Bonus Chapter 23

The Lowdown on Artificial Intelligence

In This Chapter

- Finding out about artificial intelligence
- Understanding simple deterministic algorithms
- Making your game creature follow a pattern or script
- Getting an overview of behavioral state systems
- Using memory and learning to enhance your game's artificial intelligence
- Discovering neural networks and genetic algorithms

This chapter is going to answer a lot of questions on the art of making game creatures and objects seem as if they are thinking. In fact, depending on how you look at the issue, artificial intelligence is not at all artificial. It is an intelligence based on logic, mathematics, probability, and memory. If you read this entire chapter, you will be able to write code and algorithms to make your game creatures perform in a reasonable manner and do almost anything that "professional" games do.

Introduction to Artificial Intelligence

Artificial intelligence (AI), in the most academic sense of the word, has come to mean building a piece of hardware or writing computer software that allows the machine to "think" or process information in a fashion somewhat similar to the way humans do. A few years ago, applications in AI were just starting to surface; today, AI and other related fields, such as artificial life-forms and intelligent agents, are maturing at an exponential rate.

Today, right now, systems exist that are basically “alive” as far as anyone can define life. A number of companies have created artificial life-forms that live, die, explore, get sick, reproduce, evolve, get depressed, get hungry, and so on — all within the virtual domain of the computer. This kind of technology has been made possible by the following developments:

- **Artificial neural networks:** Crude approximations of a human brain network
- **Genetic algorithms:** A set of techniques and suppositions that are used for evolution of software systems based on biological paradigms

Sound far out? It is, but the technology is real, and it’s only going to get more advanced. Remember, cloning a human was once considered science fiction; now that’s well within the realm of possibility.

Getting back to Earth, you aren’t going to create anything as complex as state-of-the-art AI for your games. This chapter looks at the most simplistic and fundamental techniques game programmers use to create intelligent creatures — or, at least, creatures that *seem* intelligent. In fact, many game programmers are still very behind on AI technology and haven’t begun to really embrace all the possibilities in the field. I suspect that AI and related technologies are going to make the same kind of impact on the gaming world that the *Doom* graphics technology made a few years back.

Truthfully, 3D graphics are starting to slow down. Game creatures are looking pretty real these days, but they still act pretty dumb. The next super-cool game is going to be one that not only looks good but, more importantly, offers characters that think and are as cunning and devious as the best of us.

As you read the following pages and experiment with the accompanying programs, remember that all these techniques are just that: techniques. There isn’t a right way or a wrong way, just a way that works. If the computer tank can kick your player’s butt, then that’s all you need. If it can’t, then you need to do more.

Regardless of how primitive the underlying AI techniques are, the human players will always imagine more detail and project personalities of their own onto your virtual opponents. This concept is key: The player will believe that the objects in the game are plotting, planning, and thinking, as long as they look like they are. Get it?

I cover three types of useful game AI for game characters (such as alien invaders) or items (such as asteroids) in this chapter:

- **Deterministic algorithms:** Predetermined behaviors, random or otherwise

- **Patterns and scripts:** Series of actions determined by various inputs, from you or (unknowingly) from the player
- **State machines:** Behaviors based on conditions and results of game play
- **Neural Networks:** Models of computation based on biological brain functions

From now on in the chapter, I refer to anything that has to do with making game creatures act intelligently — software, algorithms, and techniques — as an *AI*.

Simple Deterministic Algorithms

Deterministic algorithms are behaviors that are predetermined or programmed. For example, take a look at the AI for the asteroids in Star Ferret (as shown in Figure 23-1).

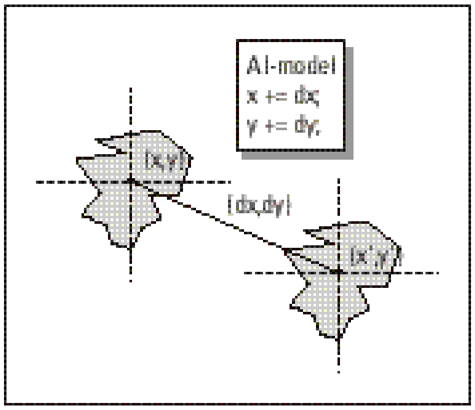


Figure 23-1: The asteroid's AI.

The process is very simple. The AI creates an asteroid and then sends it in a random direction at a random velocity (mostly downward). The following code shows this type of intelligence:

```
asteroid_x += asteroid_x_velocity;
asteroid_y += asteroid_y_velocity;
```

The asteroids have one mission: to follow their course. Granted, the AI is simple — the asteroids don't process any outside input, make course changes, and so on. But the asteroids do "know" how to explode, so in a sense they are intelligent. However, their intelligence is rather deterministic or predictable.

This deterministic intelligence is the first kind of AI that I want to explain — the simple, predictable, programmable kind. In this class of AI, a number of techniques were born in the *Pong/Pac Man* era.

Random motion

Just one step above moving an object in a straight line or curve is moving an object randomly or changing its properties randomly (as shown in Figure 23-2). For example, suppose that you want to model an atom, fly, or something similar that doesn't have a lot of brains but does have a fairly predictable behavior — to bounce around in an erratic way without much thought. (Well, at least the behavior looks that way.)

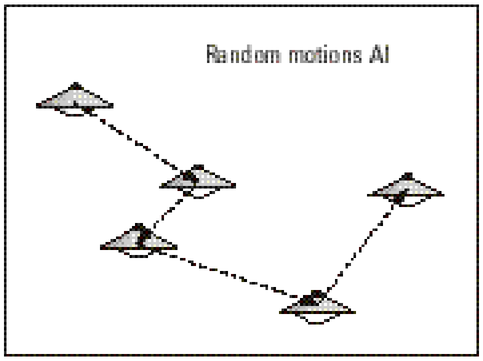


Figure 23-2: Random-motion AI.

As a starting AI model, try the following code to model a fly brain:

```
fly_x_velocity = -8 + rand()%16;
fly_y_velocity = -8 + rand()%16;
```

Then you can move the fly around for a few cycles:

```
int fly_count = 0; // fly's "new thought" counter
// fly in the same direction for 10 ticks of time
while(++fly_count < 10)
    {
    fly_x+=fly_x_velocity;
    fly_y+=fly+y_velocity;
    } // end while
// insert similar code to pick a new direction and loop
```

In this example, the fly picks a random direction and velocity, moves that way for a moment, and then picks another. That sounds like a fly to me! Of course, you may want to add even more randomness, such as changing how long the motion occurs rather than fixing it at 10 cycles. In addition, you may want to weigh certain directions more heavily. For example, when you select the new direction, you may want to lean toward westward directions rather than eastward to simulate the breeze.

In any case, you can see that it's possible to make some creature seem intelligent with very little code. As a working example, check out PROG23_1.CPP and the executable PROG23_1.EXE on the CD. It's an example of the fly brain in action.

Random motion is a very important part of behavioral modeling of intelligent creatures. I live in Silicon Valley, and I can attest that the people driving on the roads around here make random lane changes and even drive in the wrong direction, just like the brainless motion of a fly.

Tracking

Although random motion can be interesting and totally unpredictable, it's rather boring because no matter what, it works the same way: randomly. If you want to add excitement to your games, consider the next step up on the AI ladder: algorithms that perceive something in the environment and then react to it in some manner. For an example, I have chosen *tracking* algorithms. A tracking AI senses the position of the object being tracked and then changes the trajectory of the AI object so that it moves toward the object.

The tracking can be "brute force" by literally vectoring directly toward the object or a more realistic model of turning toward the object much like a heat-seeking missile (see Figure 23-3).

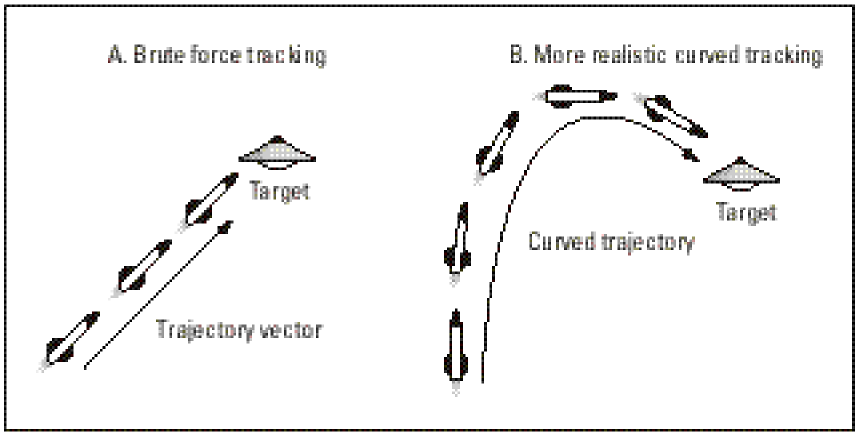


Figure 23-3: Tracking methods.

For an example of the brute force method (because the heat-seeking method takes a little more physics modeling than I want to show right now), take a look at the following algorithm:

```
// given: player is at player_x, player_y
// and game creature is at
// monster_x, monster_y
// first test x-axis
if (player_x > monster_x)
   monster_x++;
if (player_x < monster_x)
   monster_x--;
// now y-axis
if (player_y > monster_y)
   monster_y++;
if (player_y < monster_y)
   monster_y--;
```

If you dropped the preceding AI into a simple demo, the player would be tracked down as if the monster were the Terminator! The code is simple, but effective. This code is much the same way that the *Pac Man* AI was written. Of course, *Pac Man* could only make right-angle turns and had to move in a straight line and avoid obstacles, but this AI is in the same ballpark.

For an example, check out PROG23_2.CPP and the executable PROG23_2.EXE on the CD. You control a ghost with the keyboard arrow keys and a bat tries to hunt down the ghost.

Evasion

Starting to get little quantum disturbances in your brain? That is, are you getting some ideas? Good! The next AI technique provides a way for the creatures in the game to get away from you. Making an evasion AI that mimics the action of a powered-up *Pac Man* chasing ghosts is simple. In fact, you already have the code! The preceding tracking code is the exact opposite of what you want; just flip the equalities around in the code and presto! You have an evasion algorithm. Here’s the code after the inversions:

```
// given: player is at player_x, player_y
// and game creature is at
// monster_x, monster_y
// first, test x-axis
if (player_x  < monster_x)
   monster_x++;
```

```
if (player_x > monster_x)
   monster_x--;
// now y-axis
if (player_y <  monster_y)
   monster_y++;
if (player_y > monster_y)
   monster_y--;
```

You may have noticed that the preceding code doesn't include a conditional for equal to (==). This omission is because I don't want the object to move in this case. I want the object to sit on the player. If you want to, you can process the case when the positions are equal in a different way.

Now you can create a fairly impressive AI system with just random motion, chasing, and evasion. In fact, you have enough to make a *Pac Man* brain. Not what AI researchers would call impressive, but good enough to sell 100 million copies, so not too bad! To check out evasion in action, run `PROG23_3.CPP` and the executable `PROG23_3.EXE` on the CD. It is basically the same as `PROG23_2.CPP`, but with the evasion AI rather than the tracking AI.

Patterns and Scripts

Algorithmic and deterministic algorithms such as those I cover in the last section are great, but sometimes you need to make a game creature follow a sequence of steps or a script of sorts.

For example, when you start your car (or get on your bike, whatever the case may be), you perform a specific sequence of steps:

1. **Get the keys out of your pocket.**
2. **Put the key in the car door.**
3. **Open the door.**
4. **Get in the car.**
5. **Close the door.**
6. **Put the key in the ignition.**
7. **Turn the key to start the car.**

Or if you're Bill Gates, you just say: "To the opera, James."

My point: A sequence occurs that you don't think much about; you just replay it every time. Of course, if something goes wrong, you may change your sequence (such as pressing the gas pedal, or jump-starting the car because you left the lights on last night).

Patterns are an important part of intelligent behavior, and even the epitome of intelligent creatures on this planet — people — uses them. Of course, if any nonhuman, alien life-forms are reading this, please don't take that as an insult.

Basic patterns

The complexity of creating patterns for game creatures depends on the game creature itself. For example, motion-control patterns are very simple to implement. Suppose that you are writing a shoot-'em-up game similar to *Phoenix* or *Galaxian*. The alien attackers must perform a left-right pattern and then, at some point, attack you with a specific attack pattern. This kind of pattern or scripted AI can be achieved using a number of different techniques, but I think that the easiest technique to illustrate the process is based on interpreted motion instructions (as shown in Figure 23-4).

Each motion pattern is stored as a sequence of directions or instructions, as shown in Table 23-1.

Table 23-1 A Hypothetical Pattern Language Instruction Set

Meaning	*Value*
GO_FORWARD	1
GO_BACKWARD	2
TURN_RIGHT_90	3
TURN_LEFT_90	4
SELECT_RANDOM_DIRECTION	5
STOP	6

Along with each directional instruction may be another *operand* or piece of data that further qualifies the instruction, such as how long to perform the task. Therefore, the pattern language instruction format may look like:

```
[INSTRUCTION], [OPERAND]
```

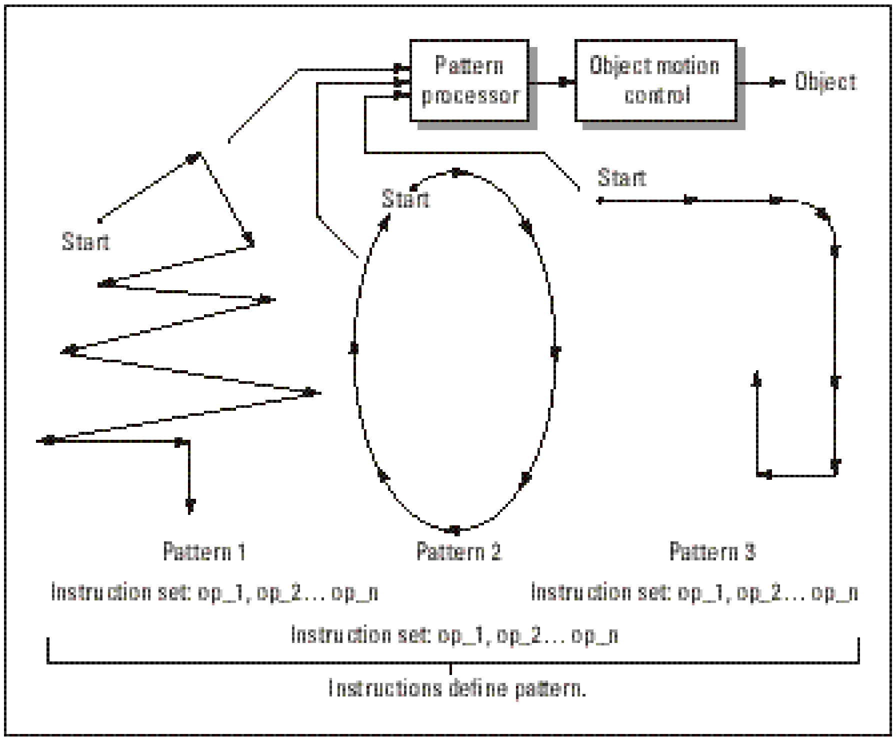

Pattern processor
Object motion control
Object
Start
Start
Start
Pattern 1
Pattern 2
Pattern 3
Instruction set: op_1, op_2… op_n
Instruction set: op_1, op_2… op_n
Instruction set: op_1, op_2… op_n
Instructions define pattern.

Figure 23-4: The pattern engine.

INSTRUCTION stands for a number from the list in Table 23-1 (encoded as a single number usually), and OPERAND stands for another number that helps further define or modify the behavior of the instruction. With this simple instruction format, you can create a program (sequence of instructions) that defines the pattern. Then you can write an interpreter that feeds from a source pattern and controls the game creature appropriately.

For example, suppose that your pattern language is formatted with the first number being the instruction itself, and the second number indicating how long to perform the motion in cycles. Creating a square pattern with a spin and stop (as shown in Figure 23-5) would be easy.

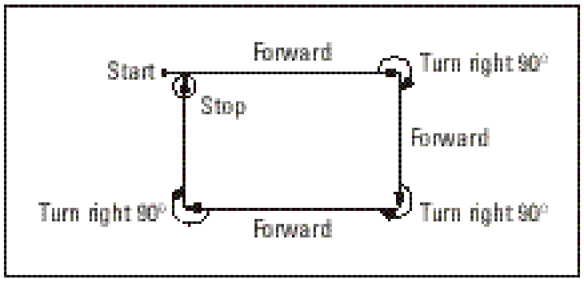


Figure 23-5: Detailed square pattern.

Here's an example of that detailed square pattern in [INSTRUCTION, OPERAND] format:

```
int num_instructions = 6; // number of instructions in script pattern
// the following holds the actual pattern script
int square_stop_spin[
    1,30, 4,1,  // go forward then turn right
    1,30, 4,1,  // go forward and turn right
    1,30, 4,1,  // go forward and turn right
    1,30,       // go forward and finish square
    6,60,       // stop for 60 cycles
    4,8, }; // spin for 8 cycles
```

Of course, you may want to use a better data structure than an array. For example, you can use a class or structure containing a list of records in INSTRUCTION, OPERAND format along with the number of instructions. This way, you can very easily create an array of these structures, each containing a different pattern, and then select a pattern and pass it to the pattern processor very efficiently.

To process the pattern instructions, all you need is a big switch() statement that interprets each instruction and instructs the game creature to do what it is supposed to do, like this:

```
// points to first instruction (2 words per instruction)
int instruction_ptr = 0;
// first extract the number of cycles
int cycles = square_stop_spin[instruction_ptr+1];
// now process instruction
switch(square_stop_spin[instruction_ptr])
{
case GO_FORWARD:  // move creature forward...
    break;
case GO_BACKWARD: // move creature backward...
    break;
case TURN_RIGHT_90: // turn creature 90 degrees right...
        break;
case TURN_LEFT_90:  // turn creature 90 degrees left...
        break;
case SELECT_RANDOM_DIRECTION: // select random dir...
        break;
case STOP: // stop the creature
        break;
} // end switch
```

```
// advance instruction pointer (2 words per instruction)
instruction_ptr+=2;
// test whether end of sequence has been detected...
if (instruction_ptr > num_instructions*2)
   { /* sequence over */ }
```

And of course, you would add the logic to track the cycle counter and make the motion happen.

There's one catch to all this pattern stuff: *reasonable motion.* Because the game object is feeding off of a pattern, it may decide to select a pattern that forces the object to smash into something. If the pattern AI doesn't take this possibility into consideration, then patterns will be followed blindly. Thus, you must have a feedback loop with your pattern AI (as with any AI for that matter) that instructs the AI that it has done something illegal, impossible, or unreasonable, and it must reset to another pattern or strategy (as shown in Figure 23-6).

Stop for a minute to think about the power of patterns. With them, you can record hundreds of moves and flight patterns. You can create in minutes and play back patterns that would be nearly impossible to create in any reasonable amount of time using other AI techniques. By using patterning, you can make a game creature look as if it is extremely intelligent. This is one of the AI techniques used by nearly all games, including most fighting games such as *TeKeN, Soul Blade,* and *Mortal Kombat.*

Furthermore, you don't need to stop with motion patterns. You can use patterns to control weapon selection, animation control, and so on. There's no limit to what you can apply patterns to.

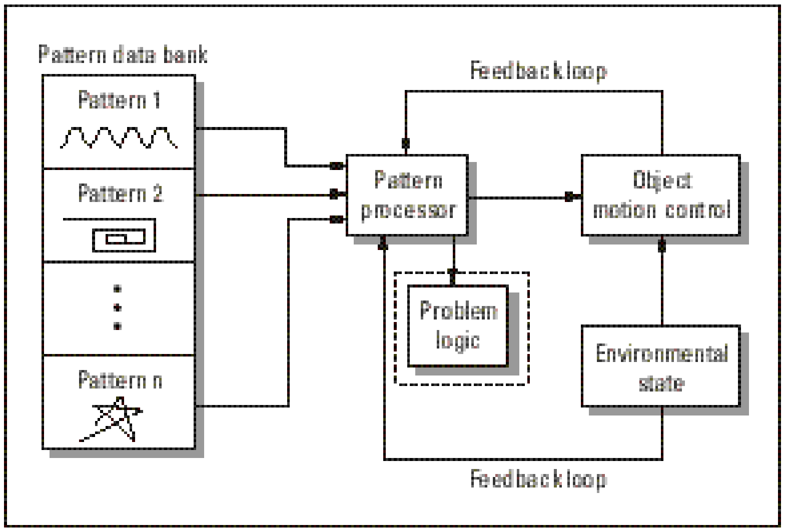


Figure 23-6: Pattern engine with feedback control.

For an example of patterns in action, take a look at PROG23_4.CPP and its executable PROG23_4.EXE. It demonstrates a monster that moves around using a number of patterns and that periodically selects a new random pattern.

Patterns with conditional logic

Patterns are cool, but they are extremely deterministic. That is, once the player has memorized a pattern, the game's challenge is over. The player can always beat your AI because he or she knows what's going to happen next.

The solution to this problem and to others that pop up with patterns is to add a bit of conditional logic that selects patterns based on more than random selection — based on the conditions of the game world and the player himself. Take a look at Figure 23-7 to see this abstractly.

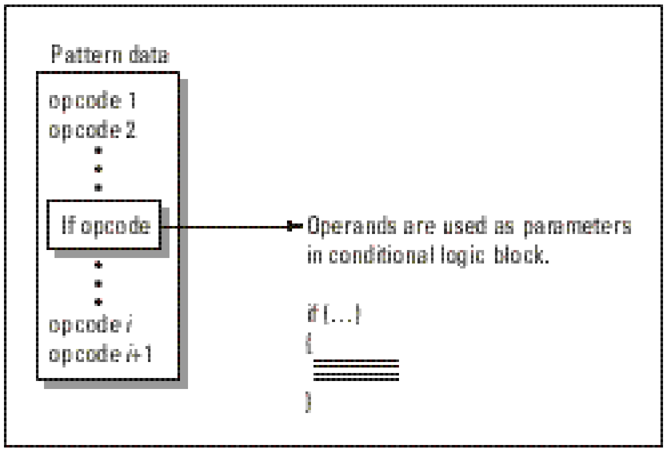


Figure 23-7: Patterns with conditional logic.

Patterns with conditional logic give one more level of abstraction to your AI model. You can select patterns that within them have conditional branches, as well as the patterns being selected based on conditional logic. For example, you may add a new instruction to the pattern language that is a conditional, such as the following:

```
TEST_DISTANCE          7
```

The TEST_DISTANCE conditional may work by testing the distance that the player is to the object performing the pattern. If the distance is too close or too far, the pattern AI engine may change what the object is doing to create a seemingly more intelligent opponent.

For example, you can insert a TEST_DISTANCE instruction every few instructions in a standard pattern like this:

```
TURN RIGHT 90, GO FORWARD, STOP, ...TEST DISTANCE,
                ...TURN_LEFT_90,...TEST_DISTANCE, ... GO_BACKWARD
```

The pattern does its thing, but every time a TEST_DISTANCE instruction is encountered, the pattern AI uses the operand following the TEST_DISTANCE instruction as a measure to test the player's position. If the player is getting too far away, then the pattern AI stops the current pattern and branches to another pattern or, better yet, switches to a deterministic tracking algorithm to get closer to the player. Take a look at the following code:

```
if (instruction_stream[instruction_ptr] == TEST_DISTANCE)
{
// obtain distance; note that on the test
// instructions the operand is no
// longer a time or cycle count
// but becomes context-dependent
int min_distance = instruction_stream[instruction_ptr];
// if statement to test whether player is too far
if (Distance(player, object) > min_distance)
    {
    // set system state to switch to track
    ai_state = TRACK_PLAYER;
    // or you can insert code that just switches to
    // another pattern and waits for
    // the object to possibly get closer
    } // end if
} // end if
```

Of course, you can perform conditional tests of almost limitless complexity in the pattern script. In addition, you may want to create patterns on the fly and use them, for example, to mimic the player's motion. You can sample what the player does each time she kills one of your game characters and then use the same tactic against her!

In conclusion, technology like this (much more sophisticated, of course) is used in many sports games, such as football, baseball, and hockey, as well as action and strategy games. It enables the game objects to make predictable moves — but if they need to change their minds, they can.

As an example, PROG23_5.CPP and its executable PROG23_5.EXE illustrate the conditional technique. You control a bat creature with the arrow keys, and an AI skeleton is on the screen. The skeleton performs randomly selected patterns until you get too far away; then it gets lonely and chases you because it wants your attention. Reflect on what I just said. . . . I'm describing a codependent skeleton. In fact, I placed an emotional motive on 100 lines of computer code. But isn't that what it seems like from a spectator's point of view?

Behavioral State Systems

If you've read this book from the first page to here, you have seen quite a few finite state machines (FSMs) in various forms — a blinking light, the main event loop state machine, and so on. Now, I want to formalize how FSMs generate AIs that exhibit intelligence.

To create a truly robust FSM, you need two properties:

- A reasonable number of states that each represent a different goal or motive
- Lots of input to the FSM, such as the state of the environment and the other objects

The need for a reasonable number of states is easy enough to understand and appreciate. Humans have hundreds, if not thousands, of emotional states that direct our motives and goals. And within each of these emotional states, we may have further substates. The point is, a game character should be able to act as if it is motivated by emotion. At the very least, the character should move around in a free manner. For example, you may set up the following states:

- State 1: Move forward.
- State 2: Move backward.
- State 3: Turn.
- State 4: Stop.
- State 5: Fire weapon.
- State 6: Chase player.
- State 7: Evade player.

States 1 to 4 are straightforward, but states 5, 6, and 7 may need substates to be properly modeled; I mean, states 5, 6, and 7 may need more than one *phase*. For example, chasing the player may involve turning and then moving forward. (Take a look at Figure 23-8 to see the concept of substates.)

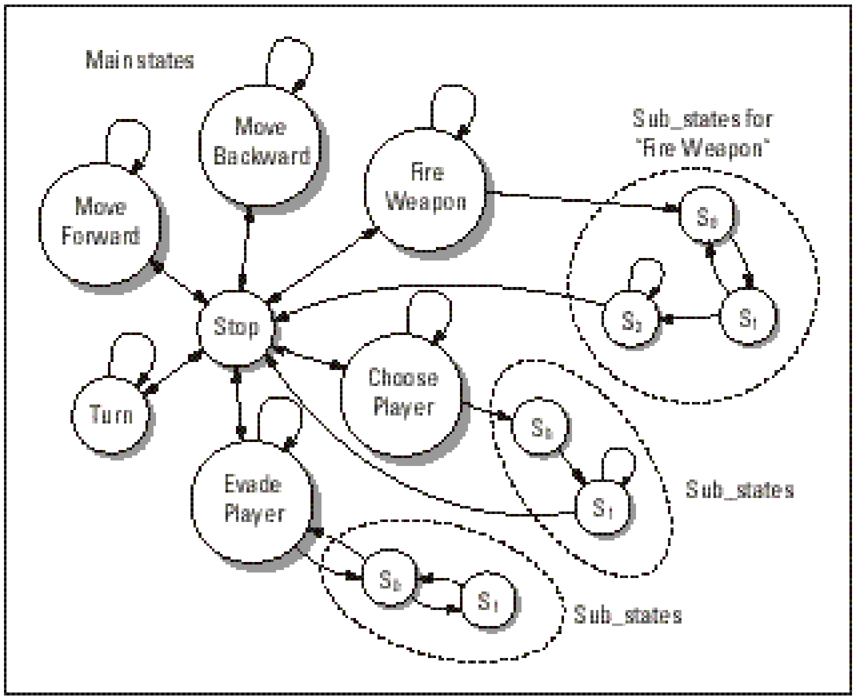


Figure 23-8: A master FSM with substates.

However, don't think that substates must be based on states that exist; they may be totally artificial for the state in question. My point in this discussion on states is that the game object needs to have enough variety to do intelligent things. If the only two states are Stop and Forward, there isn't going to be much action! (Did you ever have one of those stupid remote-control cars that only had two directions: forward and reverse in a left turn?)

Moving on to the second property of robust FSM AIs, you need to have feedback or input from the other objects in the game world and from the player and environment. Entering a state and running it until completion is pretty dumb. The state may have been intelligently selected 100 milliseconds ago, but now events have changed and the AI needs to respond to the player's most recent action. Thus, the FSM needs to track the game state and, if needed, be preempted from its current state into another one.

If you take these ideas into consideration, you can create an FSM that models commonly experienced behaviors such as aggression, curiosity, and so on. In the following sections, you can see how this approach works with some concrete examples, beginning with simple state machines and following up with more advanced personality-based FSMs.

Using simple state machines

At this point, you should be seeing a lot of overlap in the AI techniques. For example, the pattern techniques are based on state machines at the lowest level that perform the actual motion or effect. What I want to do now is take state machines to another level and talk about high-level states that can be implemented with simple conditional logic, randomness, and patterns. In essence, I want to create a virtual brain model that directs and dictates the action of the "creature" that it is controlling.

You can better understand what I'm talking about if you model a few behaviors with the aforementioned techniques. Then, on top of these behaviors, you can place a master FSM to run the show and set the general direction of events and goals.

Most games are of the conflict genre. Whether conflict is the main idea of the game or its underlying theme, in many games the player is running around destroying the enemies and blowing things up. Hence, you can arrive at a few behaviors that a game creature may need to survive, given the constant onslaught of the human opponent. Take a look at Figure 23-9, which illustrates the relationship of the following states:

- Master state 1: Attack.
- Master state 2: Retreat.
- Master state 3: Move randomly.
- Master state 4: Stop or pause for a moment.
- Master state 5: Look for something, such as food, energy, light, dark, or other computer-controlled creatures.
- Master state 6: Select a pattern and perform it.

Right off, you can see the difference in these states in comparison to the previous examples (in the introduction to this section). These states are much higher level, and definitely contain possible substates and/or further logic to make happen. So, analyzing the states from the simplest to the most complex: States 1 and 2 can be accomplished by using a deterministic algorithm. States 3 and 4 are nothing more than a couple of lines of code. State 6 is very complex because the creature must be able to perform complex patterns controlled by the master FSM. Finally, state 5 could be yet another deterministic algorithm or may be a mix of deterministic algorithms along with preprogrammed search patterns that are successful.

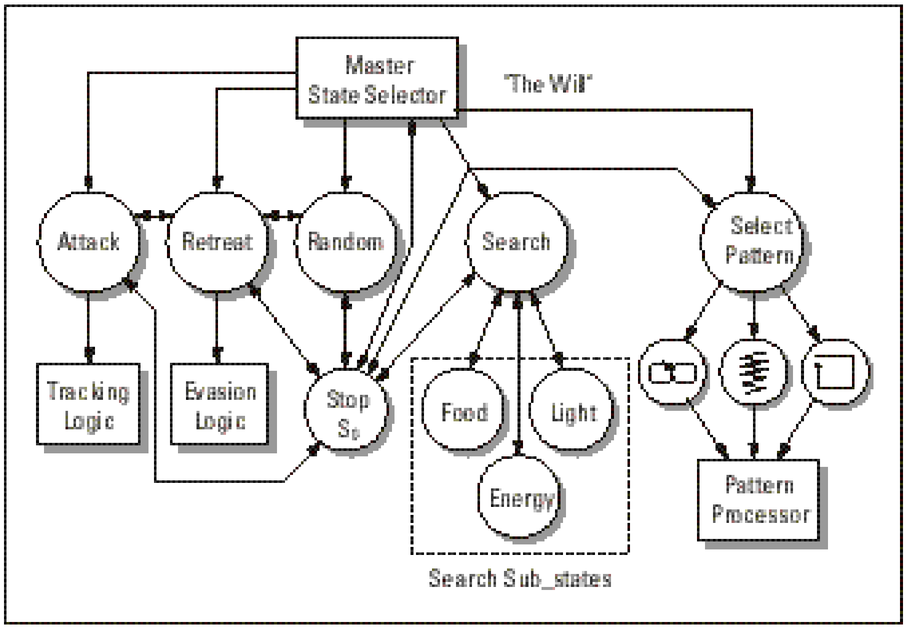


Figure 23-9: Building a better brain.

As you can see, the AI is getting fairly sophisticated. You want to model a creature from the top down — first thinking of how complex you want the AI of the creature to be and then implementing each state and algorithm.

If you refer to Figure 23-9, you see that in addition to the master FSM that selects the states themselves, another part of the AI model is doing the selection. This part represents the “will” or “agenda” of the creature. You can implement this module in a number of ways, such as random selection, conditional logic, and so on. But for now, just know that the states must be selected in an intelligent manner based on the current state of the game.

The following code fragment implements a crude version of the master state machine I discuss earlier in this subsection. Of course, the code is only partially functional because a complete AI would cover many pages, but the most important structural elements are there. Fill in all the blanks and details, generalize, and drop it into your code. For now, just assume that the game’s world consists of the AI creature and the player. Here’s the code:

```
// these are the master states
#define STATE_ATTACK  0       // attack the player
#define STATE_RETREAT 1       // retreat from player
#define STATE_RANDOM  2       // move randomly
#define STATE_STOP    3       // stop for a moment
```

(continued)

(continued)

```
#define STATE_SEARCH  4       // search for energy
#define STATE_PATTERN 5       // select a pattern and execute it
// variables for creature
int creature_state = STATE_STOP, // state of creature
    creature_counter = 0,   // used to time states
    creature_x       = 320, // position of creature
    creature_y       = 200,
    creature_dx      = 0,   // current trajectory
    creature_dy      = 0;
// player variables
int player_x = 10,
    player_y = 20;
// main logic for creature
// process current state
switch(creature_state)
    {
    case STATE_ATTACK:
          {
          // step 1: move toward player
          if (player_x > creature_x) creature_x++;
          if (player_x < creature_x) creature_x--;
          if (player_y > creature_y) creature_y++;
          if (player_y < creature_y) creature_y--;
          // step 2: fire cannon, which has
          // 20 percent probability to hit
          if ((rand()%5)==1)
               Fire_Cannon();
       } break;
    case STATE_RETREAT:
           {
        // move away from player
           if (player_x > creature_x) creature_x--;
           if (player_x < creature_x) creature_x++;
           if (player_y > creature_y) creature_y--;
           if (player_y < creature_y) creature_y++;
           } break;
    case STATE_RANDOM:
       {
           // move creature in random direction
           // that was set when this state was entered
           creature_x+=creature_dx;
           creature_y+=creature_dy;
       } break;
    case STATE_STOP:
            {
            // do nothing!
            } break;
    case STATE_SEARCH:
            {
            // pick an object to search for, such as
            // an energy pellet, and then track it
            // as you would the player
if (energy_x > creature_x) creature_x--;
            if (energy_x < creature_x) creature_x++;
            if (energy_y > creature_y) creature_y--;
            if (energy_y < creature_y) creature_y++;
            } break;
    case STATE_PATTERN:
            {
            // continue processing pattern
            Process_Pattern();
            } break;
    default: break;
    } // end switch
// update state counter and test whether a state
// transition is in order
if (--creature_counter <= 0)
    {
    // pick a new state, use logic, random, script, etc.
    // for now just random
    creature_state = rand()%6;
    // now depending on the state, we might need some
    // setup code goes here...
if (creature_state == STATE_RANDOM)
        {
        // set up random trajectory
        creature_dx = -4+rand()%8;
        creature_dy = -4+rand()%8;
        } // end if
    // perform setups on other states if needed
    // set time to perform state, use appropriate method...
// at 30 fps, 1 to 5 seconds for the state
    creature_counter = 30 + 30*rand()5;
    } // end if
```

In the beginning of this code block, the current state is processed. This task involves local logic, algorithms, and even function calls to other AIs, such as pattern processing. After the state has been processed, the state counter is updated and the code tests to see whether the state is complete. If so, then a new state is selected. Furthermore, if the new state needs setup, then the setup is performed. Finally, a new state count is selected using a random number, and the cycle continues.

A lot of improvements can be made. For example, you can mix the state transitions with the state processing. And you may want to use much more involved logic to make state transitions and decisions.

Adding more personality

Personality is nothing more than a set of predictable behaviors. For example, I have a friend that has a very “tough guy” personality. I can guarantee that, if you say something he doesn’t like, he’ll probably let you know with a swift blow to the head. Furthermore, he’s very impatient and doesn’t like to think much. On the other hand, I have a friend that’s very small and wimpy. He has learned that, due to his size, he can’t speak his mind because he may get smacked. He has developed a much more passive personality.

Of course, human beings are a lot more complex than the previous examples, but these are still good descriptions of those people. Thus, you can model personality types using logic and probability distributions that track a few behavioral traits and place a probability on each. Then this probability graph can make state transitions. Take a look at Figure 23-10 to see what I’m talking about.

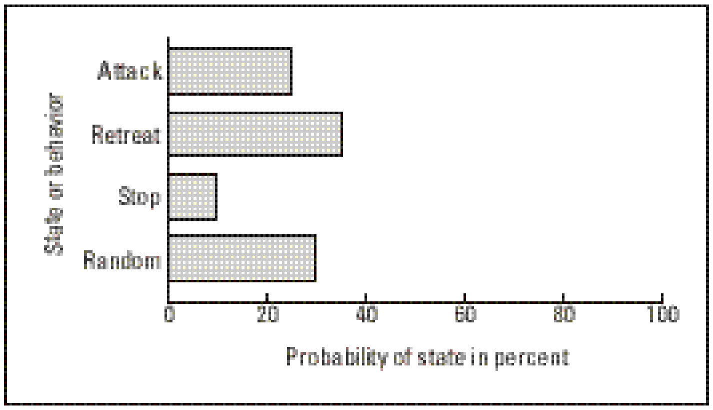


Figure 23-10: Personality distribution for states.

This model contains these four states or behaviors:

- State 1: Attack
- State 2: Retreat
- State 3: Stop
- State 4: Random

Now, instead of randomly selecting a new state, you can create a probability distribution that defines the "personality" of each creature based on these states. For example, Table 23-2 describes my friends Rex (the tough one) and Joel (the wimpy one).

Table 23-2 Personality Probability Distributions

State	*Rex p(x)*	*Joel p(x)*
Attack	50%	15%
Retreat	20%	40%
Stop	5%	30%
Random	25%	15%

The hypothetical data seems to make sense. Rex likes to attack without thinking, and Joel likes to run if he can and thinks much more. In addition, Rex isn't that much of a planner, so he often acts randomly — smashing walls and eating glass. However, Joel knows what he is doing most of the time.

Of course, the entire example was totally fictitious, but I bet that you have a picture of Rex and Joel in your head, or you know people like them. Therefore, my supposition is true: The outward behaviors of a person define, at least in a general way, their personality as perceived by others. Thus, behavioral simulation is a very important asset to your AI modeling and state selection.

To use this technique, simply set up a table consisting of 20 to 50 entries, with each entry as a state. Then fill the table with desired probabilities. When you select a new state, you'll get one that has a little personality in it. For example, here's Rex's probability table in the form of a 20-element array, which means that each element has a 5-percent weight:

```
int rex_pers[20] =
          {1,1,1,1,1,1,1,1,1,1,2,2,2,2,3,4,4,4,4,4}
```

In addition to this technique, you may want to add a *radius of influence*. This term means that, based on some variable such as distance to the player or some other object, the program switches probability distributions (as shown in Figure 23-11). As the figure illustrates, whenever the game creature gets too far away from the player, the creature switches to a nonaggressive search mode rather than the aggressive combat mode it's in whenever it's in close quarters to the player.

Memory and Learning

The last elements of a good AI are *memory* and *learning*. As the AI-controlled creatures in your game run around, they are controlled by state machines, conditional logic, patterns, random numbers, probability distributions, and so on. However, the creatures always "think" on the fly. They never look at their past history to help them make a decision.

For example, what if a creature is in attack mode, and the player keeps dodging to the right, and the creature keeps missing? The creature should track the player's motions and remember that the player moves right during every attack, and then the creature should change its targeting a little.

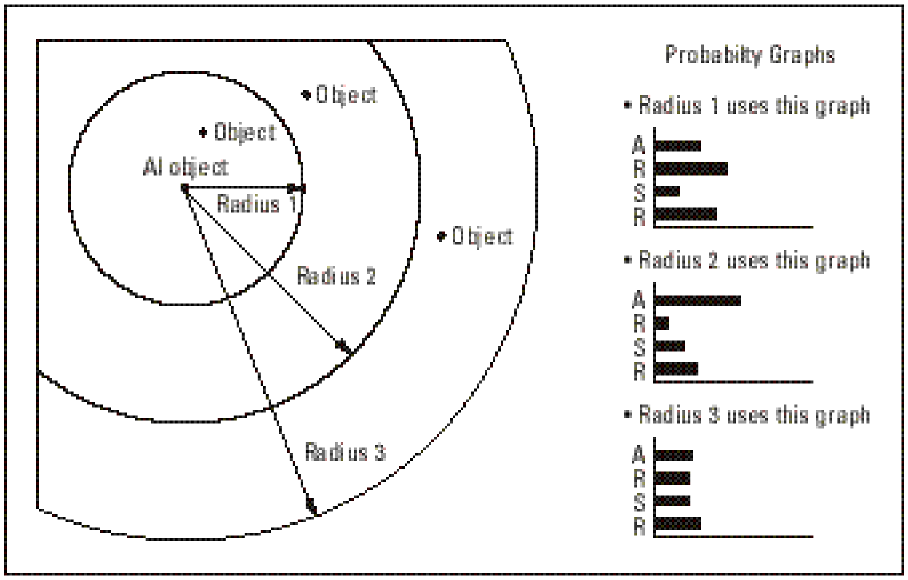


Figure 23-11: Switching personality probabilities based on distance.

For another example, imagine that your game forces creatures to find ammunition, just like the player must, to make the game more realistic. However, every time the creature wants ammo, it has to search randomly for it (maybe with a pattern). But wouldn't it be more realistic if the AI-controlled creature could "remember" locations in which it found ammo last and try that position first?

These are just a couple of examples of using memory and learning to make a game AI seem more intelligent. Frankly, implementing memory is easy to do, but few game programmers ever do it, because they don't have time or they believe that the results aren't worth the effort. No way!

Memory and learning in a game program are very cool, and your players will notice the difference, so it's worth trying to find areas in which simple memory and learning can be implemented with reasonable ease and have a visible effect on the AIs decision making.

All right, so that's the general motive for you to use memory, but how do you do it? The method depends on the situation. For example, take a look at Figure 23-12, which shows a map of a game world with a record attached to each room.

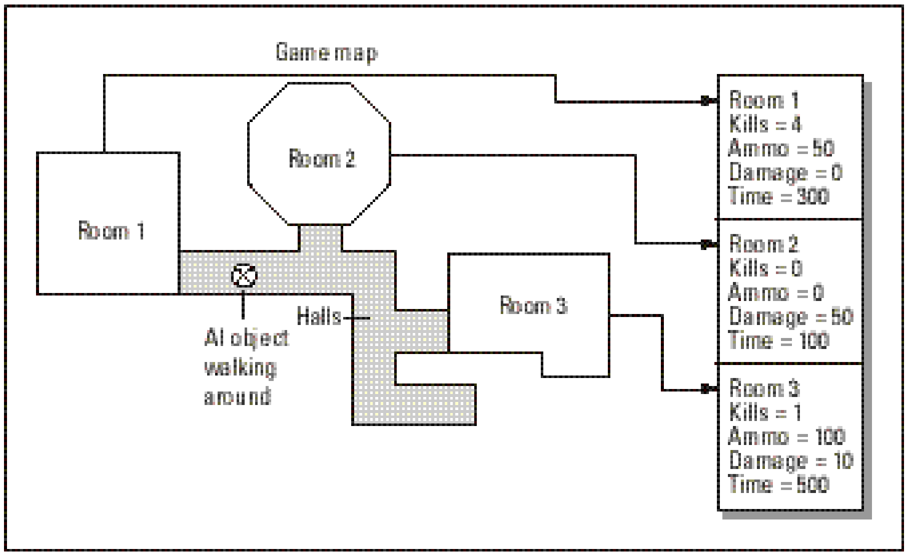

Game map
Room 1
Room 2
Room 3
Halls
AI object walking around
Room 1 Kills = 4 Ammo = 50 Damage = 0 Time = 300
Room 2 Kills = 0 Ammo = 0 Damage = 50 Time = 100
Room 3 Kills = 1 Ammo = 100 Damage = 10 Time = 500

Figure 23-12: Using geographical-temporal memory.

The records in the figure store the following information:

- Kills
- Damage from player
- Ammo found
- Time in room

Now every time the creature processes its AI to have a more robust selection process (that is based on memory and learning), you would refer to the record of events within the creature's memory for the room. For example, when the creature enters a room, you can make a quick check to see whether the creature has sustained a great deal of damage in the room. If so, you can have it back out and try another room.

On the other hand, the creature may run out of ammo; instead of hunting randomly for it, the creature can scan through its memory of all the rooms it has visited and see which one had the best ammo lying around. Of course, for the memory to work, the AI has to continually update the memory every few cycles with new information, but that part is simple.

In addition, you can let creatures exchange information! For example, suppose that one creature bumps into another in a hallway; then they can merge memory records so that they both know of each others' travels. Or maybe, the creature that is stronger performs a force upload on the weaker creature, because it obviously has a better set of probabilities and experience.

The kinds of game innovations you can do with memory and learning are unlimited. The tricky part is working them into the AI in a fair manner. For example, letting the game AI "see" the whole world and memorize it is unfair. The AI should have to explore the game world just like the player does.

A lot of game programmers like to use bit strings or vectors to memorize data. This method is much more compact, and it's easy to flip single bits simulating memory loss or degradation.

Making Your Very Own Frankenstein

Earlier sections in this chapter show you a few techniques to get you started with AI, but you may not know which technique to use in a particular situation or how to mix different techniques to make new models. Here are some basic guidelines:

- Use simple deterministic AIs for objects that have simple behaviors to begin with, such as rocks, missiles, and so on.

- For objects that are supposed to be smart but are more a part of the environment rather than the main action (such as birds that fly around or a spaceship that flies by once in a while), use a deterministic AI coupled with patterns and a bit of randomness.
- For your important game characters that the player interacts with, you definitely need FSMs coupled with the other supporting techniques. However, some creatures don't have to be as smart as others; the FSMs for your basic creatures don't need to have probability distributions for personality coupled with memory for learning.
- Finally, the main computer-controlled character(s) in the game should be very smart. Integrate everything. The AI should be state-driven with a lot of conditional logic, probability, and memory that is used to control state transitions. In addition, the AI should be able to change from one state to another even when the state hasn't come to completion — if conditions arise that make a change necessary.

Basically, you don't need to go all out programming AI for a randomly moving rock; but for a tank that plays against the player, you should invest the time and effort. A model that works well for me is as follows:

- I like to make an AI that has at its highest level a set of conditionals and probabilities that select states. The states emulate a number of behaviors, and usually there are about five to ten.
- I like to use memory to track key elements in the game and use them to make better decisions. Also, I like to throw random generators in a lot of the decisions, even if those decisions are totally simple. This approach adds a little uncertainty to the AI.
- I definitely like to have scripted patterns available to create the illusion of complex thought. However, again, I throw random events into the patterns themselves. For example, my AI moves into a pattern state and selects a circle, but sometimes as it's creating the circle, it makes an egg shape! The point of this randomness is that people aren't perfect, and sometimes we make mistakes. This random quality is very important in game AI, so a lot of virtual coin tosses in the code help to shake things up.

And, finally, a very complex system can evolve from very simple constituents. In other words, even though the AI for each individual creature may not be that complex, their own interaction will create an emergent behavioral system that seems to go beyond its programming. Thus, it's important to help facilitate this evolution with some kind of sharing or merging of information or states between creatures whenever they get close enough or at specific intervals.

Neural Networks, Genetic Algorithms, and Other Esoteric Topics for $1,000

We're on the brink of the 21st century, and it's about time for computers to really start *thinking.* The technologies of neural nets, genetic algorithms, and fuzzy logic are going to make this idea a reality very soon. Maybe we won't like the results — Sky Net, Cylons, or maybe something worse — but every computer scientist in the world knows that it's only a matter of time. Maybe video games are the first place that advanced AI will be used, so read on.

Artificial neural networks

Artificial neural networks have been a popular topic for theory and speculation, but the reality seemed elusive. Well, those days are gone. I can tell you for a fact that in the past three to five years, humankind has made leaps and bounds in the area of artificial neural networks. Not because a major breakthrough has occurred, but because people are finding an interest in them, experimenting with them, and using them. In fact, a number of games use extremely advanced neural networks: *Creatures, Dogz, Fin Fin,* and others.

A neural network is a model of our brain. Our brain consists of 10 to 100 billion brain cells. Each of these cells can both process information and send information. Figure 23-13 is a biological model of a human brain cell containing three main parts: the soma, axon, and dendrites. The soma is the main cell body and performs the processing, and the axon transmits the signal to the dendrites which then pass the signal to other neurons.

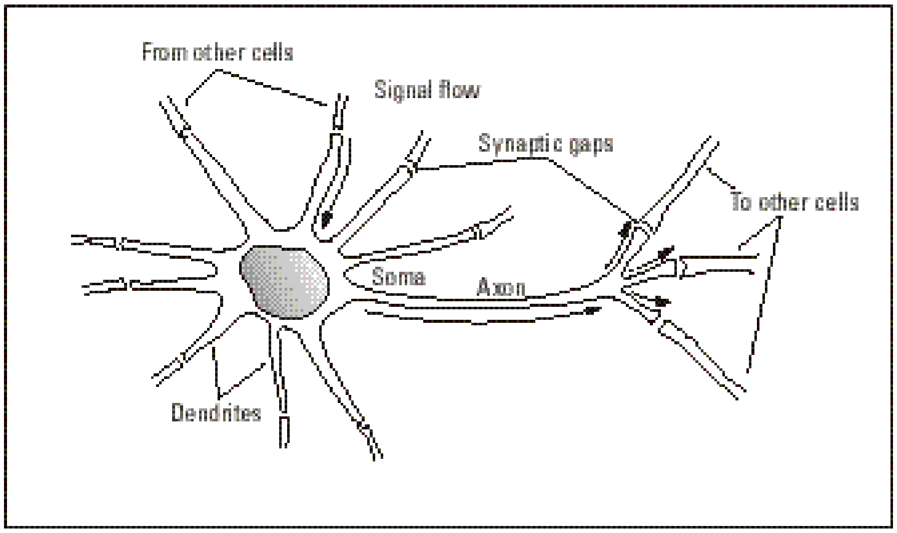


Figure 23-13: A biological neuron.

Each neuron has a fairly simple function: to process input and fire or not to fire. *Firing* means sending an electrochemical signal. So, basically, neurons have a number of inputs and a single output (that may be distributed), and some rule that it uses to process the inputs and generate an output. The rules for processing are extremely complex and beyond the little space I have in this book, but suffice it to say that a summation of signals occurs, and the results of the summation cause the neuron to fire.

Well, that's great, but how can you use this information to make games think? Well, instead of trying to accomplish something as bold as thought or consciousness, maybe you can begin by creating computer models for simple memory, pattern recognition, and learning. I recommend this approach because our organic brains are very good at these tasks, and their digital counterparts are very bad. So it's intriguing to explore a biological computer to perform these tasks. Implementations of simple biological computing models are exactly what artificial neural networks — or simply *neural networks* — are. They are simple digital models that can process information in parallel similar to the way our brains function.

Take a look at the most basic kinds of artificial neuron or *neurode*. The first artificial neural networks were created in 1943 by electrical engineers W. McCulloch and W. Pitts, who wanted to model electronic hardware after the human brain. So they came up with what they called a neurode. Today, the form of the neurode hasn't changed much, as shown in Figure 23-14.

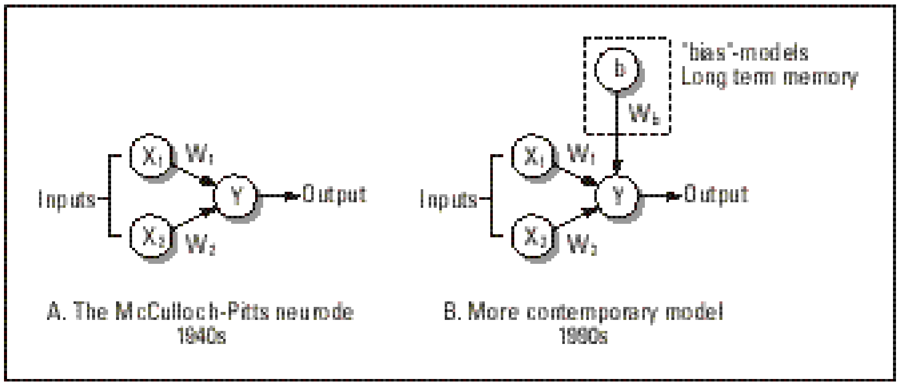


Figure 23-14: Basic artificial neurons.

Neurodes ad nauseam

The neurode consists of a number of inputs *X*(*i*) that are scaled by weights *w*(*i*), summed up, and then processed by an activation function. This activation function may be a simple threshold as in the McCulloch-Pitts (MP) model or a more complex step, linear, or exponential function. But in the case of the MP model, the sum is compared to threshold value θ (theta). If the sum is greater than theta, then the neurode fires; otherwise it doesn't. So mathematically, you have:

McCulloch-Pitts Neurode Summation Function:

$$\text{Output } Y = \sum_{i=1}^{n} X_i * w_i$$

General Neurode with Bias:

$$\text{Output } Y = B*b + \sum_{i=1}^{n} X_i * w_i$$

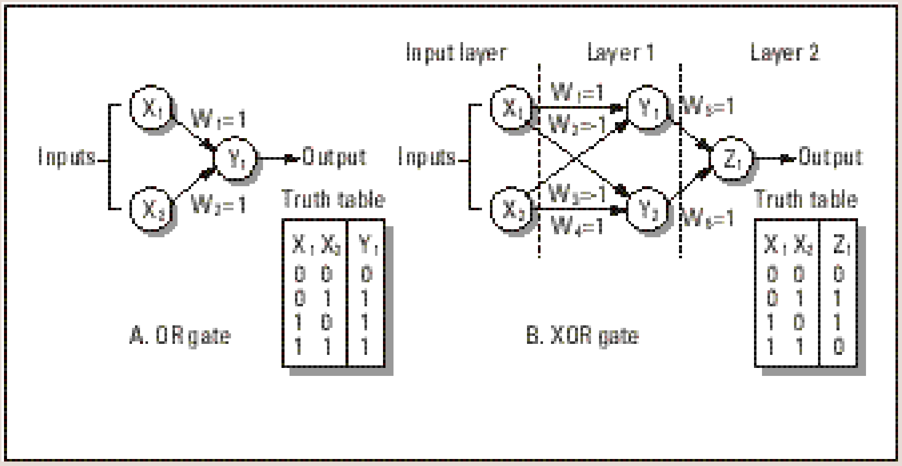
To see how a basic neurode works, assume that two inputs X_1, and X_2 can take on the binary values 0 and 1. Then set the threshold at 2 and w_1=1 and w_2=1. The summation function looks like this:

$Y = X_1*w_1 + X_2*w_2$

Then compare the result to the threshold theta of 2. If Y is greater than or equal to 2, then the neurode fires and outputs a 1.0; otherwise it outputs a 0. The following truth table shows what this single neurode network does.

X1	X2	Sum Y	Final Output
0	0	0	0
0	1	1	0
1	0	1	0
1	1	2	1

If you stare at the truth table for a moment, you'll realize that it basically represents an AND circuit. Cool, huh! So a simple little neurode can perform an AND operation. In fact, by using neurodes, you can build any logic circuit you want to. For example, the following figure shows an OR and an XOR.



X_1	X_2	Y_1
0	0	0
0	1	1
1	0	1
1	1	1

X_1	X_2	Z_1
0	0	0
0	1	1
1	0	1
1	1	0

Real neural networks are very complex, of course. Neural nets can consist of multiple layers, complex activation functions, and hundreds or thousands of neurodes, but if you read the sidebar, you can understand their fundamental building block. Neural networks will continue to bring an unprecedented new level of competition and AI to games. Soon, games will be able to make decisions, learn, and even come up with creative solutions to problems based on making pseudo-random attempts.

Because this is such an important area of interest and I don't have time to properly cover it here, I include on the CD an article on neural networks that I wrote a while ago. This information gives you a more solid foundation on the topic. It covers all the various types of networks, shows you learning algorithms, and illustrates just what they can do. It's called NETWARE.ZIP and is in Microsoft Word 95 format. As a bonus, NETWARE.ZIP includes a number of programs that will enable you to create some simple neural nets.

Genetic algorithms

Genetic algorithms are a method of computing that relies on biological models to evolve solutions (if you're reading this, Dr. Koza, don't have a heart attack). Nature is great at evolution, and genetic algorithms try to capture some of the essence of natural selection and genetic evolution in computer models to help solve problems that normally couldn't be solved by standard means of computing.

Basically, genetic algorithms work like this. You string together a number of informational indicators into a bit vector just like a strand of DNA (as shown in Figure 23-15). This bit vector represents the strategy or coding of an algorithm or solution. You need a few of these bit vectors to begin. Then you process the bit string and whatever it represents by some objective function. The results are it's score. This score is used to compare various strategies to each other.

Hence, a bit vector is really a concatenation of various control variables or settings for some algorithm. You must come up with a few experimental sets of values to start with. Then you run each set, and you get the score of each set. You find that out of the five you created manually, two of them did really well, and the other three did really bad. Now here's where the genetic algorithm comes in.

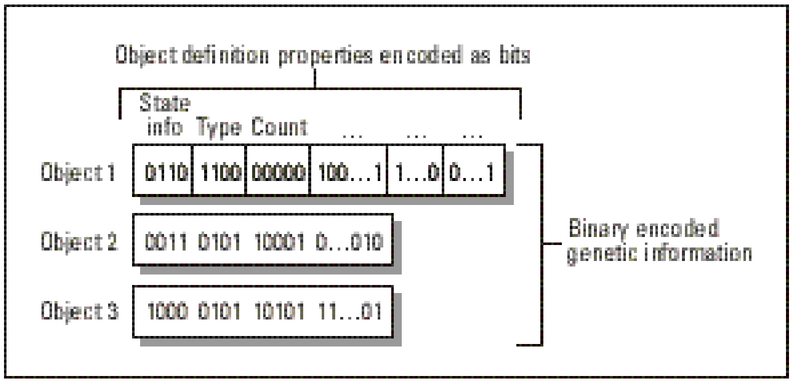


Figure 23-15: Binary encoding of genetic information.

You could just tweak from this point, knowing that you're on the right track. Or you could let genetic algorithms do the task for you. What you do is mix the two solutions or control vectors together to create two new offspring (as shown in Figure 23-16). To add a little bit of uncertainty, flip a bit here and there during the crossover process to simulate mutation. Then try your new solutions along with the last generation's best solutions and see what happens with the scores. Pick the best results out of the generation and do the process again. This is the process of g*enetic evolution*. Amazingly, the best possible solution will slowly evolve, and the result may be something you never imagined.

The key idea about genetic algorithms is that they try new ideas, and they can search a very large search space that normally you couldn't manually search one-by-one. This search space coverage property is due to the fact that mutations occur that represent completely random evolutionary events. These mutations may or may not be better adapted.

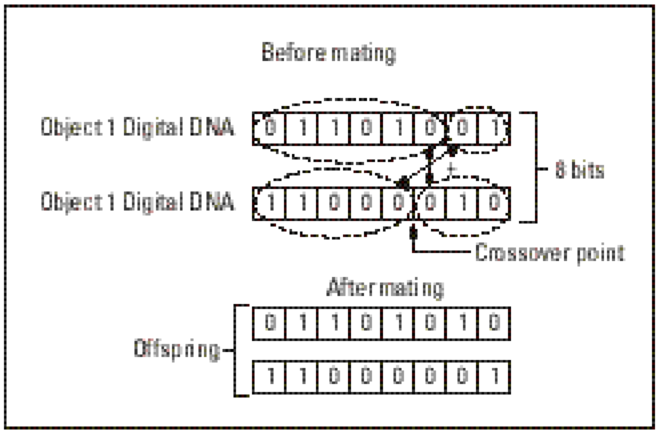


Figure 23-16: Digital reproduction.

So how do you use this information in a game? There are millions of ways I can think of off the top of my head, but I'm giving you just one to get you started. You can use the probability settings of your AI as the genetic source for digital DNA. Then you can merge and evolve the probabilities of the game creatures that have survived the longest, thus giving the best traits to future generations. Of course, you would only do this when you need to spawn a new creature, but you get the idea.

Fuzzy logic

Fuzzy logic is the last technology I'm going to cover and perhaps one of the most interesting. Fuzzy logic should really be referred to as fuzzy set theory. In other words, *fuzzy logic* is a method of analyzing sets of data so that the elements of the sets can have partial inclusion.

Most people are accustomed to *crisp logic,* in which something is either included or it isn't. For example, if I were to create the sets *child* and *adult,* I would fall into the adult category and my three-year-old nephew would be part of the child category.

Fuzzy logic, on the other hand, allows objects to be contained within a set even if they aren't totally in the set. For example, I may say that I am 10 percent part of the child set and 100 percent part of the adult set. Similarly, my nephew may be 2 percent part of the adult set and 100 percent part of the child set. These are fuzzy values. Also, you'll notice that they don't have to add up to 100 percent; they can be more or less.

The cool thing about fuzzy logic is that it enables you to make decisions that are based on fuzzy or error-ridden data, and the decisions are usually correct. With a crisp logic system, you can't do this. If you're missing a variable or input, then the analysis won't work. But a fuzzy system can still function and function well, just like a human brain.

I mean, how many decisions do you make each day that feel fuzzy to you? You don't have all the facts, but you're still fairly confident of the decision. This is fuzzy logic and its application to game AI is obvious in the areas of decision making, behavioral selections, and input/output filtering.

Bonus Chapter 24
Game Programming Potpourri

In This Chapter

▶ Choosing data structures

▶ Writing a good algorithm

▶ Understanding optimization theory

▶ Creating a demo

▶ Writing a save-game feature

▶ Implementing multiple players

In this chapter, I cover all those little details that slip through the cracks in any game programming book. I discuss everything from writing games so they can be saved, to making demos, to optimization theory! I think that this chapter will answer any further questions that you may have. If it doesn't, e-mail me at `necron@slip.net`, and I'll answer it!

Data Structures

Probably one of the most frequent questions I'm asked is this: What kind of data structures should be used in a game? My answer: Use the fastest, most efficient data structure possible for the task at hand. Note, however, that in most cases, the task at hand doesn't require the most advanced, complex data structures that computer science has to offer. Rather, try to keep things simple. When it comes to games in Windows, speed is more important than memory these days. So sacrifice memory before you sacrifice speed!

In the following sections, I cover some of the most common data structures used in games and give you some insight into when to use them.

Static structures and arrays

The most basic of all data structures is, of course, a single occurrence of a data item such as a single structure or class. For example:

```
typedef struct PLAYER_TYP // tag for forward references
        {
        int state; // state of player
        int x,y;   // position of player
        // more fields here...
        } PLAYER, *PLAYER_PTR;
```

In C++, you don't need to use `typedef` on structure definitions to create a type, as in C; a type is automatically created for you when you use the keyword `struct`.

```
PLAYER player_1, player_2; // create a couple of players
```

In this case, a single data structure along with two statically defined records does the job. On the other hand, if the game calls for three or more players, using an array like this is probably a good idea, because you can process all the players with a simple loop:

```
PLAYER players[20]; // the players of the game
```

Okay, great, but what if you don't know the number of players or records until the game runs? When this situation arises, I figure out the maximum number of elements that the array would have to hold in the most demanding case. If the number is less than or equal to 256 and each element is reasonably small (less than 256 bytes) then I usually statically allocate it and use a counter to count how many of the elements are active at any time.

You may think that this process is a waste of memory, and it is; but a preallocated array of a fixed size is easier and faster for the processor to traverse than a linked list or a more dynamic structure. My point: If you know the number of elements ahead of time and that number is small, go ahead and preallocate it or `malloc()` the memory at start up.

Don't get carried away with static arrays. Suppose that you have a 4K structure and you will need from 1 to 256 static records. Allocating 1MB of memory — in case the number *may* increase to 256 at some point — is a poor strategy.

Linked lists

Arrays are fine for simple data structures that can be precounted or estimated at compilation or start up, but data structures that can grow or shrink during run-time should use some form of *linked list*. Figure 24-1 depicts a standard abstract linked list. A linked list consists of a number of nodes, with each node containing information and a link to the next node in the list.

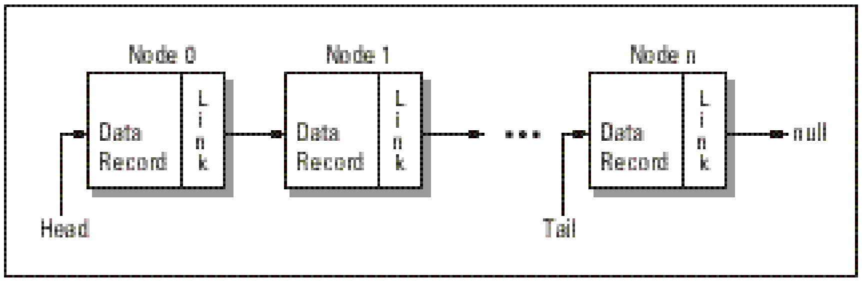


Figure 24-1: A linked list.

Linked lists are cool because you can insert or delete a node anywhere in the list (see Figure 24-2). The capability of a linked list to insert and delete nodes (and, therefore, information) during run-time makes them very attractive as a data structure for games.

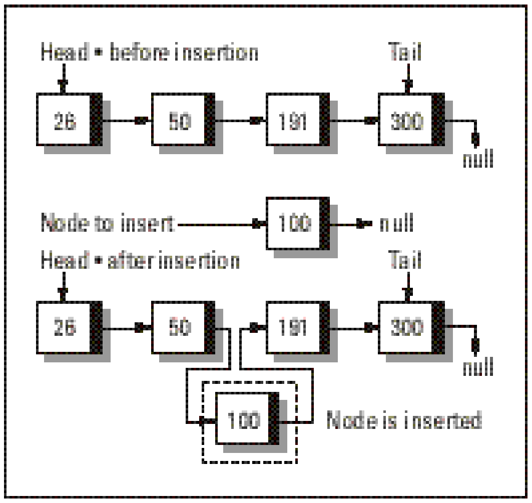


Figure 24-2: Inserting into a linked list.

The only bad thing about linked lists is that you must traverse them node-by-node to find what you are looking for. For example, suppose that you want the 15th element in an array; you can access it like this:

```
players[15]
```

But with linked lists, you need a *traversal algorithm* (which is a method to visit each node in the list) to traverse the list to find the 15th element. In the worst case, the searching of linked lists can take a number of iterations equal to the length of the list, represented mathematically as O(*n*) — read "big O of n." Of course, you can employ optimizations and secondary data structures to maintain a sorted indexed list that allows access almost as fast as the simple array.

Creating a linked list

For an example of a simple linked list, take a look at how to create a linked list, add a node, delete a node, and search for an item with a given key. Here's the basic node:

```
typedef struct NODE_TYP
   {
   int id;          // ID number of this object
   int x,y;         // position of object
   int color;       // color of object
   NODE_TYP *next; // this is the link to the next node
                    // more fields go here
   } NODE, *NODE_PTR;
```

Then to start the list off, you need a *head* pointer and a *tail* pointer that points to the head and tail of the list, respectively. However, because the list is empty, the pointers start off pointing to NULL.

```
NODE_PTR    head = NULL,
            tail = NULL;
```

Traversing a linked list

Ironically, traversing a linked list is the easiest of all operations. To traverse a linked list, follow these steps:

1. **Start at the head pointer.**
2. **Visit the node.**
3. **Link to the next node.**
4. **If the node is not NULL, then go to Step 2.**

And here’s the code:

```
void Traverse_List(NODE_PTR head)
{
// this function traverses the linked list and prints out
// each node

// first test whether head is null
if (head==NULL)
   {
   printf(“\nLinked List is empty!”);
   return;
   } // end if
// traverse while nodes
while (head!=NULL)
      {
      // visit the node, print it out, or whatever...
      printf(“\nNode Data: id=%d”, head->id);
      printf(“\nx=%d, y=%d”,head->x, head->y);
      printf(“\ncolor=%d\n”,head->color);
      // advance to next node (simple!)
      head = head->next;
      } // end while
} // end Traverse_List
```

Pretty cool, huh? In the next subsection, I explain how to add a node.

Adding a node (insertion)

The first step in adding a node is to create it. You can use either of two approaches:

- Send the new data elements to the insertion function and let it build up a new node.
- Build up a new node and then pass it to the insertion function.

Both methods achieve the same result. You can choose from a number of ways to insert a node into a linked list. The brute force method is to add it to the front or the end. This approach is fine if you don’t care about the order; but if you want the list to remain sorted, use a more intelligent insertion algorithm that maintains order in either ascending or descending order. This process makes searching much faster.

For simplicity’s sake, I took the easy way out and inserted at the end of the list, but inserting with sorting is not that much more complex. You first need to scan the list, find the location at which the new element should be inserted, and then insert the new element. Your only problem will be keeping track of the pointers and not losing any nodes or links.

Here's the code to insert a new node at the end of the list (a bit more difficult than the front of the list). Notice the special cases for empty lists and lists with a single element.

```
// access the global head and tail to make code easier;
// in real life, you may choose to use ** pointers and
// modify head and tail in the function
NODE_PTR Insert_Node(int id, int x, int y, int color)
{
// this function inserts a node at the end of the list
NODE_PTR new_node = NULL;

// Step 1: create the new node
new_node = malloc(sizeof(NODE)); // in C++ use new operator
// fill in fields
new_node->id = id;
new_node->x  = x;
new_node->y  = y;
new_node->color = color;
new_node->next = NULL; // good practice

// Step 2: find the current state of the linked list
if (head==NULL) // case 1
   {
   // finding an empty list means using the simplest case
   head = tail = new_node;
   // return new node
   return(new_node);
   } // end if
else
if ((head != NULL) && (head==tail)) // case 2
   {
   // you have exactly one element; this code is really
   // just a little finesse...
   head->next = new_node;
   tail = new_node;
   // return new node
   return(new_node);
   } // end if
else // case 3
   {
   // in case 2 or more elements are in list,
   // simply move to end of the list and add
   // the new node
```

```
    tail->next = new_node;
    tail = new_node;
    // return the new node
    return(new_node);
    } // end else
} // end Insert_Node
```

As you can see, the code is rather simple, but it is easy to mess up because you are dealing with pointers, so be careful! Also, the astute programmer very quickly realizes that, with a little thought, cases 2 and 3 can be combined; however, the preceding code is easier to follow than the code which combines cases 2 and 3.

Deleting a node

Deleting a node is the most complex of all linked-list operations, or at least up there in the record books.

The problem with deletion is that in most cases you want to delete a specific node. The node may be at the head, tail, or in the middle; therefore, you must write a very general algorithm that takes all these cases into consideration. If you're careful, deletion isn't a problem; but if you don't take all the cases into consideration and test them, you'll be sorry!

Now that you're scared of the linked-list police, here's the code to delete a node from a fictitious linked list using the `id` as the key:

```
// this function will modify the globals
// head and tail (possibly)
int Delete_Node(int id) // node to delete
{
// this function deletes a node from
// the linked list given its ID
NODE_PTR curr_ptr = head, // used to search the list
         prev_ptr = head; // previous record
// test whether a linked list to delete from is present
if (!head)
    return(-1);
// traverse the list and find node to delete
while(curr_ptr->id != id)
     {
     // save this position
     prev_ptr = curr_ptr;
     curr_ptr = curr_ptr->next;
     } // end while
```

(continued)

(continued)

```
// at this point we have found either the node
// or the end of the list
if (curr_ptr == NULL)
    return(-1); // couldn't find record

// the record was found, so delete it, but be careful;
// there are a number of cases to test for

// need to test cases
// case 1: one element
if (head==tail)
   {
   // delete node
   free(head);
   // fix up pointers
   head=tail=NULL;
   // return id of deleted node
   return(id);
   } // end if
else // case 2: front of list
if (curr_ptr == head)
   {
   // move head to next node
   head=head->link;
   // delete the node
   free(curr_node);
   // return id of deleted node
   return(id);
   } // end if
else // case 3: end of list
if (curr_ptr == tail)
   {
   // fix previous pointer to point to null
   prev_ptr = NULL;
   // delete the last node
   free(curr_ptr);
   // point tail to previous node
   tail = prev_ptr;
   // return id of deleted node
   return(id);
   } // end if
```

```
else // case 4: node is in middle of list
   {
   // connect the previous node to the next node
   prev_ptr->next = curr_ptr->next;
   // now delete the current node
   free(curr_ptr);
   // return id of deleted node
   return(id);
   } // end else
} // end Delete_Node
```

Note that the code contains a lot of special cases. Each is simple, but you have to think of every possible scenario — which I hope that I did!

Finally, you may have noticed the drama in the code when deleting nodes from the interior of the list. The problem occurs because, once a node is traversed, you can't get back to it. Therefore, I had to keep track of a previous NODE_PTR to keep track of the last node.

This problem can be solved along with others by using what is called a *double linked list* (as shown in Figure 24-3). The cool thing about a double linked list is that you can traverse in both directions from any point, and insertions and deletions are much easier. And the only change to the data structure is another link field, as shown (in bold) in the following code:

```
typedef struct NODE_TYP
   {
   int id;     // ID number of this object
   int x,y;    // position of object
   int color; // color of object
   NODE_TYP *next; // link to the next node
   NODE_TYP *prev; // link to previous node
   // more fields go here
   } NODE, *NODE_PTR;
```

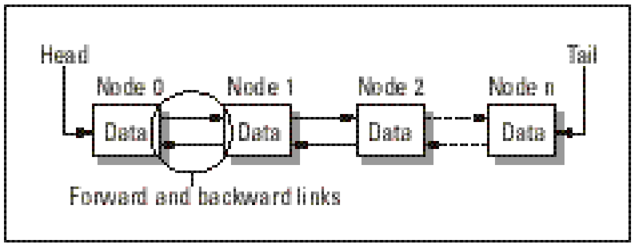


Figure 24-3: A double linked list.

Trees

The next class of advanced data structures are *trees*. Take a look at Figure 24-4 to see a number of different treelike data structures.

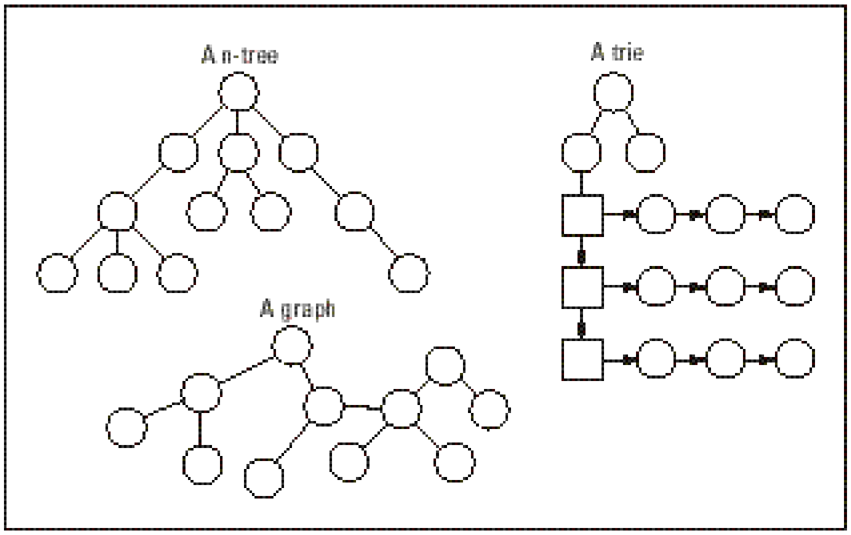


Figure 24-4: Some tree topologies.

Trees were invented to help with searching and storing large amounts of data. The most popular kind of tree is the *binary tree* or *B-tree*. The binary tree is a tree data structure emanating from a single root that is composed of a collection of nodes. Each node has one or two child nodes descending from it — hence, the term *binary*. Moreover, we talk of the *order* or number of levels of a tree, meaning how many layers (or levels) of nodes. For example, the tree in Figure 24-5 is a three-level tree.

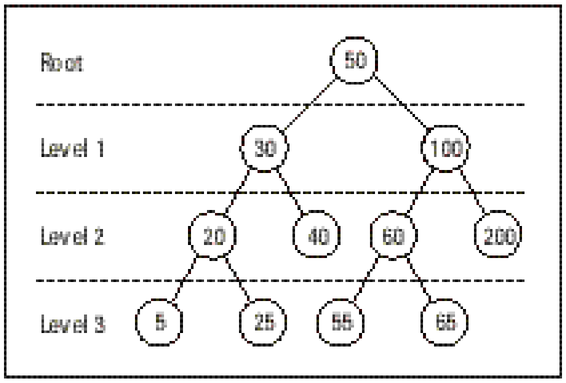


Figure 24-5: A three-level tree.

The interesting thing about trees is how fast the information can be searched. Most B-trees use a single search key to order the data in the tree. Then a searching algorithm searches the tree for the data.

For example, suppose that you want to create a B-tree that contains records of game objects, each with a number of properties. You can use the time of creation as the key. Here's the data structure that you would use to hold a single node:

```
typedef struct TNODE_TYP
   {
   int time;  // time of creation
   int x,y;    // position of object
   int color; // color of object
   NODE_TYP *right; // link to right node
   NODE_TYP *left;  // link to left node
   } TNODE, *TNODE_PTR;
```

Notice the similarity between the tree node and the linked-list node (covered in the earlier subsection "Linked lists"). The only difference is really the way you use the data structure and build up the tree.

Continuing with the example, suppose that I have five objects with the following creation times: t={0,25,3,12,10}. Figure 24-6 depicts two different B-trees that contain this data. However, a number of topologies exist that would maintain the properties of a B-tree.

In Figure 24-6, I use the convention that any right child is greater than or equal to its parent and any left child is less than its parent. You can use a different convention as long as you stick to it.

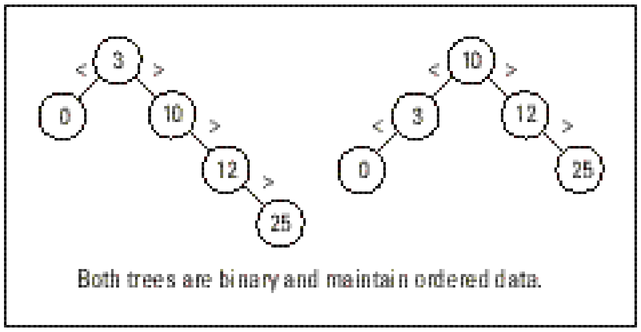


Figure 24-6: B-tree encoding of data set (0,25,3,12,10).

Unfortunately, I don't have time to cover the code for creating, searching, and working with B-trees, so you'll have to get a book or do some more research if you're interested (try *Programs and Data Structures In C,* by Leendert Ammeraal, Wiley Press). But I can tell you what B-trees bring to game programming.

Binary trees can hold enormous amounts of data, and that data can be quickly searched by using a binary search. This property is a manifestation of the binary structure of the tree. For example, if you have a tree with a million nodes, then at most it will take you 20 comparisons to find any desired record! Is that crazy or what? The reason for such a small number of comparisons is that at each iteration of your search (as you compare the key you are looking for against the current node you are visiting), you cut half the nodes out of the search space.

The above statement about search time is only true for balanced trees (trees that have an equal number of right and left children per level). If a tree is totally unbalanced, it degrades into a linked list and search time degrades into a linear function.

The next cool thing about B-trees is that if you take a branch (a subtree) and process it separately, the branch maintains the properties of a B-tree. Therefore, if you know where to look, you can search only the branch for whatever it is you're looking for.

When do you use B-trees? I suggest that you use treelike structures when the problem or data is treelike to begin with. If you find yourself drawing out the problem and you see branches to the left and right, then a tree is definitely for you. For example, in Bonus Chapter 23 on artificial intelligence, I speak of creating memories for the game characters. A tree structure would be perfect for memory. Each node could represent a room, and the children off of each node could represent the various objects that exist in each room.

Algorithmic Xtasy

Algorithm design and algorithmic analysis are complex subjects and usually are senior-level computer science material, but I can at least touch upon some common-sense techniques and ideas to help you out when you start writing more complex algorithms — because brute-force, sloppy programming just isn't good enough in many cases.

A good algorithm is better than all the assembly language or optimization in the world. For example, just by re-ordering your data, you can reduce the amount of time necessary to search for a data element by orders of magnitude. So the moral of the story is to select a good solid algorithm that fits the problem and the data, but at the same time to pick a data structure that can be accessed and manipulated with a good algorithm. I mean, if you always use linear arrays, you're never going to get better than linear search time (unless you use secondary data structures); but if you use sorted arrays, you can get logarithmic search time.

The first step to *writing* good algorithms is having some clue about how to *analyze* them. The art of analyzing algorithms is called *asymptotic analysis* and is usually calculus-based, so I'm just going to skim some of the concepts.

The basic idea of analyzing an algorithm is to compute how many times the main loop is executed for *n* elements, whatever *n* means. Of course, how long each execution takes plus the overhead of setup can also be important after you have a good algorithm, but the first place to start is the general counting of how many times. Take a look at two examples:

```
for (int index=0; index<n; index++)
    {
    // do work, 50 cycles
    } // end for index
```

In this case, the loop is going to execute for `n` iterations, thus the execution time is of the order *n*, or O(*n*). As explained in the earlier section called "Linked lists," Big O is a very rough upper estimate of execution time. You can be more precise in this case because you know that the inner computation takes 50 cycles; so the total execution time is:

```
n*50 cycles
```

Right? *Wrong!* If you are going to count cycles, then you had better count the cycles that it takes for the loop itself. This calculation consists of an initialization of a variable, a comparison, an increment, and a jump for each iteration. Adding in these factors, you end up with something like this:

$Cycles_{initialization}+(50+Cycles_{inc}+Cycles_{comp}+Cycles_{jump})*n$

This estimate is much more accurate. Of course, $Cycles_{inc}$, $Cycles_{comp}$, and $Cycles_{jump}$, are the number of cycles for the increment, comparison, and jump, respectively, and are each around 1 to 2 cycles on a Pentium-class processor. Therefore, in this case, the loop itself contributes just as much to the overall time of the inner loop as does the work performed by the loop!

Loop overhead is a key point. For example, many game programmers write a pixel-plotting function as a function instead of a macro or inline code. Because a pixel-plotting function is so simple, the call to the function takes more time than the pixel plotting! So make sure that you do enough work within your loop to warrant the usage of a loop in the first place. If the work within the loop "drowns" out the loop mechanics, then you should be okay.

The following code example has a much worse running time than *n*:

```
// outer loop
for (i=0; i<n; i++)
    {
    // inner loop
    for (j=1; j<2*n; j++)
        {
        // do work
        } // end for j
    } // end for i
```

In this code block, I'm assuming that the "work" part takes much more time than the actual code that supports the loop mechanics, so I'm not interested in the loop mechanics. What I am interested in is how many times this loop executes. The outer loop executes *n* times and the inner loop $2\times n-1$ times; thus the total amount of time the inner code will be executed is:

$$n\times(2\times n-1) = 2\times n^2-n$$

Look at these two terms for a moment. The $2\times n^2$ term is the dominant term and will drown out the *n* term as *n* gets larger (see Figure 24-7).

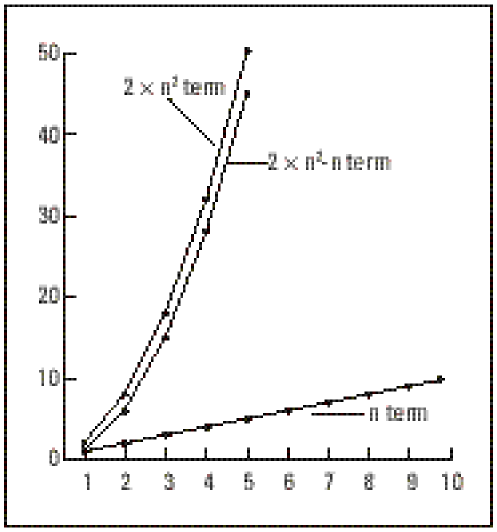


Figure 24-7: Rates of growth for the term of $2\times n^2n$.

For a small *n* — for example, when *n* equals 2 — the *n* term *is* relevant:

$$2\times(2)^2-2 = 6$$

In this case, the *n* term contributed to subtracting 25 percent of the total time away. But take a look at what happens when *n* gets larger; for example, when *n* equals 1,000.

$$2\times(1{,}000)^2-1{,}000 = 1{,}999{,}000$$

In this case, the n term contributes a decrease of only .05 percent; hardly important. Thus, you can see that the dominant term is indeed the $2\times n^2$ term, or more simply the n^2 itself. Therefore, this algorithm is $O(n^2)$. This result is very bad. Algorithms that run in n^2 time will just kill you — well, at least will kill the performance of your code — so if you come up with an algorithm like this, then try, try again!

That's it for asymptotic analysis; the bottom line is that you must be able to roughly estimate the run-time of your loops. This estimation will help you pick out the best algorithms and recode areas that need work.

Optimization Theory

No other programming has the kind of performance requirements that games do. Video games have always pushed the limits of hardware and software and will continue to do so. The reason for this: Enough is never enough. Game programmers always want to add one more creature, effect, or sound, as well as increase or improve the AI. Therefore, optimization is of the utmost importance. In this section, I cover some optimization techniques to get you started. If you are interested in reading more about this subject, a number of good books on the subject are available (try *Black Art of 3D Game Programming,* by André LaMothe; Waite Group Press).

Using your head

The first key to writing optimized code is understanding the compiler, data types, and the way your C/C++ is finally transformed into executable machine language. The best idea is to use simple programming and simple data structures. The more complex and contrived your code is, the more difficult time the compiler is going to have converting to machine code and, thus, the slower your code is going to execute (in most cases). Here are some basic rules to keep in mind:

- Use 32-bit wide data as much as possible; 8-bit data may take up less space, but Intel processors like 32-bit data are optimized to access it.
- Use inline functions for small functions that you call a lot.
- Use globals as much as possible without making ugly code.
- Avoid floating-point numbers for addition and subtraction.
- Use integers whenever possible, even though the floating point processor is almost as fast as the integer processor. Integers are exact, so if you don't need decimal accuracy, use integers.
- Align all data structures to 32-byte boundaries. You can do this manually or with compiler directives on most compilers.
- Never pass data to functions as *value* if the data is anything other than a simple type; always use a pointer.
- Don't use the `register` keyword in your code. Although Microsoft says that this keyword makes faster loops, it starves the compiler of registers and ends up making horrible code.
- If you're a C++ programmer, then it's okay for you to use classes and virtual functions; just don't go crazy with inheritance and layers of software.
- The Pentium-class processors use an internal data and code cache. Be aware of this arrangement and try to keep the size of your functions relatively small so they can fit into the cache (16K to 32K). In addition, when you store data, store it in the way it will be accessed. This method minimizes cache thrashing and main memory or secondary cache access, which is ten times slower than the internal cache.
- Be aware that Pentium-class processors have RISC-like cores, and they like simple instructions, allowing two or more instructions to execute in more than one execution unit. Don't write contrived code on a single line. Writing simpler code lines is better, even though you can mash the same functionality on the same line.

Working mathematical sorcery

Because a great deal of game programming is mathematical in nature, it pays to know advanced ways to perform math functions. You can use a number of general tricks and methods to enhance math performance and speed up operations.

The first I cover briefly is fixed-point math, which is an advanced subject, and I refer you to my other book, *The Black Art of 3D Game Programming* (published by Waite Group Press) for a more complete treatise on this topic. However, here is a list of math tricks you can use to speed up operations:

- With regard to data types, always use integers with integers and floats with floats. Conversion from one to another kills performance. Hence, hold off on the conversion of data types to the very last minute.
- Integers can be multiplied by any power of 2 by shifting to the left. And likewise, they can be divided by any power of 2 by shifting to the right. Multiplication and division other than by power of 2 is accomplished by using sums or subtractions of shifts. For example, 640 is not a power of two, but 512 and 128 are, so here's the best way in C code to multiply a number by 640 using shifts:

  ```
  product=(n<<7) + (n<<9); // n*128 + n*512 = n*640
  ```

- If you use matrix operations in your algorithms, then make sure that you take advantage of the sparseness of those operations.
- When you create constants, make sure that they have the proper casts, so that the compiler doesn't reduce them to integers or interpret them incorrectly. The best idea is to use the C++ `const` directive; for example:

  ```
  const float f=12.45;
  ```

- Avoid square roots, trigonometric functions, or any complex mathematical functions. In general, find a simpler way to accomplish the operation by taking advantage of certain assumptions or making approximations. However, you can always make a lookup table as shown in the section "Appreciating lookup tables."
- If you have to zero out a large array of floats, use a `memset()` like this:

  ```
  memset((void*)float_array,0,sizeof(float)*num_floats);
  ```

  However, you can only use `memset()` in this situation, because floats are encoded in IEEE format and the only value that is the same in both integer and float values is 0.
- When you perform mathematical calculations, see if you can reduce the expressions manually before coding them. For example, $n \times (f+1) \div$ is equivalent to $(f+1)$ because the multiplication and division of n cancel out.
- If you perform a complex mathematical operation and you need it again a few lines down in the code, then cache it; for example:

  ```
  // compute term that is used in more
  // than one expression
  float n_squared = n*n;

  // use term in two different expressions
  pitch = 34.5*n_squared+100*rate;
  magnitude = n*squared / length;
  ```

- And last, but not least, make sure that you set the compiler options to use the floating point processor and create code that is *fast* (runs the quickest) rather than *small* (takes up the least amount of RAM).

Unrolling the loop

The next optimization trick is *loop unrolling,* which was one of the best optimizations possible back in the 8- and 16-bit days, but today it can backfire on you.

Unrolling the loop means to take apart a loop which iterates some number of times and to manually code each line as the loop would have mechanically. Here’s an example:

```
// loop before unrolling
for (int index=0; index<8; index++)
    {
    // do work
    sum+=data[index];
    } // end for index
```

The problem with this loop is that the “work” section takes less time than the loop does for the increment, comparison, and jump. Hence, the loop code itself doubles or triples the amount of time the code requires!

To fix this problem with the code, unroll the loop like this:

```
// the unrolled version
sum+=data[0];
sum+=data[1];
sum+=data[2];
sum+=data[3];
sum+=data[4];
sum+=data[5];
sum+=data[6];
sum+=data[7];
```

This approach is much better.

However, consider these two caveats to the code listed above:

- If the loop body is much more complex than the loop mechanics itself, then you really don’t need to unroll it. For example, if you are computing square roots in the “work” section of the loop, then a few more cycles in each iteration isn’t going to help you.

- Pentium processors have internal caches, and unrolling a loop too much may cause it to be unable to fit in the internal cache. This situation is disastrous and will bring your code to a halt. I suggest unrolling (if appropriate) 8 to 32 times, depending on the situation.

Appreciating lookup tables

This is my personal favorite optimization. *Lookup tables* are precomputed values of some computation that you know you will perform during runtime. You simply compute all possible values at startup and then run the code.

For example, suppose that you need the sine and cosine of the angles from 0 to 359 degrees. Computing them by using `sin()` and `cos()` would kill your math performance if you use the floating point processor; but by utilizing a lookup table, your code can compute `sin()` or `cos()` in a few cycles because the process involves just grabbing the number from a lookup table. Here’s an example:

```
// storage for look up tables
float SIN_LOOK[360];
float COS_LOOK[360];
// create lookup table
for (int angle=0; angle < 360; angle++)
    {
    // convert angle to radians because the math library
    // uses rads instead of degrees
    // remember that 2*pi rads are in 360 degrees
    float rad_angle = angle * (3.14159/180);

    // fill in the remaining entries in lookup tables
    SIN_LOOK[angle] = sin(rad_angle);
    COS_LOOK[angle] = cos(rad_angle);
    } // end for angle
```

As an example of using the lookup table, here’s the code to draw a circle of radius 10:

```
for (int ang = 0; ang<360; ang++)
    {
    // compute the next point on circle
    x_pos = 10*COS_LOOK[angle];
    y_pos = 10*SIN_LOOK[angle];
    // plot the pixel
    Plot_Pixel((int)x_pos+x0, (int)y_pos+y0, color);
    } // end for ang
```

Of course, lookup tables take up memory, but they are well worth it. If you can precompute a set of values that you'll need in your code, then put the set in a lookup table. That's my motto. (And if you have a hard time believing that the really cool and complex games today don't use lookup tables, think again; how do you think that *Doom* and *Quake* work?)

Using assembly language

The final optimization I want to talk about is using *assembly language*.

So you have the killer algorithm and all your data structures are good, but you just want a little bit more *oomph* to your code's speed. Hand-crafted code written in assembly language doesn't make code go 1,000 times faster with 32-bit processors like it did with 8- and 16-bit processors, but it can get you 2 to 10 times more speed, and that result is definitely worth it.

However, make sure that you only try to convert sections of your game that need converting. Don't mess with converting the menu program to assembly, because that's a waste of time. Use a profiler or similar analysis program to see where all your game's CPU cycles are being eaten up (probably in the graphics sections) and then target those for conversion to assembly language.

In the old days (a few years ago), most compilers didn't have inline assemblers, and if they did, the inline assemblers were awful and supported very few features of an external assembler. Today, the inline assemblers that come with Microsoft, Borland, or Watcom compilers are really good and just about as full featured as a standalone assembler for small jobs that range from a few dozen lines to a couple hundred. Therefore, I suggest using the inline assembler in your compiler if you want to do any assembly language.

Here's how you invoke the inline assembler in Microsoft Visual C++ 2.0+:

```
_asm
{
[assembly language code here]
} // end asm
```

The cool thing about the inline assembler is that it enables you to use variable names that have been defined by C/C++. For example, here's how to write a 32-bit memory fill function using inline assembly language:

```
void qmemset(void *memory, int value, int num_quads)
{
// this function uses 32-bit assembly language based
// on the string instructions to fill a region of memory
_asm
   {
   CLD                // clear the direction flag
   MOV EDI, memory    // move pointer into EDI
   MOV ECX, num_quads // ECX hold loop count
   MOV EAX, value     // EAX hold value
   REP STOSD          // perform fill
   } // end asm
} // end qmemset
```

To use the new function, all you do is this:

```
qmemset(&buffer, 25, 1000);
```

And 1,000 quads would be filled with the value 25 starting at the address of `buffer`.

If you're not using Microsoft Visual C++, then take a look at your particular compiler's Help file to see the exact syntax needed for inline assembly. In most cases, the changes to the prior code block are an underscore here and there and nothing more.

Making Demos

So you've got this killer game and you need a demo mode. You can use two main methods to implement a demo mode:

- Play the game yourself, record your own moves, and then play the moves back.
- Use an AI player that plays the game unattended.

Recording game play turns out to be the most common choice, because writing an AI player that can play as well as a human is difficult. In addition, it's difficult to let the AI demo player know that it needs to make a good impression on potential buyers by playing the game in a "cool" way. The next sections take a brief look at how each of these methods are implemented.

Prerecorded

To record a demo, follow these steps:

1. **Record the state of all the input devices each cycle as you create the demo.**
2. **Write the data to a file.**
3. **Play back the demo as if it were the input of the game.**

Take a look at Figure 24-8 to see this point graphically. The idea is to create your demo so that the game doesn't know whether the input is from the keyboard (input device) or from a file, so it simply plays the game back.

For this process to work, you need to have a *deterministic* game. This term means that if you play the game again and do the exact same moves, then the game creatures will also respond the same way. As well as recording the input devices, you must record the initial random-number seed as well, so that the starting state of a game is recorded as well as the input. This step ensures that the game will play back in the exact same way as you recorded it.

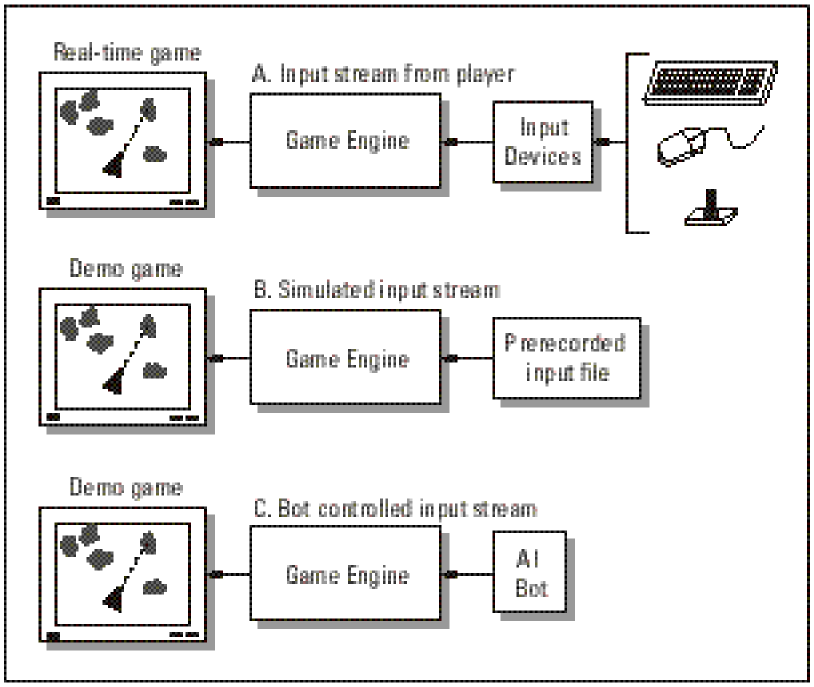


Figure 24-8: Demo playback.

To record a game, the best approach is to not sample the input at time intervals, but to sample the input at each frame. Therefore, if the game is played on a slower or faster computer, the playback data won't get out of synchronization with the game. Here are the steps your code should follow:

1. **Create a general input record.**
2. **Merge all the input devices into the single record each cycle.**
3. **As the game runs, write each input record to a file (one for each frame).**

Also, at the beginning of the file, I place any state information or random numbers that I played the demo with, so that these values can be loaded back in.

For example, the playback file may look something like this:

```
Initial State Information
Frame 1: Input Values
Frame 2: Input Values
Frame 3: Input Values . . .
Frame N: Input Values
```

After you have the file, you reset the game and simply start it up. Then you read the file as if it were the input devices. The game doesn't know the difference and simply plays!

The single mistake that you can make in creating the demo is sampling the input at the wrong time when you write records. Make absolutely certain that the input you sample and record is the actual input that the game uses for that frame. A common mistake newbies make is to sample the input for the demo mode at a point in the event loop before or after the normal input is read. Hence, you are sampling different data! It's possible that the play may have the fire button down in one part of the event loop and not in another; thus you must sample at the same point you normally read the input for the game.

AI controlled

The second method of recording a game is by writing an AI bot that plays, much like people do for Internet games such as *Quake*. The bot plays the game while in demo mode as if it were one of the AI characters in the game. The only problem (other than the technical complexity) is that the bot may not necessarily show off all the cool rooms, weapons, and so on, because it doesn't know that it's making a demo. On the other hand, the cool thing about having a bot play is that each demo is different and the attract mode of the game will never get boring.

Implementing a bot to play your game is like using any other AI character: You connect it to the input port of your game and override the normal input stream (refer to Figure 24-8). Then you write the AI algorithms for the bot and give it some main goals, such as finding its way out of the maze or killing everything in sight. Finally, you simply let the bot loose to demo until the player wants to play.

Saving the Game

One of the biggest pains in the butt is writing a save-game feature. This task is one that all game programmers do last and do by the seat of their pants, in most cases. The key is to write your game with the idea that you want to give the player a save-game option at some point, so that you don't dig yourself into a corner.

To save a game at any point in the game means to record the state of every single variable in the game and the state of every single object in the game. Therefore, you must record in a file all global variables along with the state of every single object.

The best way to approach this task is by adopting an object-oriented thought process. Instead of writing a function that writes out the state of each object and all the global variables, teach each object how to write and read its own state to a disk file.

Then to save a game, all you need to do is write the globals and create a simple function that requests each game object to write its own state. To load the game back in, all you need to do is read the globals back into the system and load the state of all the objects back into the game.

This way, if you add another object or object type, the loading/saving process is localized in the object itself, rather than strewn about all over the place in your code.

Implementing Multiple Players

The last little tidbit of game programming legerdemain is implementing multiple players. Of course, if you want to implement a networked game, that's a whole other story, but DirectPlay makes the communication part easy at least. However, if all you want to do is let two or more players play your game at the same time or by taking turns, then that flexibility requires nothing more than extra data structures and a bit of housekeeping.

Taking turns

Implementing turn-taking is simple and difficult at the same time. The task is simple because if you can implement one player, then implementing two or more is nothing more than having more than one player record. But the task is difficult because you must save the game for each player when switching players. Hence, you need to implement a save-game option if you want to allow for turn-taking. Obviously, the players shouldn't know that the game is being saved as they take turns, but that's what's really going on.

Here's a list of the steps to allow two players to play, one after the other:

1. **Start game; player 1 begins.**
2. **Player 1 plays until she dies.**
3. **The state of player 1's game is saved, and player 2 begins.**
4. **Player 2 plays until he dies.**
5. **The state of player 2's game is saved.**

   Here comes the transition.
6. **The previously saved game of player 1 is reloaded and player 1 continues.**
7. **Go back to Step 2.**

As you can see, Step 5 is where the action starts happening and the game starts pinging back and forth between players. And if you want more than two players, you simply play them one at a time until you're at the end of the list and then you start over.

Appearing on-screen at the same time

Playing two or more players on the same screen is a little more difficult than swapping, because you have to write the game a little more generally as far as game play, collision, and interaction between the players goes. Moreover, now that two or more players are on the screen at the same time, you must allocate a specific input device for each player. This device is usually a joystick for each player, or maybe one player uses the keyboard and one uses the joystick.

The other problem with putting two or more players on the screen at the same time is that some games just don't work well with two players at the same time. For example, if the game is a scrolling game, one player may want to go one way while the other wants to go another way. This dilemma can cause a conflict, and you'll have to think about it as you program. Thus, the best games for implementing more than one player are games that are single-screen, such as fighting games or other games in which the players stay relatively near each other.

If you want to allow the players to roam around freely, you can always generate more than one view — create a split-screen display (as shown in Figure 24-9). The only problem with a split-screen display is the split-screen display! You must generate two or more views of the game. This step can be technically challenging, moreover, because the players may not be able to see what's going on if the screen is too small to accommodate two views. The bottom line is this: If you can pull it off, then it's a cool option.

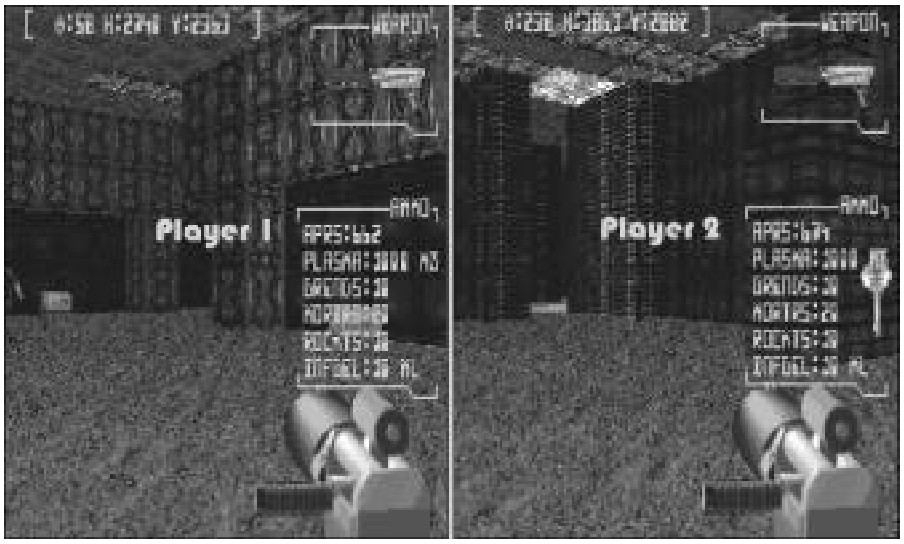


Figure 24-9: Split-screen game display.