperation out of the players using other methods.

Obstacle Avoidance

A* pathfinding algorithms are good at getting a character from point to point through static terrain. However, often the character must avoid players, other characters, and vehicles that are moving rapidly through the environment. Characters must not get stuck on each other at choke points, and they must maintain enough spacing to maneuver when traveling in groups. *Obstacle avoidance* attempts to prevent these problems using trajectory prediction and layered steering behaviors [Reynolds99].

Game Example

In an FPS game, a group of four skeletons wants to attack the player, but must first cross a narrow bridge over a river. Each skeleton has received a route to the player through the navigation system. The skeleton closest to the bridge has a clear path across. The second skeleton predicts a collision with the first, but sees space to the right, which is still within the boundaries of the path across the bridge. The last two skeletons predict collisions with the first two, so they slow their rate of travel to correctly queue up behind the first two.

Scripting

Scripting is the technique of specifying a game's data or logic outside of the game's source language. Often, the scripting language is designed from scratch, but there is a

growing movement toward using Python and Lua as alternatives. There is a complete spectrum for how far you can take the scripting concept.

Scripting influence spectrum:

- **Level 0:** Hard code everything in the source language (C/C++)
- **Level 1:** Data in files specify stats and locations of characters/objects.
- **Level 2:** Scripted cut-scene sequences (noninteractive)
- **Level 3:** Lightweight logic specified by tools or scripts, as in a trigger system
- **Level 4:** Heavy logic in scripts that rely on core functions written in C/C++
- **Level 5:** Everything coded in scripts—full alternative language to C/C++

Commercial games have been developed at all levels of this spectrum, with the oldest video games at level 0 and games such as the *Jak and Daxter* series at level 5 (with their GOAL language based on LISP). However, the middle levels are where most games have settled, since the two extremes represent increased risk, time commitment, and cost.

Game Example

Programmers must first integrate a scripting language into the game and determine the extent of its influence. The users of the scripting language will typically be artists and level designers. The written script will typically be compiled into byte code before actual gameplay and interpreted "on the fly" during gameplay.

The following are the advantages and disadvantages of scripting [Tozour02a].

Advantages of scripting:

- Game logic can be changed in scripts and tested without recompiling the code.
- Designers can be empowered without consuming programmer resources.
- Scripts can be exposed to the players to tinker with and expand (extensible AI).

Disadvantages of scripting:

- More difficult to debug.
- Nonprogrammers may be required to program.
- Time commitment and cost to create and support scripting language and complementary debugging tools.

State Machine

A *state machine* or *finite-state machine* (FSM) is a widely used software design pattern that has become a cornerstone of game AI. An FSM is defined by a set of states and transitions, with only one state active at any one time.

Game Example

In common practice, each state represents a behavior, such as `PatrolRoute`, within which an agent will perform a specific task. The state either polls or listens for events that will cause it to transition into other states. For example, a `PatrolRoute` state

might check periodically if it sees an enemy. When this event happens, it transitions into the `AttackEnemy` state.

Stack-Based State Machine

A *stack-based state machine* is a technique and design pattern that often appears in game architectures. Sometimes referred to as *push-down automata*, the stack-based state machine can remember past actions by storing them on a stack. In a traditional state machine, past states are not remembered, since control flows from state to state with no recorded history. However, it can be useful in game AI to be able to transition back to a previous state, regardless of which state it was. This stack concept can be used to capture previous states, or even entire state machines [Tozour03c, Yiskis03a].

Game Example

In a game, this technique is important when a character is performing an action, becomes interrupted for a moment, and then wants to resume the original action. For example, in a real-time strategy game, a unit might be repairing a building when it gets attacked. The unit will transition into an attack behavior and might destroy the enemy. In this case, the conflict is over and the unit should resume its previous activity. If past behaviors are stored on a stack, the current attack behavior is simply popped from the stack and the unit will resume the repair behavior.

Subsumption Architecture

A *subsumption architecture* cleanly separates the behavior of a single character into concurrently running layers of FSMs. The lower layers take care of rudimentary behavior such as obstacle avoidance, and the higher layers take care of more elaborate behaviors such as goal determination and goal seeking. Because the lower layers have priority, the system remains robust and ensures that lower layer requirements are met before allowing higher level behaviors to influence them. The subsumption architecture was popularized by the work of Rodney Brooks [Brooks89].

Game Example

Subsumption architectures have been used in many games, including the *Oddworld* series of games, *Jedi Knight: Dark Forces 2*, and *Halo: Combat Evolved*. The architecture is ideally suited for character-based games where movement and sensing must coexist with decisions and high-level goals [Yiskis03b].

Terrain Analysis

Terrain analysis is the broad term given to analyzing the terrain of a game world to identify strategic locations.

Game Example

There are many strategic locations in a game that might be identified through terrain analysis, such as resources, choke points, or ambush points [Higgins02c]. These loca-

tions can then be used by the strategic-level AI to help plan maneuvers and attacks. Other uses for terrain analysis in a real-time strategy game include knowing where to build walls [Grimani03] or where to place the starting factions. In an FPS game, terrain analysis can assist the AI in discovering sniper points, cover points, or where to throw grenades from [Lidén02, Reed03, Tozour03b, van der Sterren00]. Terrain analysis can be viewed as the alternative approach to "hard coding" regions of interest in a level.

Trigger System

A *trigger system* is a highly specialized scripting system that allows simple if/then rules to be encapsulated within game objects or the world itself. It is a useful tool for level designers since the concept is extremely simple and robust. Often, it is exposed through a level design tool or a scripting language [Orkin02a, Rabin02].

Game Example

A designer might put a floor trigger in the middle of a room. When the player steps on the floor trigger (the condition), the designer might specify that a scary sound effect is played and a dozen snakes drop from the ceiling (the response). In this way, a trigger system is a simple way to specify scripted events without designing a complex scripting language. As an example, the level editor for *StarCraft* allowed users to define their own missions using a Windows-based trigger-system tool.

Promising AI Techniques

The previous section described many common techniques that are typically employed in current games. This next section examines techniques that show potential for the future. For some reason or another, each technique has found limited use or acceptance within the games industry. Some techniques are rather complicated or difficult to understand, some are not well known, and some solve niche problems and might never gain widespread use. Regardless, it is important to be aware of these promising techniques for games.

Bayesian Networks

Bayesian networks allow an AI to perform complex humanlike reasoning when faced with uncertainty. In a Bayesian network, variables relating to particular states, features, or events in the game world are represented as nodes in a graph, and the causal relationships between them as arcs. Probabilistic inference can then be performed on the graph to infer the values of unknown variables, or conduct other forms of reasoning [IDIS99].

Game Example

One particularly important application for Bayesian networks in games lies in modeling what an AI should believe about the human player based on the information it has

available. For example, in a real-time strategy game, the AI can attempt to infer the existence or nonexistence of certain player-built units, like fighter planes or warships, based on what it has seen produced by the player so far. This keeps the AI from cheating and actually allows the human to deceive the AI by presenting misleading information, offering new gameplay possibilities and strategies for the player [Tozour02b].

Blackboard Architecture

A *blackboard architecture* is designed to solve a single complex problem by posting it on a shared communication space, called the *blackboard*. Expert objects then look at the blackboard and propose solutions. The solutions are given a relevance score, and the highest scoring solution (or partial solution) is applied. This continues until the problem is "solved."

Game Example

In games, the blackboard architecture can be expanded to facilitate cooperation among multiple agents. A problem, such as attacking a castle, can be posted, and individual units can propose their role in the attack. The volunteers are then scored, and the most appropriate ones are selected [Isla02].

Alternatively, the blackboard concept can be relaxed by using it strictly as a shared communication space, letting the individual agents regulate any cooperation. In this scheme, agents post their current activities and other agents can consult the blackboard to avoid beginning redundant work. For example, if an alarm is sounded in a building and enemies start rushing the player, it might be desirable for them to approach from different doors. Each enemy can post the door through which it will eventually enter, thus encouraging other enemies to choose alternate routes [Orkin03b].

Decision Tree Learning

A *decision tree* is a way of relating a series of inputs (usually measurements from the game world) to an output (usually representing something you want to predict) using a series of rules arranged in a tree structure. For example, inputs representing the health and ammunition of a bot could be used to predict the probability of the bot surviving an engagement with the player. At the root node, the decision tree might test to see whether the bot's health is low, indicating that the bot will not survive if that is the case. If the bot's health is not low, the decision tree might then test to see how much ammunition the bot has, perhaps indicating that the bot will not survive if its ammunition is low, and will survive otherwise. Decision trees are particularly important for applications such as in-game learning, because (in contrast to competing technologies like neural networks) extremely efficient algorithms exist for creating decision trees in near real time [Fu03].

Game Example

The best-known game-specific use of decision trees is in the game *Black & White* where they were used to allow the creature to learn and form "opinions" [Evans02]. In

Black & White, a creature will learn what objects in the world are likely to satisfy his desire to eat, based on feedback it gets from the player or world. For example, the player can provide positive or negative feedback by stroking or slapping the creature. A decision tree is then created that reflects what the creature has learned from its experiences. The creature can then use the decision tree to decide whether certain objects can be used to satisfy its hunger. While *Black & White* has demonstrated the power of decision trees to learn within games, they still remain largely untapped by the rest of the games industry.

Filtered Randomness

Filtered randomness attempts to ensure that random decisions or events in a game appear random to the players. This can be achieved by filtering the results of a random number generator such that non-random-looking sequences are eliminated, yet statistical randomness is maintained. For example, if a coin is flipped eight times in a row and turns up heads every time, a person might wonder if there was something wrong with the coin. The odds of such an event occurring are only 0.4 percent, but in a sequence of 100 flips it is extremely likely that either eight heads or eight tails in a row will be observed. When designing a game for entertainment purposes, it is desirable for random elements to always appear random to the players.

The technique involves keeping a short history of past results for each random decision that should be filtered. When a new decision is requested, a random result is generated and compared to the history. If an undesirable pattern or sequence is detected, the result is discarded and a new random result is generated. The process is repeated until a suitable result is accepted. Surprisingly, reasonable statistical randomness is maintained despite the deliberate filtering [Rabin03].

Game Example

Simple randomness filtering is actually very common in games. For example, if a character plays a random idle animation, often the game will ensure that the same idle animation won't be played twice in a row. However, filtering can be devised to remove all peculiar sequences. For example, if an enemy can randomly spawn from five different points, it would be extremely undesirable for the enemy to spawn from the same point five times in a row. It would also be undesirable for the enemy to randomly spawn in the counting sequence 12345 or favor one or two particular spawn points in the short term, like 12112121. Although these sequences can arise by chance, they are neither intended nor anticipated when the programmer wrote the code to randomly choose a spawn point. By detecting and filtering undesirable patterns or sequences with simple rules, a particular random decision can be guaranteed to always appear fair and balanced in the short term while still maintaining good statistical randomness.

Fuzzy Logic

Fuzzy logic is an extension of classical logic that is based on the idea of a fuzzy set. In classical crisp set theory, an object either does or does not belong to a set. For example,

a creature is a member of the set of hungry creatures or is not a member of that set (it is either hungry or not hungry). With fuzzy set theory, an object can have continuously varying degrees of membership in fuzzy sets. For example, a creature could be hungry with degree of membership 0.1, representing slightly hungry, or 0.9, representing very hungry, or any value in between [McCuskey00].

Genetic Algorithms

A *genetic algorithm* (GA) is a technique for search and optimization that is based on evolutionary principles. GAs represent a point within a search space using a chromosome that is based on a handcrafted genetic code. Each chromosome consists of a string of genes that together encode its location in the search space. For example, the parameters of an AI agent can be the genes, and a particular combination of parameters a chromosome. All combinations of parameters will represent the search space.

By maintaining a population of chromosomes, which are continually mated and mutated, a GA is able to explore search spaces by testing different combinations of genes that seem to work well. A GA is usually left to evolve until it discovers a chromosome that represents a point in the search space that is good enough. GAs outperform many other techniques in search spaces that contain many optima, and are controlled by only a small number of parameters, which must be set by trial and error.

Game Example

Genetic algorithms are very good at finding a solution in complex or poorly understood search spaces. For example, your game might have a series of settings for the AI, but because of interactions between the settings, it is unclear what the best combination would be. In this case, a GA can be used to explore the search space consisting of all combinations of settings to come up with a near-optimal combination [Sweetser03a]. This is typically done offline since the optimization process can be slow and because a near-optimal solution is not guaranteed, meaning that the results might not improve gameplay.

N-Gram Statistical Prediction

An *n-gram* is a statistical technique that can predict the next value in a sequence. For example, in the sequence 18181810181, the next value will probably be an 8. When a prediction is required, the sequence is searched backward for all sequences matching the most recent $n-1$ values, where n is usually 2 or 3 (a *bigram* or *trigram*). Since the sequence might contain many repetitions of the n-gram, the value that most commonly follows is the one that is predicted. If the sequence is built up over time, by representing the history of a variable (such as the last player's move), it is possible to make a prediction of a future event. The accuracy of a prediction made by an n-gram tends to improve as the amount of historical data increases.

Game Example

For example, in a street fighting game, the player's actions (various punches and kicks) can be accumulated into a move history. Using the trigram model, the last two player

moves are noted; for example, a Low Kick followed by a Low Punch. The move history is then searched for all examples where the player performed those two moves in sequence. For each example found, the move following the Low Punch and Low Kick is tallied. The statistics gathered might resemble Table 5.3.1.

Table 5.3.1 Statistics Gathered from Past Player Moves

Player Sequence	Occurrences	Frequency
Low Kick, Low Punch, Uppercut	10 times	50%
Low Kick, Low Punch, Low Punch	7 times	35%
Low Kick, Low Punch, Sideswipe	3 times	15%

The information in Table 5.3.1 can be used in two different ways. The first is to predict that the player's next move will be the one with the highest probability (the Uppercut with 50 percent likelihood based on past moves). The other is to use the probabilities as the chance that each will be predicted. Using this second technique, it is still possible to predict a Low Punch or Sideswipe as the next move, but it is less likely to make that prediction.

The statistics in Table 5.3.1 can be quickly calculated "on the fly" when a prediction is requested. A moving window into the past can be used so as not to consider moves that occurred too long ago [Laramée02b].

Neural Networks

Neural networks are complex nonlinear functions that relate one or more input variables to an output variable. They are called neural networks because internally they consist of a series of identical nonlinear processing elements (analogous to neurons) connected together in a network by weights (analogous to synapses). The form of the function that a particular neural network represents is controlled by values associated with the network's weights. Neural networks can be trained to produce a particular function by showing them examples of inputs and the outputs they should produce in response. This training process consists of optimizing the network's weight values, and several standard training algorithms are available for this purpose. Training most types of neural networks is computationally intensive, however, making neural networks generally unsuitable for in-game learning. Despite this, neural networks are extremely powerful and have found some applications in the games industry.

Game Example

In games, neural networks have been used for steering racecars in *Colin McRae Rally 2.0* and for control and learning in the *Creatures* series. Unfortunately, there are still relatively few applications of neural networks in games, as very few game developers are actively experimenting with them.

Perceptrons

A *perceptron network* is a single-layer neural network, which is simpler and easier to work with than a multilayer neural network. A perceptron network is composed of multiple *perceptrons*, each of which can either have a "yes" or "no" output. In other words, each perceptron either gets stimulated enough to trigger or it does not. Since a perceptron can classify things as "yes" or "no," it can be used to learn simple Boolean decisions such as attack or don't attack. They take up very little memory and are easier to train than a multilayer neural network or a decision tree. It is important to note, however, that perceptrons and perceptron networks have some limitations and can only learn simple (linearly separable) functions.

Game Example

In the game *Black & White*, every desire of a creature was represented by a different perceptron [Evans02]. For example, a single perceptron was used to represent the desire to eat (or hunger). Using three inputs (low energy, tasty food, and unhappiness), a perceptron would determine whether a creature was hungry. If the creature ate and received either positive or negative reinforcement, the weight associated with the perceptron would be adjusted, thus facilitating learning [Evans02].

Planning

The aim of *planning* is to find a series of actions for the AI that can change the current configuration of the game world into a target configuration. By specifying preconditions under which certain actions can be taken by the AI, and what the effects of those actions are likely to be, planning becomes a problem of searching for a sequence of actions that produces the required changes in the game world. Effective planning relies on choosing a good planning algorithm to search for the best sequence of actions, choosing an appropriate representation for the game world, and choosing an appropriate set of actions that the AI will be allowed to perform and specifying their effects.

Game Example

When the domain of a planning problem is sufficiently simple, formulating small plans is a reasonable and tractable problem that can be performed in real time. For example, in a game, a guard might run out of ammo during a gunfight with the player. The AI can then try to formulate a plan that will result in the player's demise given the guard's current situation. A planning module might come back with the solution of running to the light switch, turning it off to provide safety, running into the next room to gather ammo, and waiting in an ambush position [Orkin03a]. As game environments become more interactive and rich with possibilities, planning systems can help agents cope with the complexity by formulating reasonable and workable plans.

Player Modeling

Player modeling is the technique of building a profile of a player's behavior, with the intent of adapting the game. During play, the player's profile is continuously refined

by accumulating statistics related to the player's behavior. As the profile emerges, the game can adapt the AI to the particular idiosyncrasies of the player by exploiting the information stored in his or her profile.

Game Example

In an FPS, the AI might observe that the player is poor at using a certain weapon or isn't good at jumping from platform to platform. Information like this can then be used to regulate the difficulty of the game, either by exploiting any weaknesses or by shying away from those same weaknesses [Beal02, Houlette03].

Production Systems

A *production system* (also known as a *rule-based system* or *expert system*) is an architecture for capturing expert knowledge in the form of rules. The system consists of a database of rules, facts, and an inference engine that determines which rules should trigger, resolving any conflicts between simultaneously triggered rules. The intelligence of a production system is embodied by the rules and conflict resolution [AIISC03].

Game Example

Many games use a simple version of a production system in the form of a series of rules constructed as if/else statements. However, true production systems are generally considered more structured and elaborate.

The academic community has had some success in creating bot AI for *Quake II* using the *Soar* production system [van Lent99], although the system requires upwards of 800 rules to play as a fairly competent opponent [Laird00]. Another applicable area is sports games, where each AI player must contain a great deal of expert knowledge to play the sport correctly. Microsoft's Sports Group experimented with some success using a production system to drive their team sports games, but the group has since been disbanded for unrelated reasons.

Reinforcement Learning

Reinforcement learning (RL) is a powerful machine learning technique that allows a computer to discover its own solutions to complex problems by trial and error. RL is particularly useful when the effects of the AI's actions in the game world are uncertain or delayed. For example, when controlling physical models like steering an airplane or racing a car, how should the controls be adjusted so that the airplane or car follows a particular path? What sequences of actions should a real-time strategy AI perform to maximize its chances of winning? By providing rewards and punishments at the appropriate times, an RL-based AI can learn to solve a variety of difficult and complex problems [Manslow03].

Reputation System

A *reputation system* is a way of modeling how the player's reputation in the game world develops and changes based on his or her actions. Rather than a single reputation

model, each character in the game knows particular facts about the player [Alt02]. Characters learn new facts by witnessing player actions or by hearing information from others. Based on what the characters know about the player, they might act friendly toward the player or they might act hostile [Brockington03].

Game Example

In a cowboy gunfighter game, the player's reputation might be very important. If the player goes around killing people indiscriminately, others might witness the killings and relay the information to whomever they meet. This would give the player motivation to play nice or to make sure there are no witnesses.

Smart Terrain

Smart terrain is the technique of putting intelligence into inanimate objects. The result is that an agent can ask the object what it does and how to use it. For example, a smart microwave oven knows what it can accomplish (cook food) and how it should be used (open door, place food inside, close door, set cooking time, wait for beep, open door, take food out, close door). The advantage of such a system is that agents can use objects with which they were never explicitly programmed to interact.

The use of smart terrain is enlightened by *affordance theory*, which claims that objects by their very design allow for (or afford) a very specific type of interaction [Gibson87]. For example, a door on hinges that has no handles only permits opening by pushing on the nonhinged side. This is similar to letting the objects themselves dictate how they should be used.

Game Example

The term *smart terrain* was popularized by the very successful game *The Sims*. In *The Sims*, the objects in the game world contain most of the game's intelligence. Each object broadcasts to agents what it has to offer and how it can be used. For example, an agent might be hungry, and food on the table will broadcast "I satisfy hunger." If the agent decides to use the food, the food instructs the agent how to interact with it and what the consequences are. By using this smart terrain model, agents are able to use any new object that is added into the game through expansion packs or from Internet sites.

Speech Recognition and Text-to-Speech

The technology of *speech recognition* enables a game player to speak into a microphone and have a game respond accordingly. In the games industry, there have been a few attempts to add speech recognition to games. The most notable are Sega's *Seaman* for the Sega Dreamcast, and Nintendo's *Hey You, Pikachu!* for the Nintendo 64. While these first attempts were somewhat gimmicky, they serve an important role by feeling out the territory for viable speech recognition in games, both in terms of the current state of the technology and the possibilities for enhancing gameplay. New platforms such as the Nintendo DS have a microphone built-in, which encourages games to support speech recognition.

Text-to-speech is the technique of turning ordinary text into synthesized speech. This allows for endless amounts of speech without having to record a human actor. Unfortunately, currently, virtually no games use text-to-speech technology, perhaps because it sounds rather robotlike. In practice, it's more effective to record a human voice, especially since most games have access to enough disk space to store high-quality audio samples. The quality of voice acting in games has also risen in recent years, which makes bland text-to-speech less appealing. However, for some games, it can be quite entertaining for the player to enter his or her name and have the game speak it. For the right game, text-to-speech can be a novel technology that can set the game apart.

Weakness Modification Learning

Weakness modification learning helps prevent an AI from losing repeatedly to a human player in the same way each time. The idea is to record a key gameplay state that precedes an AI failure. When that same state is recognized in the future, the AI's behavior is modified slightly so that "history does not repeat itself." By subtly disrupting the sequence of events, the AI might not win more often or act more intelligently, but at least the same failure won't happen repeatedly. An important advantage of weakness modification learning is that potentially only one example is required in order to learn [van Rijswijck03].

Game Example

Within a soccer game, if the human scores a goal against the computer, the position of the ball can be recorded at some key moment when it was on the ground before the goal was scored. Given this ball position, the game can create a gravity well vector field that will subtly draw the closest computer players toward that position. This particular vector field is then phased in whenever the ball appears near the recorded position in a similar situation (and phased out when the ball moves away). This example lends itself well to many team sports games such as soccer, basketball, hockey, and perhaps football. However, the general concept is very simple and can be applied to almost any genre.

Summary

Game AI is distinctively different from many other related AI fields. The goal is to create intelligent opponents, allies, and neutral characters that result in an engaging and enjoyable experience for the player. Ultimately, the goal is not to beat the player, but rather to lose in a fun and challenging way.

Most games are populated by agents that sense, think, and act on their own; however, even a single opponent can be thought of as an agent. Advanced agents might also learn and remember in order to present a deeper challenge. It is important to realize that whatever an agent senses, thinks, or remembers, it is completely invisible and inconsequential to the player unless the agent can express the result through actions. An agent's outward appearance through movement, manipulation, animation, and

dialogue is critical to making the agent appear intelligent. Typically, this requires tight integration and collaboration with the people who generate the art assets.

One of the most enduring techniques for endowing intelligence on agents is the ubiquitous finite-state machine. This simple computational model allows complex expertise to be expressed in a simple, easy-to-understand manner that is also convenient to debug. The actions and mindsets of an agent eloquently map to the states of an FSM, further allowing for simple, yet effective modeling of behavior. With the many enhancements developed for FSMs, it is easy to understand why they have become so universal within AI game development.

Finally, there are dozens of common and promising techniques for adding intelligence to games. Each game is unique and might require mixing and matching several different techniques. There is no single solution, and the resulting design is highly dependant on the exact requirements of the game. Therefore, it is critical that a developer becomes familiar with a broad range of techniques in order to experiment and make intelligent implementation decisions.

Exercises

1. Name several simple ways to make an AI opponent difficult to beat.
2. How could an agent apparently get better at playing a game over time without actually learning or remembering anything?
3. Design an FSM for a patrol behavior. For example, a patrol behavior might visit three different locations in an endless loop. Compose your answer as a UML diagram.
4. Design an FSM for a smart patrolling guard. Consider how the guard might detect intruders and what his reaction might be over his lifetime. Compose your answer as a UML diagram.
5. Take the FSMs you designed in the previous two exercises and convert them to the fictional FSM scripting language as described in Listing 5.3.2.


ON THE CD

6. Using the State Machine Language included on the companion CD-ROM, investigate the messaging scheme that allows agents to communicate with each other. Write a short explanation of each messaging function (starting with `SendMsg`). Give examples of how each might be useful.
7. Research a recent game that has received acclaim for its AI. What does the game do particularly well with regard to AI? What AI techniques are likely being used?
8. The game *Black & White* was hailed for its interesting and innovative use of AI. Research this game and comment on how the game design allowed the AI to be showcased.
9. Write a one-page essay on the future of game AI. What will it look like in 10 or 20 years? How about 100 years?

References

[AIISC03] "Working Group on Rule-Based Systems Report," *The 2003 AIISC Report*, AIISC, 2003, available online at *www.igda.org/ai/report-2003/aiisc_rule_based_systems_report_2003.html.*

[Alt02] Alt, Greg, and King, Kristin, "A Dynamic Reputation System Based on Event Knowledge," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Beal02] Beal, C., Beck, J.; Westbrook, D.; Atkin, M.; and Cohen, P., "Intelligent Modeling of the User in Interactive Entertainment," *AAAI Stanford Spring Symposium*, 2002, available online at *www-unix.oit.umass.edu/~cbeal/papers/AAAISS02Slides.pdf and www-unix.oit.umass.edu/~cbeal/papers/AAAISS02.pdf.*

[Brockington02a] Brockington, Mark, "Level-Of-Detail AI for a Large Role-Playing Game," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Brockington03] Brockington, Mark, "Building A Reputation System: Hatred, Forgiveness, and Surrender in Neverwinter Nights," *Massively Multiplayer Game Development*, Charles River Media, 2003.

[Brooks89] Brooks, Rodney, "How to Build Complete Creatures Rather than Isolated Cognitive Simulators," *Architectures for Intelligence*, Lawrence Erlbaum Associates, Fall 1989, available online at *www.ai.mit.edu/people/brooks/papers/how-to-build.pdf.*

[Cain02] Cain, Timothy, "Practical Optimizations for A* Path Generation," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Darby03] Darby, Alex, "Vehicle Racing Control Using Insect Intelligence," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Dawson02] Dawson, Chad, "Formations," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Evans02] Evans, Richard, "Varieties of Learning," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Fu03] Fu, Dan, and Houlette, Ryan, "Constructing a Decision Tree Based on Past Experience," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Gibson87] Gibson, James, *The Ecological Approach to Visual Perception*, Lawrence Erlbaum Assoc., 1987.

[Grimani03] Grimani, Mario, "Wall Building for RTS Games," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Hargrove03a] Hargrove, Chris, "Simplified Animation Selection," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Hargrove03b] Hargrove, Chris, "Pluggable Animations," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Higgins02a] Higgins, Dan, "Generic A* Pathfinding," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Higgins02b] Higgins, Dan, "How to Achieve Lightning-Fast A*," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Higgins02c] Higgins, Dan, "Terrain Analysis in an RTS—The Hidden Giant," *Game Programming Gems 3*, Charles River Media, 2002.

[Houlette01] Houlette, Ryan; Fu, Daniel; and Ross, David, "Towards an AI Behavior Toolkit for Games," *AAAI Spring Symposium on AI and Interactive Entertainment*, 2001, *www.qrg.northwestern.edu/aigames.org/2001papers.html.*

[Houlette03] Houlette, Ryan, "Player Modeling for Adaptive Games," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[IDIS99] "Bayesian Networks," *IDIS Lab*, 1999, available online at *http://excalibur.brc.uconn.edu/~baynet/.*

[Isla02] Isla, Damian, and Blumberg, Bruce, "Blackboard Architectures," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Kent03] Kent, Tom, "Multi-Tiered AI Layers and Terrain Analysis for RTS Games," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Laird00] Laird, John, and van Lent, Michael, "Human-level AI's Killer Application: Interactive Computer Games," AAAI, 2000, available online at *ai.eecs.umich.edu/people/laird/papers/AAAI-00.pdf.*

[Laramée02b] Laramée, François Dominic, "Using N-Gram Statistical Models to Predict Player Behavior," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Laramée03] Laramée, François Dominic, "Dead Reckoning in Sports and Strategy Games," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Lidén02] Lidén, Lars, "Strategic and Tactical Reasoning with Waypoints," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Manslow03] Manslow, John, "Using Reinforcement Learning to Solve AI Control Problems," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[McCuskey00] McCuskey, Mason, "Fuzzy Logic for Video Games," *Game Programming Gems*, Charles River Media, 2000.

[Orkin02a] Orkin, Jeff, "A General-Purpose Trigger System," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Orkin02b] Orkin, Jeff, "A Data-Driven Architecture for Animation Selection," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Orkin03a] Orkin, Jeff, "Applying Goal-Oriented Action Planning to Games," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Orkin03b] Orkin, Jeff, "Simple Techniques for Coordinated Behavior," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Porcino03] Porcino, Nick, "An Architecture for A-Life," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Pottinger99a] Pottinger, Dave, "Coordinated Unit Movement," *Game Developer Magazine*, January 1999, available online at *www.gamasutra.com/features/19990122/movement_01.htm.*

[Pottinger99b] Pottinger, Dave, "Implementing Coordinated Movement," *Game Developer Magazine*, February 1999, available online at *www.gamasutra.com/features/19990129/implementing_01.htm.*

[Rabin98] Rabin, Steve, "Making the Play: Team Cooperation in Microsoft Baseball 3D," *Computer Game Developers Conference Proceedings*, 1998, available on the *AI Game Programming Wisdom 2* CD-ROM, Charles River Media, 2002.

[Rabin00a] Rabin, Steve, "A* Speed Optimizations," *Game Programming Gems*, Charles River Media, 2000.

[Rabin00b] Rabin, Steve, "A* Aesthetic Optimizations," *Game Programming Gems*, Charles River Media, 2000.

[Rabin02] Rabin, Steve, "An Extensible Trigger System for AI Agents, Objects, and Quests," *Game Programming Gems 3*, Charles River Media, 2002.

[Rabin03] Rabin, Steve, "Filtered Randomness for AI Decisions and Game Logic," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Reed03] Reed, Christopher, and Geisler, Benjamin, "Jumping, Climbing, and Tactical Reasoning: How to Get More out of a Navigation System," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Reynolds87] Reynolds, Craig, "Flocks, Herds, and Schools: A Distributed Behavioral Model," *Computer Graphics, 21(4)* (SIGGRAPH '87 Conference Proceedings), pp. 25–34, 1987, available online at *www.red3d.com/cwr/papers/1987/boids.html.*

[Reynolds99] Reynolds, Craig, "Steering Behaviors For Autonomous Characters," Game Developers Conference Proceedings, 1999, available at *www.red3d.com/cwr/papers/1999/gdc99steer.pdf.*

[Reynolds01] Reynolds, Craig, "Boids," available online at *www.red3d.com/cwr/boids/.*

[Reynolds02] Reynolds, John, "Tactical Team AI Using a Command Hierarchy," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Scutt02] Scutt, Tom, "Simple Swarms as an Alternative to Flocking," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Stein02] Stein, Noah, "Intercepting a Ball," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Stentz94] Stentz, Tony, "Original D*," *ICRA 94*, 1994, available online at *www.frc.ri.cmu.edu/~axs/doc/icra94.pdf.*

[Sweetser03a] Sweetser, Penny, "How to Build Evolutionary Algorithms for Games," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Tozour01] Tozour, Paul, "Influence Mapping," *Game Programming Gems 2*, Charles River Media, 2001.

[Tozour02a] Tozour, Paul, "The Perils of AI Scripting," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Tozour02b] Tozour, Paul, "Introduction to Bayesian Networks and Reasoning Under Uncertainty," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Tozour03a] Tozour, Paul, "Search Space Representations," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Tozour03b] Tozour, Paul, "Using a Spatial Database for Runtime Spatial Analysis," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Tozour03c] Tozour, Paul, "Stack-Based Finite-State Machines," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[van der Sterren00] van der Sterren, William, "AI for Tactical Grenade Handling," *CGF-AI*, 2000, available online at *www.cgf-ai.com/docs/grenadehandling.pdf.*

[van Lent99] van Lent, M., Laird, J.; Buckman, J.; Harford, J.; Houchard, S.; Steinkraus, K.; and Tedrake, R., "Intelligent Agents in Computer Games," *AAAI*, 1999, available online at *hebb.mit.edu/people/russt/publications/Intelligent_Agents_in_Computer_Games(AAAI99).pdf.*

[van Rijswijck03] van Rijswijck, Jack, "Learning Goals in Sports Games," *Game Developers Conference Proceedings*, 2003, available at *www.gdconf.com/archives/2003/Van_Ryswyck_Jack.doc* or *www.cs.ualberta.ca/~javhar/research.html.*

[Woodcock02] Woodcock, Steven, "Recognizing Strategic Dispositions: Engaging the Enemy," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Yiskis03a] Yiskis, Eric, "Finite-State Machine Scripting Language for Designers," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Yiskis03b] Yiskis, Eric, "A Subsumption Architecture for Character-Based Games," *AI Game Programming Wisdom 2*, Charles River Media, 2003.