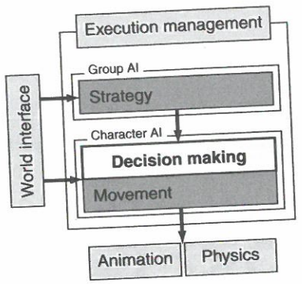


Figure 5.1 The AI model

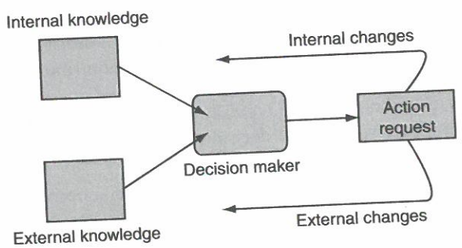


Figure 5.2 Decision making schematic

and the output is an action request. The knowledge can be further broken down into external and internal knowledge. External knowledge is the information that a character knows about the game environment around it: the position of other characters, the layout of the level, whether a switch has been thrown, the direction that a noise is coming from, and so on. Internal knowledge is information about the character's internal state or thought processes: its health, its ultimate goals, what it was doing a couple of seconds ago, and so on.

Typically, the same external knowledge can drive any of the algorithms in this chapter, whereas the algorithms themselves control what kinds of internal knowledge can be used (although they don't constrain what that knowledge represents, in game terms).

Actions, correspondingly, can have two components: they can request an action that will change the external state of the character (such as throwing a switch, firing a weapon, moving into a room) or actions that only affect the internal state (see Figure 5.2). Changes to the internal state

are less obvious in game applications but are significant in some decision making algorithms. They might correspond to changing the character's opinion of the player, changing its emotional state, or changing its ultimate goal. Again, algorithms will typically have the internal actions as part of their makeup, while external actions can be generated in a form that is identical for each algorithm.

The format and quantity of the knowledge depend on the requirements of the game. Knowledge representation is intrinsically linked with most decision making algorithms. It is difficult to be completely general with knowledge representation, although we will consider some widely applicable mechanisms in Chapter 11.

Actions, on the other hand, can be treated more consistently. We'll return to the problem of representing and executing actions at the end of this chapter.

5.2 Decision Trees

Decision trees are fast, easily implemented, and simple to understand. They are the simplest decision making technique that we'll look at, although extensions to the basic algorithm can make them quite sophisticated. They are used extensively to control characters and for other in-game decision making, such as animation control.

They have the advantage of being very modular and easy to create. We've seen them used for everything from animation to complex strategic and tactical AI.

Although it is rare in current games, decision trees can also be learned, and the learned tree is relatively easy to understand (compared to, for example, the weights of a neural network). We'll come back to this topic later in Chapter 7.

5.2.1 The Problem

Given a set of knowledge, we need to generate a corresponding action from a set of possible actions.

The mapping between input and output may be quite complex. The same action will be used for many different sets of input, but any small change in one input value might make the difference between an action being sensible and an action appearing stupid.

We need a method that can easily group lots of inputs together under one action, while allowing the input values that are significant to control the output.

5.2.2 The Algorithm

A decision tree is made up of connected decision points. The tree has a starting decision, its root. For each decision, starting from the root, one of a set of ongoing options is chosen.

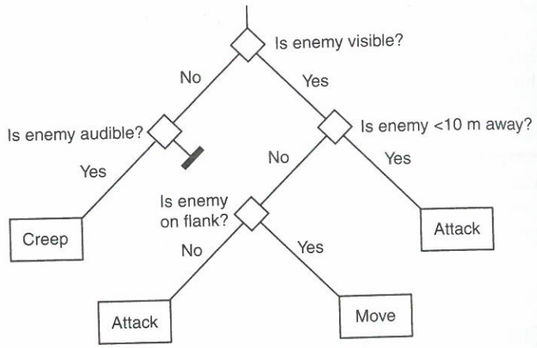


Figure 5.3 A decision tree

Each choice is made based on the character's knowledge. Because decision trees are often used as simple and fast decision mechanisms, characters usually refer directly to the global game state rather than have a representation of what they personally know.

The algorithm continues along the tree, making choices at each decision node until the decision process has no more decisions to consider. At each leaf of the tree an action is attached. When the decision algorithm arrives at an action, that action is carried out immediately.

Most decision treenodes make very simple decisions, typically with only two possible responses. In Figure 5.3 the decisions relate to the position of an enemy.

Notice that one action can be placed at the end of multiple branches. In Figure 5.3 the character will choose to attack unless it can't see the enemy or is flanked. The attack action is present at two leaves.

Figure 5.4 shows the same decision tree with a decision having been made. The path taken by the algorithm is highlighted, showing the arrival at a single action, which may then be executed by the character.

Decisions

Decisions in a tree are simple. They typically check a single value and don't contain any Boolean logic (i.e., they don't join tests together with AND or OR).

Depending on the implementation and the data types of the values stored in the character's knowledge, different kinds of tests may be possible. A representative set is given in the following table, based on a game engine we've worked on:

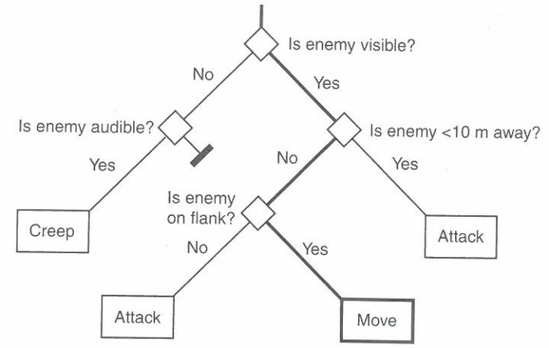


Figure 5.4 The decision tree with a decision made

Data Type	Decisions
Boolean	Value is true
Enumeration (i.e., a set of values, only one of which might be allowable)	Matches one of a given set of values
Numeric value (either integer or floating point)	Value is within a given range
3D Vector	Vector has a length within a given range (this can be used to check the distance between the character and an enemy, for example)

In addition to primitive types, in object-oriented game engines it is common to allow the decision tree to access methods of instances. This allows the decision tree to delegate more complex processing to optimized and compiled code, while still applying the simple decisions in the previous table to the return value.

Combinations of Decisions

The decision tree is efficient because the decisions are typically very simple. Each decision makes only one test. When Boolean combinations of tests are required, the tree structure represents this.

To AND two decisions together, they are placed in series in the tree. The first part of Figure 5.5 illustrates a tree with two decisions, both of which need to be true in order for action 1 to be carried out. This tree has the logic "if A AND B, then carry out action 1, otherwise carry out action 2."

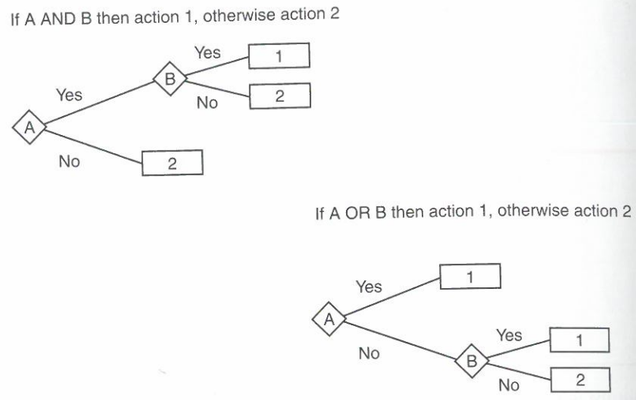


Figure 5.5 Trees representing AND and OR

To OR two decisions together, we also use the decisions in series, but with the two actions swapped over from the AND example above. The second part of Figure 5.5 illustrates this. If either test returns true, then action 1 is carried out. Only if neither test passes is action 2 run. This tree has the logic "if A OR B, then carry out action 1, otherwise carry out action 2."

This ability for simple decision trees to build up any logical combination of tests is used in other decision making systems. We'll see it again in the Rete algorithm in Section 5.8 on rule-based systems.

Decision Complexity

Because decisions are built into a tree, the number of decisions that need to be considered is usually much smaller than the number of decisions in the tree. Figure 5.6 shows a decision tree with 15 different decisions and 16 possible actions. After the algorithm is run, we see that only four decisions are ever considered.

Decision trees are relatively simple to build and can be built in stages. A simple tree can be implemented initially, and then as the AI is tested in the game, additional decisions can be added to trap special cases or add new behaviors.

Branching

In the examples so far, and in most of the rest of the chapter, decisions will choose between two options. This is called a binary decision tree. There is no reason why you can't build your decision

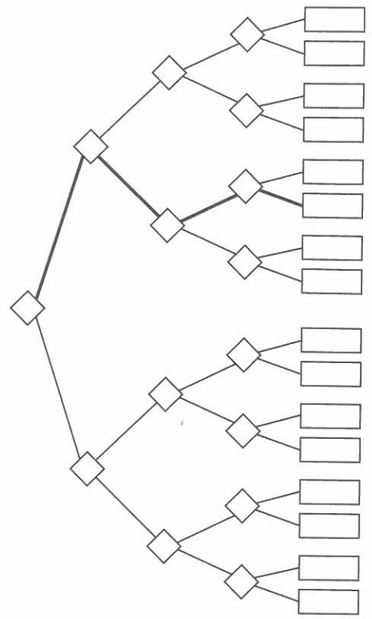


Figure 5.6 Wide decision tree with decision

tree so that decisions can have any number of options. You can also have different decisions with different numbers of branches.

Imagine having a guard character in a military facility. The guard needs to make a decision based on the current alert status of the base. This alert status might be one of a set of states: "green," "yellow," "red," or "black," for example. Using the simple binary decision making tree described above, we'd have to build the tree in Figure 5.7 to make a decision.

The same value (the alert state) may be checked three times. This won't be as much of a problem if we order the checks so the most likely states come first. Even so, the decision tree may have to do the same work several times to make a decision.

We could allow our decision tree to have several branches at each decision point. With four branches, the same decision tree now looks like Figure 5.8.

This structure is flatter, only ever requires one decision, and is obviously more efficient.

Despite the obvious advantages, it is more common to see decision trees using only binary decisions. First, this is because the underlying code for multiple branches usually simplifies

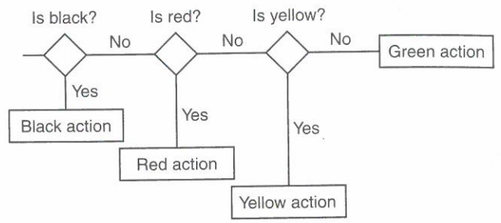


Figure 5.7 Deep binary decision tree

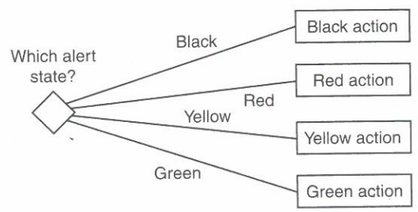


Figure 5.8 Flat decision tree with four branches

down to a series of binary tests (`if` statements in C/C++, for example). Although the decision tree is simpler with multiple branches, the implementation speed is usually not significantly different.

Second, decision trees are typically binary because they can be more easily optimized. In addition, some learning algorithms that work with decision trees require them to be binary.

You can do anything with a binary tree that you can do with a more complex tree, so it has become traditional to stick with two branches per decision. Most, although not all, of the decision tree systems we've worked with have used binary decisions. We think it is a matter of implementation preference. Do you want the extra programming work and reduction in flexibility for the sake of a marginal speed up?

5.2.3 Pseudo-Code

A decision tree takes as input a tree definition, consisting of decision tree nodes. Decision tree nodes might be decisions or actions. In an object-oriented language, these may be sub-classes of

the tree node class. The base class specifies a method used to perform the decision tree algorithm. It is not defined in the base class (i.e., it is a pure virtual function):

```
class DecisionTreeNode:
  def makeDecision() # Recursively walks through the tree
```

Actions simply contain details of the action to run if the tree arrives there. Their structure depends on the action information needed by the game (see Section 5.11 later in the chapter on the structure of actions). Their `makeDecision` function simply returns the action (we'll see how this is used in a moment):

```
class Action:
  def makeDecision():
    return this
```

Decisions have the following format:

```
class Decision (DecisionTreeNode):
  trueNode
  falseNode
  testValue
  def getBranch() # carries out the test
  def makeDecision() # Recursively walks through the tree
```

where the `trueNode` and `falseNode` members are pointers to other nodes in the tree, and the `testValue` member points to the piece of data in the character's knowledge which will form the basis of the test. The `getBranch` function carries out the test and returns which branch to follow. Often, there are different forms of the decision node structure for different types of tests (i.e., for different data types). For example, a decision for floating point values might look like the following:

```
class FloatDecision (Decision):
  minValue
  maxValue

  def getBranch():
    if maxValue >= testValue >= minValue:
      return trueNode
    else:
      return falseNode
```

A decision tree can be referred to by its root node: the first decision it makes. A decision tree with no decisions might have an action as its root. This can be useful for prototyping a character's AI, by forcing a particular action to always be returned from its decision tree.

The decision tree algorithm is recursively performed by the `makeDecision` method. It can be trivially expressed as:

```
class Decision:

  def makeDecision():

    # Make the decision and recurse based on the result
    branch = getBranch()
    return branch.makeDecision()
```

The `makeDecision` function is called initially on the root node of the decision tree.

Multiple Branches

We can implement a decision that supports multiple branches almost as simply. Its general form is:

```
class MultiDecision (DecisionTreeNode):
  daughterNodes
  testValue

  # Carries out the test and returns the node to follow
  def getBranch():
    return daughterNodes[testValue]

  # Recursively runs the algorithm, exactly as before
  def makeDecision():
    branch = getBranch()
    return branch.makeDecision()
```

where `daughterNodes` is a mapping between possible values of the `testValue` and branches of the tree. This can be implemented as a hash table, or for a numeric test value it might be an array of daughter nodes that can be searched using a binary search algorithm.

5.2.4 On the Website

To see the decision tree in action, run the Decision Tree program that is available on the website. It is a command line program designed to let you see behind the scenes of a decision making process.

Each decision in the tree is presented to you as a true or false option, so you are making the decision, rather than the software. The output clearly shows how each decision is considered in turn until a final output action is available.

5.2.5 Knowledge Representation

Decision trees work directly with primitive data types. Decisions can be based on integers, floating point numbers, Booleans, or any other kind of game-specific data. One of the benefits of decision trees is that they require no translation of knowledge from the format used by the rest of the game.

Correspondingly, decision trees are most commonly implemented so they access the state of the game directly. If a decision tree needs to know how far the player is from an enemy, then it will most likely access the player and enemy's position directly.

This lack of translation can cause difficult-to-find bugs. If a decision in the tree is very rarely used, then it may not be obvious if it is broken. During development, the structure of the game state regularly changes, and this can break decisions that rely on a particular structure or implementation. A decision might detect, for example, which direction a security camera is pointing. If the underlying implementation changes from a simple angle to a full quaternion to represent the camera rotation, then the decision will break.

To avoid this situation, some developers choose to insulate all access to the state of the game. The techniques described in Chapter 10 on world interfacing provide this level of protection.

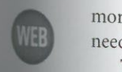
5.2.6 Implementation Nodes

The function above relies on being able to tell whether a node is an action or a decision and being able to call the test function on the decision and have it carry out the correct test logic (i.e., in object-oriented programming terms, the test function must be polymorphic).

Both are simple to implement using object-oriented languages with runtime-type information (i.e., we can detect which class an instance belongs to at runtime).

Most games written in C++ switch off RTTI (runtime-type information) for speed reasons. In this case the "is instance of" test must be made using identification codes embedded into each class or another manual method.

Similarly, many developers avoid using virtual functions (the C++ implementation of polymorphism). In this case, some manual mechanism is needed to detect which kind of decision is needed and to call the appropriate test code.



Library

The implementation on the website demonstrates both these techniques. It uses neither RTTI nor virtual functions, but relies on a numeric code embedded in each class.

The implementation also stores nodes in a single block of memory. This avoids problems with different nodes being stored in different places, which causes memory cache problems and slower execution.

5.2.7 Performance of Decision Trees

You can see from the pseudo-code that the algorithm is very simple. It takes no memory, and its performance is linear with the number of nodes visited.

If we assume that each decision takes a constant amount of time and that the tree is balanced (see the next section for more details), then the performance of the algorithm is $O(\log_2 n)$, where n is the number of decision nodes in the tree.

It is very common for the decisions to take constant time. The example decisions we gave in the table at the start of the section are all constant time processes. There are some decisions that take more time, however. A decision that checks if any enemy is visible, for example, may involve complex ray casting sight checks through the level geometry. If this decision is placed in a decision tree, then the execution time of the decision tree will be swamped by the execution time of this one decision.

5.2.8 Balancing the Tree

Decision trees are intended to run fast and are fastest when the tree is balanced. A balanced tree has about the same number of leaves on each branch. Compare the decision trees in Figure 5.9. The second is balanced (same number of behaviors in each branch), while the first is extremely unbalanced. Both have 8 behaviors and 7 decisions.

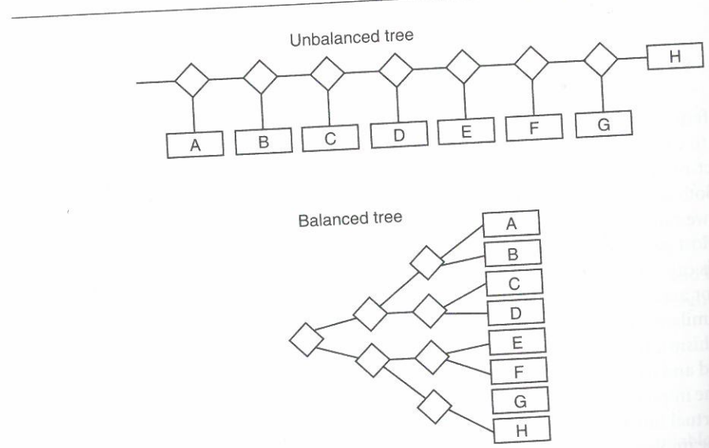


Figure 5.9 Balanced and unbalanced trees

To get to behavior H, the first tree needs 8 decisions, whereas the second tree only needs 3. In fact, if all behaviors were equally likely, then the first tree would need an average of nearly $4\frac{1}{2}$ decisions, whereas the second tree would always only need 3.

At its worst, with a severely unbalanced tree, the decision tree algorithm goes from being $O(\log_2 n)$ to $O(n)$. Clearly, we'd like to make sure we stay as balanced as possible, with the same number of leaves resulting from each decision.

Although a balanced tree is theoretically optimal, in practice the fastest tree structure is slightly more complex.

In reality, the different results of a decision are not equally likely. Consider the example trees in Figure 5.9 again. If we were likely to end up in behavior A the majority of the time, then the first tree would be more efficient; it gets to A in one step. The second tree takes 3 decisions to arrive at A.

Not all decisions are equal. A decision that is very time consuming to run (such as one that searches for the distance to the nearest enemy) should only be taken if absolutely necessary. Having this further down the tree, even at the expense of having an unbalanced tree, is a good idea.

Structuring the tree for maximum performance is a black art. Since decision trees are very fast anyway, it is rarely important to squeeze out every ounce of speed. Use these general guidelines: balance the tree, but make commonly used branches shorter than rarely used ones and put the most expensive decisions late.

5.2.9 Beyond the Tree

So far we have kept a strict branching pattern for our tree. We can extend the tree to allow multiple branches to merge into a new decision. Figure 5.10 shows an example of this.

The algorithm we developed earlier will support this kind of tree without modification. It is simply a matter of assigning the same decision to more than one `trueNode` or `falseNode` in the tree. It can then be reached in more than one way. This is just the same as assigning a single action to more than one leaf.

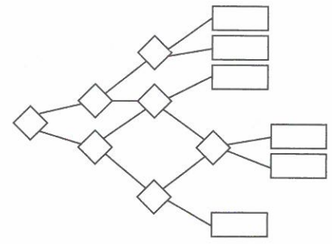


Figure 5.10 Merging branches

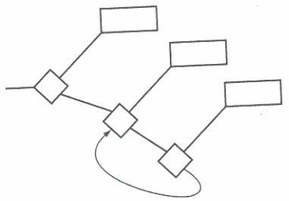


Figure 5.11 Pathological tree

You need to take care not to introduce possible loops in the tree. In Figure 5.11, the third decision in the tree has a `falseNode` earlier in the tree. The decision process can loop around forever, never finding a leaf.

Strictly, the valid decision structure is called a *directed acyclic graph* (DAG). In the context of this algorithm, it still is always called a decision tree.

5.2.10 Random Decision Trees

Often, we don't want the choice of behavior to be completely predictable. Some element of random behavior choice adds unpredictability, interest, and variation.

It is simple to add a decision into the decision tree that has a random element. We could generate a random number, for example, and choose a branch based on its value.

Because decision trees are intended to run frequently, reacting to the immediate state of the world, random decisions cause problems. Imagine running the tree in Figure 5.12 for every frame.

As long as the agent isn't under attack, the stand still and patrol behaviors will be chosen at random. This choice is made at every frame, so the character will appear to vacillate between standing and moving. This is likely to appear odd and unacceptable to the player.

To introduce random choices in the decision tree, the decision making process needs to become stable—if there is no relevant change in world state, there should be no change in decision. Note that this isn't the same as saying the agent should make the same decision every time for a particular world state. Faced with the same state at very different times, it can make different decisions, but at consecutive frames it should stay with one decision.

In the previous tree, every time the agent is not under attack it can stand still or patrol. We don't care which it does, but once it has chosen, it should continue doing that.

This is achieved by allowing the random decision to keep track of what it did last time. When the decision is first considered, a choice is made at random, and that choice is stored. The next time the decision is considered, there is no randomness, and the previous choice is automatically taken.

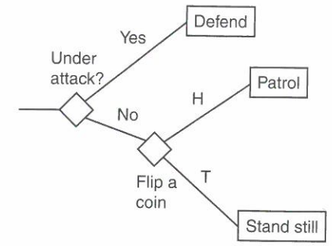


Figure 5.12 Random tree

If the decision tree is run again, and the same decision is not considered, it means that some other decision went a different way—something in the world must have changed. In this case we need to get rid of the choice we made.

Pseudo-Code

This is the pseudo-code for a random binary decision:

```
struct RandomDecision (Decision):
  lastFrame = -1
  lastDecision = false

  def test():
    # check if our stored decision is too old
    if frame() > lastFrame + 1:
      # Make a new decision and store it
      lastDecision = randomBoolean()

    # Either way we need to update the frame value
    lastFrame = frame()

    # We return the stored value
    return lastDecision
```

To avoid having to go through each unused decision and remove its previous value, we store the frame number at which a stored decision is made. If the test method is called, and the previous stored value was stored on the previous frame, we use it. If it was stored prior to that, then we create a new value.

This code relies on two functions:

- `frame()` returns the number of the current frame. This should increment by one each frame. If the decision tree isn't called every frame, then `frame` should be replaced by a function that increments each time the decision tree is called.
- `randomBoolean()` returns a random Boolean value, either true or false.

This algorithm for a random decision can be used with the decision tree algorithm provided above.

Timing Out

If the agent continues to do the same thing forever, it may look strange. The decision tree in our example above, for example, could leave the agent standing still forever, as long as we never attack.

Random decisions that are stored can be set with time-out information, so the agent changes behavior occasionally.

The pseudo-code for the decision now looks like the following:

```
struct RandomDecisionWithTimeOut (Decision):
  lastFrame = -1
  firstFrame = -1
  lastDecision = false

  timeOut = 1000 # Time out after this number of frames

  def test():
    # check if our stored decision is too old, or if
    # we've timed out
    if frame() > lastFrame + 1 or
       frame() > firstFrame + timeOut:

      # Make a new decision and store it
      lastDecision = randomBoolean()

      # Set when we made the decision
      firstFrame = frame()

    # Either way we need to update the frame value
    lastFrame = frame()

    # We return the stored value
    return lastDecision
```


Again, this decision structure can be used directly with the previous decision tree algorithm.

There can be any number of more sophisticated timing schemes. For example, make the stop time random so there is extra variation, or alternate behaviors when they time out so the agent doesn't happen to stand still multiple times in a row. Use your imagination.

On the Website

WEB PROGRAM

The Random Decision Tree program available on the website is a modified version of the previous Decision Tree program. It replaces some of the decisions in the first version with random decisions and others with a timed-out version. As before, it provides copious amounts of output, so you can see what is going on behind the scenes.

Using Random Decision Trees

We've included this section on random decision trees as a simple extension to the decision tree algorithm. It isn't a common technique. In fact, we've come across it just once.

It is the kind of technique, however, that can breathe a lot more life into a simple algorithm for very little implementation cost. One perennial problem with decision trees is their predictability; they have a reputation for giving AI that is overly simplistic and prone to exploitation. Introducing just a simple random element in this way goes a long way toward rescuing the technique. Therefore, we think it deserves to be used more widely.

5.3 State Machines

Often, characters in a game will act in one of a limited set of ways. They will carry on doing the same thing until some event or influence makes them change. A Covenant warrior in **Halo** [Bungie Software, 2001], for example, will stand at its post until it notices the player, then it will switch into attack mode, taking cover and firing.

We can support this kind of behavior using decision trees, and we've gone some way toward doing that using random decisions. In most cases, however, it is easier to use a technique designed for this purpose: state machines.

State machines are the technique most often used for this kind of decision making and, along with scripting (see Section 5.10), make up the vast majority of decision making systems used in current games.

State machines take account of both the world around them (like decision trees) and their internal makeup (their state).

A Basic State Machine

In a state machine each character occupies one state. Normally, actions or behaviors are associated with each state. So, as long as the character remains in that state, it will continue carrying out the same action.