5 Decision Making

Ask a gamer about game AI, and they think about decision making: the ability of a character to decide what to do. Carrying out that decision (movement, animation, and the like) is taken for granted.

In reality, decision making is typically a small part of the effort needed to build great game AI. Most games use very simple decision making systems: state machines and decision trees. Rule-based systems are rarer, but important.

In recent years a lot of interest has been shown in more sophisticated decision making tools, such as fuzzy logic and neural networks. However, developers haven't been in a rush to embrace these technologies. It can be hard to get them working right.

Decision making is the middle component of our AI model (Figure 5.1), but despite this chapter's name, we will also cover a lot of techniques used in tactical and strategic AI. All the techniques here are applicable to both intra-character and inter-character decision making.

This chapter will look at a wide range of decision making tools, from very simple mechanisms that can be implemented in minutes to comprehensive decision making tools that require more sophistication but can support richer behaviors to complete programming languages embedded in the game. At the end of the chapter we'll look at the output of decision making and how to act on it.

5.1 Overview of Decision Making

Although there are many different decision making techniques, we can look at them all as acting in the same way.

The character processes a set of information that it uses to generate an action that it wants to carry out. The input to the decision making system is the knowledge that a character possesses,

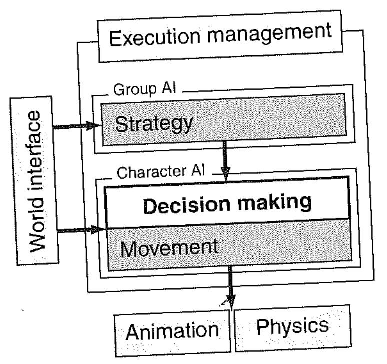

Figure 5.1 The AI model

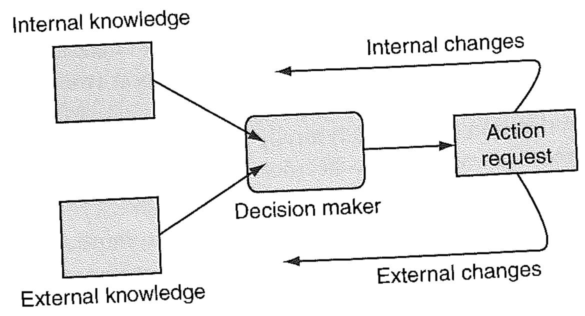

Figure 5.2 Decision making schematic

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.

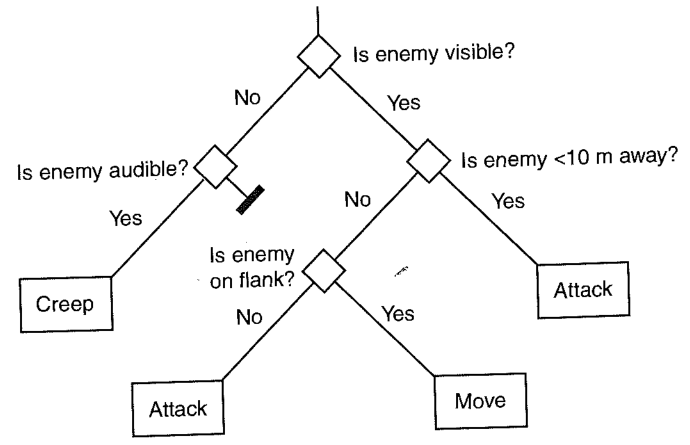

Figure 5.3 A decision tree

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:

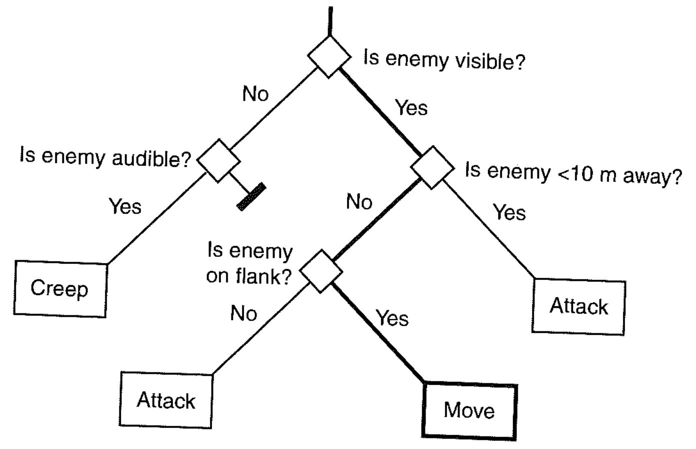

Figure 5.4 The decision tree with a decision made

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."

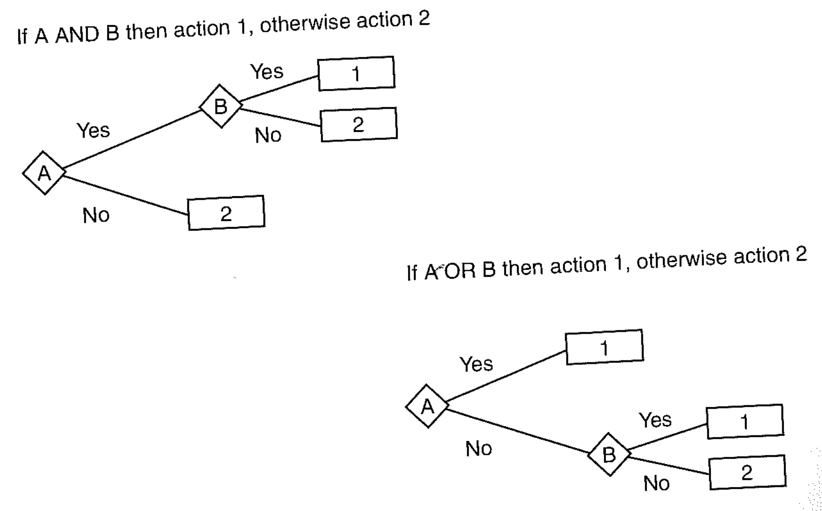
If A AND B then action 1, otherwise action 2
Yes
A
B
Yes
1
No
2
No
2
If A OR B then action 1, otherwise action 2
Yes
1
A
No
B
Yes
1
No
2

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

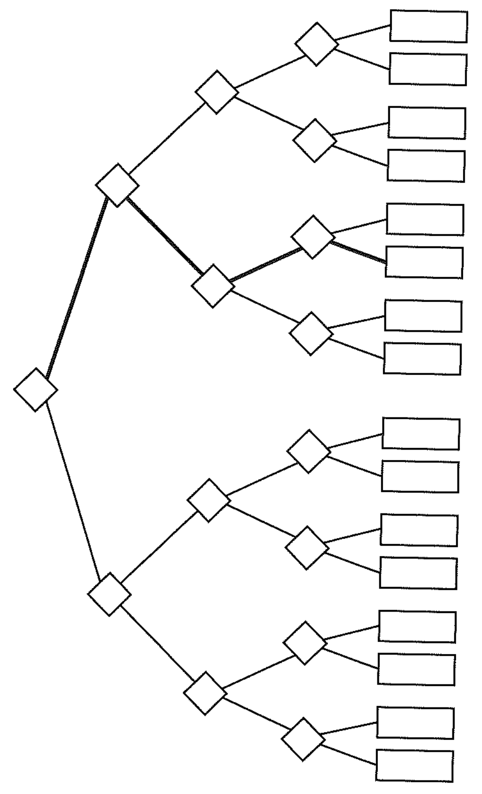

Figure 5.6 Wide decision tree with decision

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

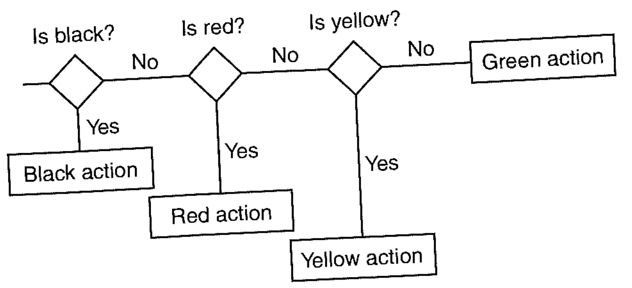

Figure 5.7 Deep binary decision tree

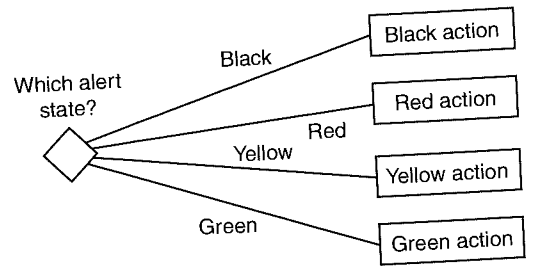

Figure 5.8 Flat decision tree with four branches

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.

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.

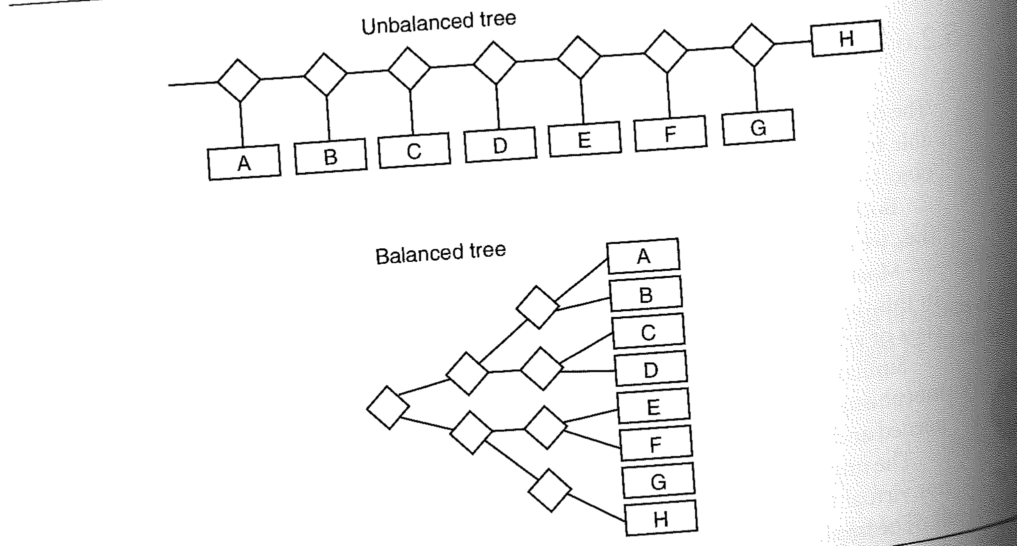

Figure 5.9 Balanced and unbalanced trees

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.

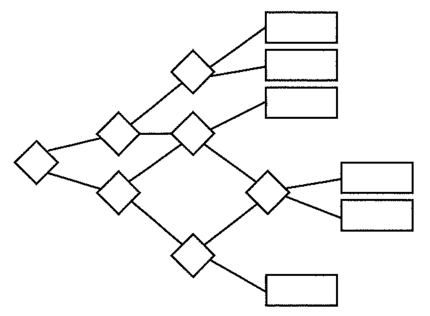

Figure 5.10 Merging branches

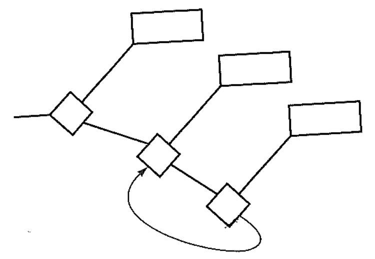

Figure 5.11 Pathological tree

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.

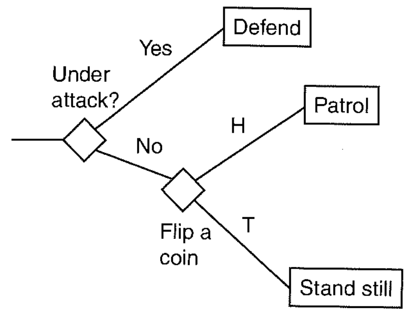

Figure 5.12 Random tree

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.

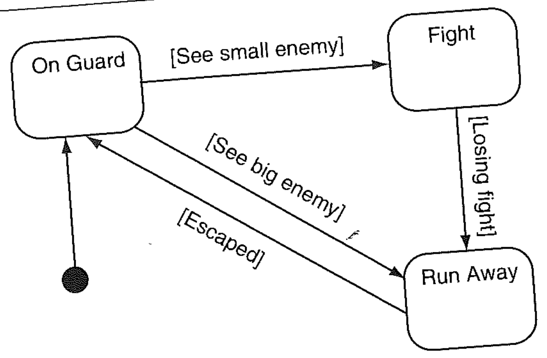

Figure 5.13 A simple state machine

States are connected together by transitions. Each transition leads from one state to another, the target state, and each has a set of associated conditions. If the game determines that the conditions of a transition are met, then the character changes state to the transition's target state. When a transition's conditions are met, it is said to trigger, and when the transition is followed to a new state, it has fired.

Figure 5.13 shows a simple state machine with three states: On Guard, Fight, and Run Away. Notice that each state has its own set of transitions.

The state machine diagrams in this chapter are based on the Unified Modeling Language (UML) state chart diagram format, a standard notation used throughout software engineering. States are shown as curved corner boxes. Transitions are arrowed lines, labeled by the condition that triggers them. Conditions are contained in square brackets.

The solid circle in Figure 5.13 has only one transition without a trigger condition. The transition points to the initial state that will be entered when the state machine is first run.

You won't need an in-depth understanding of UML to understand this chapter. If you want to find out more about UML, we'd recommend Pilone and Pitman [2005].

In a decision tree, the same set of decisions is always used, and any action can be reached through the tree. In a state machine, only transitions from the current state are considered, so not every action can be reached.

Finite State Machines

In game AI any state machine with this kind of structure is usually called a finite state machine (FSM). This and the following sections will cover a range of increasingly powerful state machine implementations, all of which are often referred to as FSMs.

This causes confusion with non-games programmers, for whom the term FSM is more commonly used for a particular type of simple state machine. An FSM in computer science normally refers to an algorithm used for parsing text. Compilers use an FSM to tokenize the input code into symbols that can be interpreted by the compiler.

The Game FSM

The basic state machine structure is very general and admits any number of implementations. We have seen tens of different ways to implement a game FSM, and it is rare to find any two developers using exactly the same technique. That makes it difficult to put forward a single algorithm as being the "state machine" algorithm.

Later in this section, we'll look at a range of different implementation styles for the FSM, but we work through just one main algorithm. We chose it for its flexibility and the cleanness of its implementation.

5.3.1 The Problem

We would like a general system that supports arbitrary state machines with any kind of transition condition. The state machine will conform to the structure given above and will occupy only one state at a time.

5.3.2 The Algorithm

We will use a generic state interface that can be implemented to include any specific code. The state machine keeps track of the set of possible states and records the current state it is in. Alongside each state, a series of transitions is maintained. Each transition is again a generic interface that can be implemented with the appropriate conditions. It simply reports to the state machine whether it is triggered or not.

At each iteration (normally each frame), the state machine's update function is called. This checks to see if any transition from the current state is triggered. The first transition that is triggered is scheduled to fire. The method then compiles a list of actions to perform from the currently active state. If a transition has been triggered, then the transition is fired.

This separation of the triggering and firing of transitions allows the transitions to also have their own actions. Often, transitioning from one state to another also involves carrying out some action. In this case, a fired transition can add the action it needs to those returned by the state.

5.3.3 Pseudo-Code

The state machine holds a list of states, with an indication of which one is the current state. It has an update function for triggering and firing transitions and a function that returns a set of actions to carry out:

```
class StateMachine:

    # Holds a list of states for the machine
    states
```

```
  # Holds the initial state
  initialState

  # Holds the current state
  currentState = initialState

  # Checks and applies transitions, returning a list of
  # actions.
  def update():

    # Assume no transition is triggered
    triggeredTransition = None

    # Check through each transition and store the first
    # one that triggers.
    for transition in currentState.getTransitions():
      if transition.isTriggered():
        triggeredTransition = transition
        break

    # Check if we have a transition to fire
    if triggeredTransition:
      # Find the target state
      targetState = triggeredTransition.getTargetState()

      # Add the exit action of the old state, the
      # transition action and the entry for the new state.
      actions = currentState.getExitAction()
      actions += triggeredTransition.getAction()
      actions += targetState.getEntryAction()

      # Complete the transition and return the action list
      currentState = targetState
      return actions

    # Otherwise just return the current state's actions
    else: return currentState.getAction()
```

5.3.4 Data Structures and Interfaces

The state machine relies on having states and transitions with a particular interface.

The state interface has the following form:

```
class State:
  def getAction()
  def getEntryAction()
  def getExitAction()

  def getTransitions()
```

Each of the `getXAction` methods should return a list of actions to carry out. As we will see below, the `getEntryAction` is only called when the state is entered from a transition, and the `getExitAction` is only called when the state is exited. The rest of the time that the state is active, `getAction` is called. The `getTransitions` method should return a list of transitions that are outgoing from this state.

The transition interface has the following form:

```
class Transition:
  def isTriggered()
  def getTargetState()
  def getAction()
```

The `isTriggered` method returns true if the transition can fire, the `getTargetState` method reports which state to transition to, and the `getAction` method returns a list of actions to carry out when the transition fires.

Transition Implementation

Only one implementation of the state class should be required: it can simply hold the three lists of actions and the list of transitions as data members, returning them in the corresponding get methods.

In the same way, we can store the target state and a list of actions in the transition class and have its methods return the stored values. The `isTriggered` method is more difficult to generalize. Each transition will have its own set of conditions, and much of the power in this method is allowing the transition to implement any kind of tests it likes.

Because state machines are often defined in a data file and read into the game at runtime, it is a common requirement to have a set of generic transitions. The state machine can then be set up from the data file by using the appropriate transitions for each state.

In the previous section on decision trees, we saw generic testing decisions that operated on basic data types. The same principle can be used with state machine transitions: we have generic transitions that trigger when data they are looking at are in a given range.

Unlike decision trees, state machines don't provide a simple way of combining these tests together to make more complex queries. If we need to transition based on the condition that the enemy is far away AND health is low, then we need some way of combining triggers together.

In keeping with our polymorphic design for the state machine, we can accomplish this with the addition of another interface: the condition interface. We can use a general transition class of the following form:

```
class Transition:

  actions
  def getAction(): return actions

  targetState
  def getTargetState(): return targetState

  condition
  def isTriggered(): return condition.test()
```

The isTriggered function now delegates the testing to its condition member.

Conditions have the following simple format:

```
class Condition:
  def test()
```

We can then make a set of sub-classes of the Condition class for particular tests, just like we did for decision trees:

```
class FloatCondition (Condition):
  minValue
  maxValue

  testValue # Pointer to the game data we're interested in

  def test():
    return minValue <= testValue <= maxValue
```

We can combine conditions together using Boolean sub-classes, such as AND, NOT, and OR:

```
class AndCondition (Condition):
  conditionA
  conditionB

```

```
  def test():
    return conditionA.test() and conditionB.test()


class NotCondition (Condition):
  condition

  def test():
    return not condition.test()


class OrCondition (Condition):
  conditionA
  conditionB

  def test():
    return conditionA.test() or conditionB.test()
```

and so on, for any level of sophistication we need.

Weaknesses

This approach to transitions gives a lot of flexibility, but at the price of lots of method calls. In C++ these method calls have to be polymorphic, which can slow down the call and confuse the processor. All this adds time, which may make it unsuitable for use in every frame on lots of characters.

Several developers we have come across use a homegrown scripting language to express conditions for transitions. This still allows designers to create the state machine rules but can be slightly more efficient. In practice, however, the speed up over this approach is quite small, unless the scripting language includes some kind of compilation into machine code (i.e., just-in-time compiling). For all but the simplest code, interpreting a script is at least as time consuming as calling polymorphic functions.

5.3.5 On the Website

WEB PROGRAM

To get a sense of what is happening during an iteration, run the State Machine program that is available from the website. It is a command line program that allows you to manually trigger transitions.

The software displays the current state (states have letters from A to G), and lets you select a transition to trigger. The program clearly shows what is happening at each stage. You see the transition triggered, then which methods are being called, and finally the transition fire. You can also select no transition and see the state's regular action being returned.

5.3.6 Performance

The state machine algorithm only requires memory to hold a triggered transition and the current state. It is O(1) in memory, and O(m) in time, where m is the number of transitions per state.

The algorithm calls other functions in both the state and the transition classes, and in most cases the execution time of these functions accounts for most of the time spent in the algorithm.

5.3.7 Implementation Notes

As we mentioned earlier, there are any number of ways to implement a state machine.

The state machine described in this section is as flexible as possible. We've tried to aim for an implementation that allows you to experiment with any kind of state machine and add interesting features. In many cases, it may be too flexible. If you're only planning to use a small subset of its flexibility, then it is very likely to be unnecessarily inefficient.

5.3.8 Hard-Coded FSM

A few years back, almost all state machines were hard coded. The rules for transitions and the execution of actions were part of the game code. It has become less common as level designers get more control over building the state machine logic, but it is still an important approach.

Pseudo-Code

In a hard-coded FSM, the state machine consists of an enumerated value, indicating which state is currently occupied, and a function that checks if a transition should be followed. Here, we've combined the two into a class definition (although hard-coded FSMs tend to be associated with developers still working in C).

```
class MyFSM:

  # Defines the names for each state
  enum State:
    PATROL
    DEFEND
    SLEEP

  # Holds the current state
  myState

  def update():
```

```

    # Find the correct state
    if myState == PATROL:

      # Example transitions
      if canSeePlayer(): myState = DEFEND
      if tired(): myState = SLEEP

    elif myState == DEFEND:

      # Example transitions
      if not canSeePlayer(): myState = PATROL

    elif myState == SLEEP:

      # Example transitions
      if not tired(): myState = PATROL

  def notifyNoiseHeard(volume):
    if myState == SLEEP and volume > 10:
      myState = DEFEND
```

Notice that this is pseudo-code for a particular state machine rather than a type of state machine. In the update function there is a block of code for each state. In that block of code the conditions for each transition are checked in turn, and the state is updated if required. The transitions in this example all call functions (`tired` and `canSeePlayer`), which we are assuming have access to the current game state.

In addition, we've added a state transition in a separate function, `notifyNoiseHeard`. We are assuming that the game code will call this function whenever the character hears a loud noise. This illustrates the difference between a polling (asking for information explicitly) and an event-based (waiting to be told information) approach to state transitions. Chapter 10 on world interfacing contains more details on this distinction.

The update function is called in each frame, as before, and the current state is used to generate an output action. To do this, the FSM might have a method containing conditional blocks of the following form:

```
def getAction():
  if myState == PATROL: return PatrolAction
  elif myState == DEFEND: return DefendAction
  elif myState == SLEEP: return SleepAction
```

Often, the state machine simply carries out the actions directly, rather than returning details of the action for another piece of code to execute.

Performance

This approach requires no memory and is $O(n + m)$, where n is the number of states, and m is the number of transitions per state.

Although this appears to perform worse than the flexible implementation, it is usually faster in practice for all but huge state machines (i.e., thousands of states).

Weaknesses

Although hard-coded state machines are easy to write, they are notoriously difficult to maintain. State machines in games can often get fairly large, and this can appear as ugly and unclear code.

Most developers, however, find that the main drawback is the need for programmers to write the AI behaviors for each character. This implies a need to recompile the game each time the behavior changes. While it may not be a problem for a hobby game writer, it can become critical in a large game project that takes many minutes or hours to rebuild.

More complex structures, such as hierarchical state machines (see below), are also difficult to coordinate using hard-coded FSMs. With a more flexible implementation, debugging output can easily be added to all state machines, making it easier to track down problems in the AI.

5.3.9 Hierarchical State Machines

On its own, one state machine is a powerful tool, but it can be difficult to express some behaviors. One common source of difficulty is "alarm behaviors."

Imagine a service robot that moves around a facility cleaning the floors. It has a state machine allowing it to do this. It might search around for objects that have been dropped, pick one up when it finds it, and carry it off to the trash compactor. This can be simply implemented using a normal state machine (see Figure 5.14).

Unfortunately, the robot can run low on power, whereupon it has to scurry off to the nearest electrical point and get recharged. Regardless of what it is doing at the time, it needs to stop, and when it is fully charged again it needs to pick up where it left off. The recharging periods could allow the player to sneak by unnoticed, for example, or allow the player to disable all electricity to the area and thereby disable the robot.

This is an alarm mechanism: something that interrupts normal behavior to respond to something important. Representing this in a state machine leads to a doubling in the number of states.

With one level of alarm this isn't a problem, but what would happen if we wanted the robot to hide when fighting breaks out in the corridor. If its hiding instinct is more important than its refueling instinct, then it will have to interrupt refueling to go hide. After the battle it will need to pick up refueling where it left off, after which it will pick up whatever it was doing before that. For just 2 levels of alarm, we would have 16 states.

Rather than combining all the logic into a single state machine, we can separate it into several. Each alarm mechanism has its own state machine, along with the original behavior. They are arranged in a hierarchy, so the next state machine down is only considered when the higher level state machine is not responding to its alarm.

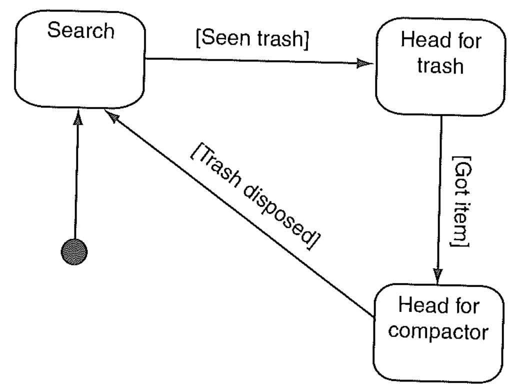

Figure 5.14 The basic cleaning robot state machine

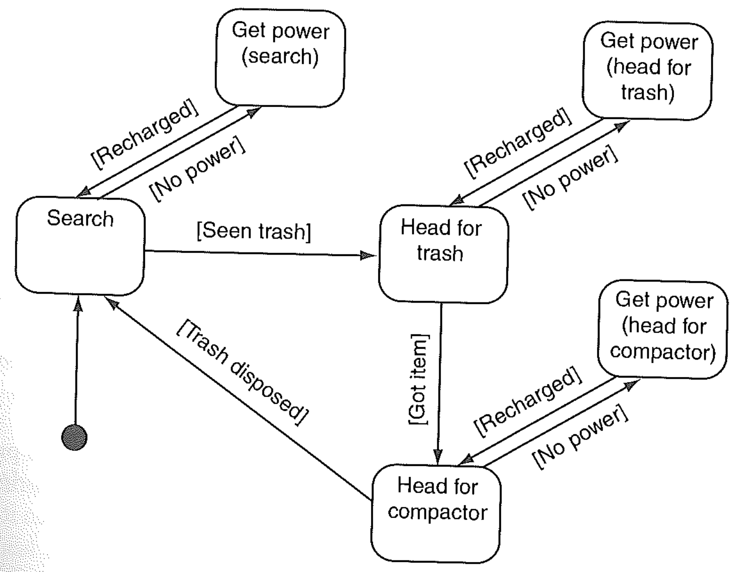

Figure 5.15 An alarm mechanism in a standard state machine

Figure 5.15 shows one alarm mechanism and corresponds exactly to the diagram above.

We will nest one state machine inside another to indicate a hierarchical state machine (Figure 5.16). The solid circle again represents the start state of the machine. When a composite state is first entered, the circle with H* inside it indicates which sub-state should be entered.

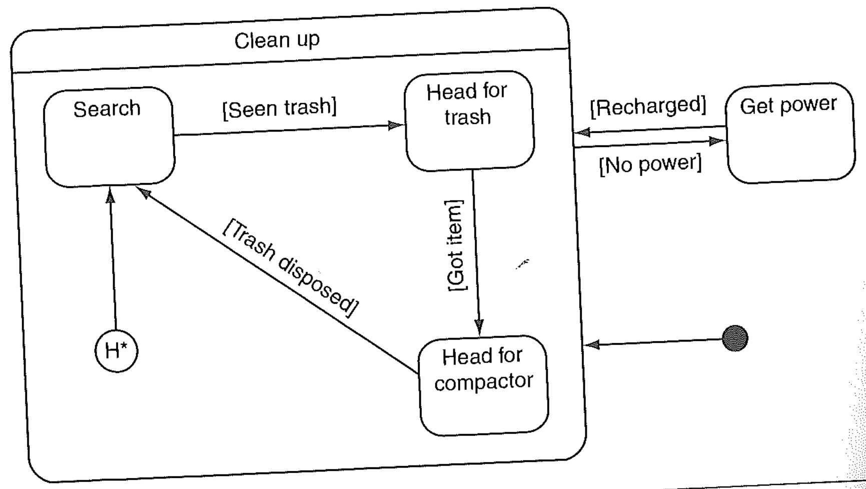

Figure 5.16 A hierarchical state machine for the robot

If the composite state has already been entered, then the previous sub-state is returned to. The H* node is called the "history state" for this reason.

The details of why there's an asterisk after the H, and some of the other vagaries of the UML state chart diagram, are beyond the scope of this chapter. Refer back to Pilone and Pitman [2005] for more details.

Rather than having separate states to keep track of the non-alarm state, we introduce nested states. We still keep track of the state of the cleaning state machine, even if we are in the process of refueling. When the refueling is over, the cleaning state machine will pick up where it left off.

In effect, we are in more than one state at once. We might be in the "Refuel" state in the alarm mechanism, while at the same time we are also in the "Pick Up Object" state in the cleaning machine. Because there is a strict hierarchy, there is never any confusion about which state wins out: the highest state in the hierarchy is always in control.

To implement this, we could simply arrange the state machines in our program so that one state machine calls another if it needs to. So if the refueling state machine is in its "Clean Up" state, it calls the cleaning state machine and asks it for the action to take. When it is in the "Refuel" state, it returns the refueling action directly.

While this would lead to slightly ugly code, it would implement our scenario. Most hierarchical state machines, however, support transitions between levels of the hierarchy, and for that we'll need more complex algorithms.

For example, let's expand our robot so that it can do something useful if there are no objects to collect. It makes sense that it will use the opportunity to go and recharge, rather than standing around waiting for its battery to go flat. The new state machine is shown in Figure 5.17.

Notice that we've added one more transition: from the "Search" state right out into the "Refuel" state. This transition is triggered when there are no objects to collect. Because we transitioned

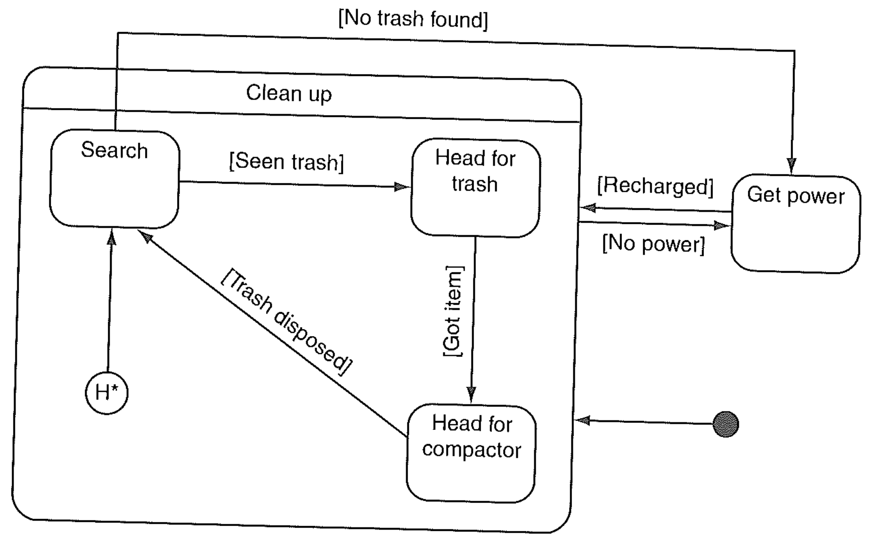

Figure 5.17 A hierarchical state machine with a cross-hierarchy transition

directly out of this state, the inner state machine no longer has any state. When the robot has refueled and the alarm system transitions back to cleaning, the robot will not have a record of where to pick up from, so it must start the state machine again from its initial node ("Search").

The Problem

We'd like an implementation of a state machine system that supports hierarchical state machines. We'd also like transitions that pass between different layers of the machine.

The Algorithm

In a hierarchical state machine, each state can be a complete state machine in its own right. We therefore rely on recursive algorithms to process the whole hierarchy. As with most recursive algorithms, this can be pretty tricky to follow. The simplest implementation covered here is doubly tricky because it recurses up and down the hierarchy at different points. We'd encourage you to use the informal discussion and examples in this section alongside the pseudo-code in the next section and play with the Hierarchical State Machine program that is available on the website to get a feel for how it is all working.

The first part of the system returns the current state. The result is a list of states, from highest to lowest in the hierarchy. The state machine asks its current state to return its hierarchy. If the

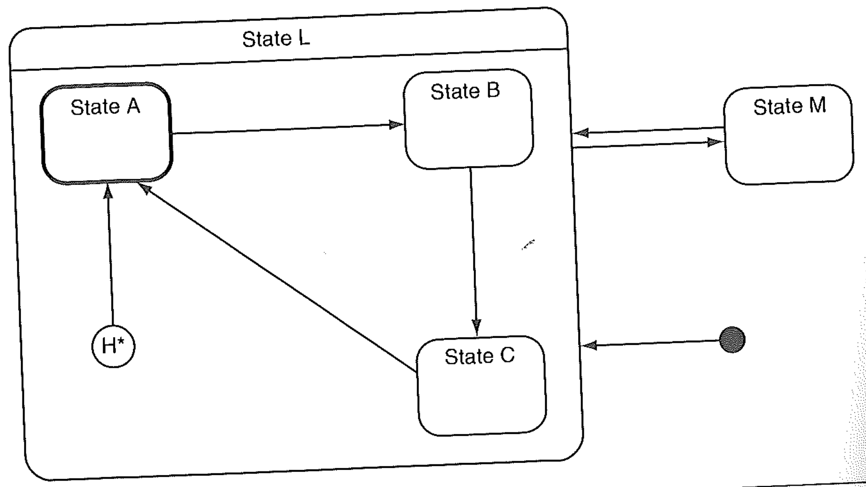

Figure 5.18 Current state in a hierarchy

state is a terminal state, it returns itself; otherwise, it returns itself and adds to it the hierarchy of state from its own current state.

In Figure 5.18 the current state is [State L, State A].

The second part of the hierarchical state machine is its update. In the original state machine we assumed that each state machine started off in its initial state. Because the state machine always transitioned from one state to another, there was never any need to check if there was no state. State machines in a hierarchy can be in no state; they may have a cross-hierarchy transition. The first stage of the update, then, is to check if the state machine has a state. If not, it should enter its initial state.

Next, we check if the current state has a transition it wants to execute. Transitions at higher levels in the hierarchy always take priority, and the transitions of sub-states will not be considered if the super-state has one that triggers.

A triggered transition may be one of three types: it might be a transition to another state at the current level of the hierarchy, it might be a transition to a state higher up in the hierarchy, or it might be a transition to a state lower in the hierarchy. Clearly, the transition needs to provide more data than just a target state. We allow it to return a relative level; how many steps up or down the hierarchy the target state is.

We could simply search the hierarchy for the target state and not require an explicit level. While this would be more flexible (we wouldn't have to worry about the level values being wrong), it would be considerably more time consuming. A hybrid, but fully automatic, extension could search the hierarchy once offline and store all appropriate level values.

So the triggered transition has a level of zero (state is at the same level), a level greater than zero (state is higher in the hierarchy), or a level less than zero (state is lower in the hierarchy). It acts differently depending on which category the level falls into.

If the level is zero, then the transition is a normal state machine transition and can be performed at the current level, using the same algorithm used in the finite state machine.

If the level is greater than zero, then the current state needs to be exited and nothing else needs to be done at this level. The exit action is returned, along with an indication to whomever called the update function that the transition hasn't been completed. We will return the exit action, the transition outstanding, and the number of levels higher to pass the transition. This level value is decreased by one as it is returned. As we will see, the update function will be returning to the next highest state machine in the hierarchy.

If the level is less than zero, then the current state needs to transition to the ancestor of the target state on the current level in the hierarchy. In addition, each of the children of that state also needs to do the same, down to the level of the final destination state. To achieve this we use a separate function, `updateDown`, that recursively performs this transition from the level of the target state back up to the current level and returns any exit and entry actions along the way. The transition is then complete and doesn't need to be passed on up. All the accumulated actions can be returned.

So we've covered all possibilities if the current state has a transition that triggers. If it does not have a transition that triggers, then its action depends on whether the current state is a state machine itself. If not, and if the current state is a plain state, then we can return the actions associated with being in that state, just as before.

If the current state is a state machine, then we need to give it the opportunity to trigger any transitions. We can do this by calling its update function. The update function will handle any triggers and transitions automatically. As we saw above, a lower level transition that fires may have its target state at a higher level. The update function will return a list of actions, but it may also return a transition that it is passing up the hierarchy and that hasn't yet been fired.

If such a transition is received, its level is checked. If the level is zero, then the transition should be acted on at this level. The transition is honored, just as if it were a regular transition for the current state. If the level is still greater than zero (it should never be less than zero, because we are passing up the hierarchy at this point), then the state machine should keep passing it up. It does this, as before, by exiting the current state and returning the following pieces of information: the exit action, any actions provided by the current state's update function, the transition that is still pending, and the transition's level, less one.

If no transition is returned from the current state's update function, then we can simply return its list of actions. If we are at the top level of the hierarchy, the list alone is fine. If we are lower down, then we are also within a state, so we need to add the action for the state we're in to the list we return.

Fortunately, this algorithm is at least as difficult to explain as it is to implement. To see how and why it works, let's work through an example.

Examples

Figure 5.19 shows a hierarchical state machine that we will use as an example.

To clarify the actions returned for each example, we will say S-entry is the set of entry actions for state S and similarly S-active and S-exit for active and exit actions. In transitions, we use the same format: 1-actions for the actions associated with transition 1.

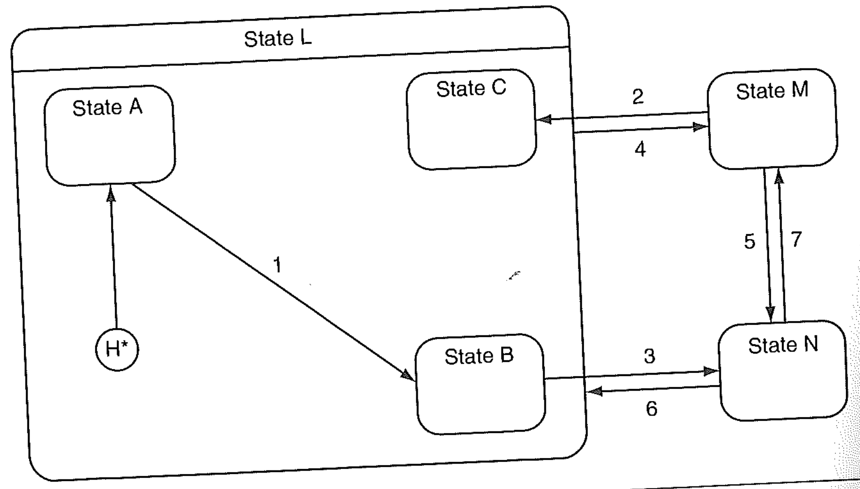

Figure 5.19 Hierarchical state machine example

These examples can appear confusing if you skim through them. If you're having trouble with the algorithm, we urge you to follow through step by step with both the diagram above and the pseudo-code from the next section.

Suppose we start just in State L, and no transition triggers. We will transition into State [L, A], because L's initial state is A. The update function will return L-active and A-entry, because we are staying in L and just entering A.

Now suppose transition 1 is the only one that triggers. The top-level state machine will detect no valid transitions, so it will call state machine L to see if it has any. L finds that its current state (A) has a triggered transition. Transition 1 is a transition at the current level, so it is handled within L and not passed anywhere. A transitions to B, and L's update function returns A-exit, 1-actions, B-entry. The top-level state machine accepts these actions and adds its own active action. Because we have stayed in State L throughout, the final set of actions is A-exit, 1-actions, B-entry, L-active. The current state is [L, B].

From this state, transition 4 triggers. The top-level state machine sees that transition 4 triggers, and because it is a top-level transition it can be honored immediately. The transition leads to State M, and the corresponding actions are L-exit, 4-actions, M-entry. The current state is [M]. Note that L is still keeping a record of being in State B, but because the top-level state machine is in State M, this record isn't used at the moment.

We'll go from State M to State N in the normal way through transition 5. The procedure is exactly the same as for the previous example and the non-hierarchical state machine. Now transition 6 triggers. Because it is a level zero transition, the top-level state machine can honor it immediately. It transitions into State L and returns the actions N-exit, 6-actions, L-entry. But now L's record of being in State B is important; we end up in State [L, B] again. In our implementation we don't return the B-entry action, because we didn't return the B-exit action when we left State L

previously. This is a personal preference on our part and isn't fixed in stone. If you want to exit and reenter State B, then you can modify your algorithm to return these extra actions at the appropriate time.

Now suppose from State [L, B] transition 3 triggers. The top-level state machine finds no triggers, so it will call state machine L to see if it has any. L finds that State B has a triggered transition. This transition has a level of one; its target is one level higher in the hierarchy. This means that State B is being exited, and it means that we can't honor the transition at this level. We return B-exit, along with the uncompleted transition, and the level minus one (i.e., zero, indicating that the next level up needs to handle the transition). So, control returns to the top-level update function. It sees that L returned an outstanding transition, with zero level, so it honors it, transitioning in the normal way to State N. It combines the actions that L returned (namely, B-exit) with the normal transition actions to give a final set of actions: B-exit, L-exit, 3-actions, N-entry. Note that, unlike in our third example, L is no longer keeping track of the fact that it is in State B, because we transitioned out of that state. If we fire transition 6 to return to State L, then State L's initial state (A) would be entered, just like in the first example.

Our final example covers transitions with level less than zero. Suppose we moved from State N to State M via transition 7. Now we make transition 2 trigger. The top-level state machine looks at its current state (M) and finds transition 2 triggered. It has a level of minus one, because it is descending one level in the hierarchy. Because it has a level of minus one, the state machine calls the `updateDown` function to perform the recursive transition. The `updateDown` function starts at the state machine (L) that contains the final target state (C), asking it to perform the transition at its level. State machine L, in turn, asks the top-level state machine to perform the transition at its level. The top-level state machine changes from State M to State L, returning M-exit, L-entry as the appropriate actions. Control returns to state machine L's `updateDown` function. State machine L checks if it is currently in any state (it isn't, since we left State B in the last example). It adds its action (C-entry) to those returned by the top-level machine. Control then returns to the top-level state machine's update function: the descending transition has been honored; it adds the transition's actions to the result and returns M-exit, 2-actions, L-entry, C-entry.

If state machine L had still been in State B, then when L's `updateDown` function was called it would transition out of B and into C. It would add B-exit and C-entry to the actions that it received from the top-level state machine.

Pseudo-Code

The hierarchical state machine implementation is made up of five classes and forms one of the longest algorithms in this book. The `State` and `Transition` classes are similar to those in the regular finite state machine. The `HierarchicalStateMachine` class runs state transitions, and `SubMachineState` combines the functionality of the state machine and a state. It is used for state machines that aren't at the top level of the hierarchy. All classes but `Transition` inherit from a `HSMBase` class, which simplifies the algorithm by allowing functions to treat anything in the hierarchy in the same way.

The HSMBase has the following form:

```
class HSMBase:
  # The structure returned by update
  struct UpdateResult:
    actions
    transition
    level

  def getAction(): return []

  def update():
    UpdateResult result
    result.actions = getAction()
    result.transition = None
    result.level = 0
    return result

  def getStates() # unimplemented function
```

The HierarchicalStateMachine class has the following implementation:

```
class HierarchicalStateMachine (HSMBase):

  # List of states at this level of the hierarchy
  states

  # The initial state for when the machine has no
  # current state.
  initialState

  # The current state of the machine.
  currentState = initialState

  # Gets the current state stack
  def getStates():
    if currentState: return currentState.getStates()
    else: return []

  # Recursively updates the machine.
  def update():

    # If we're in no state, use the initial state
```

```
    if not currentState:
      currentState = initialState
      return currentState.getEntryAction()

    # Try to find a transition in the current state
    triggeredTransition = None
    for transition in currentState.getTransitions():
      if transition.isTriggered():
        triggeredTransition = transition
        break

    # If we've found one, make a result structure for it
    if triggeredTransition:
      result = UpdateResult()
      result.actions = []
      result.transition = triggeredTransition
      result.level = triggeredTransition.getLevel()

    # Otherwise recurse down for a result
    else:
      result = currentState.update()

    # Check if the result contains a transition
    if result.transition:

      # Act based on its level
      if result.level == 0:

        # Its on our level: honor it
        targetState = result.transition.getTargetState()
        result.actions += currentState.getExitAction()
        result.actions += result.transition.getAction()
        result.actions += targetState.getEntryAction()

        # Set our current state
        currentState = targetState

        # Add our normal action (we may be a state)
        result.actions += getAction()

        # Clear the transition, so nobody else does it
        result.transition = None

    else if result.level > 0:
      # Its destined for a higher level
      # Exit our current state
      result.actions += currentState.getExitAction()
      currentState = None

      # Decrease the number of levels to go
      result.level -= 1

    else:

      # It needs to be passed down
      targetState = result.transition.getTargetState()
      targetMachine = targetState.parent
      result.actions += result.transition.getAction()
      result.actions += targetMachine.updateDown(
        targetState, -result.level
        )

      # Clear the transition, so nobody else does it
      result.transition = None

  # If we didn't get a transition
  else:

    # We can simply do our normal action
    result.action += getAction()

  # Return the accumulated result
  return result

# Recurses up the parent hierarchy, transitioning into
# each state in turn for the given number of levels
def updateDown(state, level):

  # If we're not at top level, continue recursing
  if level > 0:
    # Pass ourself as the transition state to our parent
    actions = parent.updateDown(this, level-1)

  # Otherwise we have no actions to add to
  else: actions = []

  # If we have a current state, exit it
```

```
  if currentState:
    actions += currentState.getExitAction()

  # Move to the new state, and return all the actions
  currentState = state
  actions += state.getEntryAction()
  return actions
```

The State class is substantially the same as before, but adds an implementation for getStates:

```
class State (HSMBase):

  def getStates():
    # If we're just a state, then the stack is just us
    return [this]

  # As before...
  def getAction()
  def getEntryAction()
  def getExitAction()
  def getTransitions()
```

Similarly, the Transition class is the same but adds a method to retrieve the level of the transition:

```
class Transition:

  # Returns the difference in levels of the hierarchy from
  # the source to the target of the transition.
  def getLevel()

  # As before...
  def isTriggered()
  def getTargetState()
  def getAction()
```

Finally, the SubMachineState class merges the functionality of a state and a state machine:

```
class SubMachineState (State, HierarchicalStateMachine):

  # Route get action to the state
```

```
def getAction(): return State::getAction()

# Route update to the state machine
def update(): return HierarchicalStateMachine::update()

# We get states by adding ourself to our active children
def getStates():
  if currentState:
    return [this] + currentState.getStates()
  else:
    return [this]
```

Implementation Notes

We've used multiple inheritance to implement `SubMachineState`. For languages (or programmers) that don't support multiple inheritance, there are two options. The `SubMachineState` could encapsulate `HierarchicalStateMachine`, or the `HierarchicalStateMachine` can be converted so that it is a sub-class of `State`. The downside with the latter approach is that the top-level state machine will always return its active action from the update function, and `getStates` will always have it as the head of the list.

WEB LIBRARY

We've elected to use a polymorphic structure for the state machine again. It is possible to implement the same algorithm without any polymorphic method calls. Given that it is complex enough already, however, we'll leave that as an exercise. Our experience deploying a hierarchical state machine involved an implementation using polymorphic method calls (provided on the website). In-game profiling on both PC and PS2 showed that the method call overhead was not a bottleneck in the algorithm. In a system with hundreds or thousands of states, it may well be, as cache efficiency issues come into play.

Some implementations of hierarchical state machines are significantly simpler than this by making it a requirement that transitions can only occur between states at the same level. With this requirement, all the recursion code can be eliminated. If you don't need cross-hierarchy transitions, then the simpler version will be easier to implement. It is unlikely to be any faster, however. Because the recursion isn't used when the transition is at the same level, the code above will run about as fast if all the transitions have a zero level.

Performance

The algorithm is $O(n)$ in memory, where n is the number of layers in the hierarchy. It requires temporary storage for actions when it recurses down and up the hierarchy.

Similarly, it is $O(nt)$ in time, where t is the number of transitions per state. To find the correct transition to fire, it potentially needs to search each transition at each level of the hierarchy and $O(nt)$ process. The recursion, both for a transition level <0 and for a level >0 is $O(n)$, so it does not affect the $O(nt)$ for the whole algorithm.

On the Website

Following hierarchical state machines, especially when they involve transitions across hierarchies, can be confusing at first. We've tried to be as apologetic as possible for the complexity of the algorithm, even though we've made it as simple as we can. Nonetheless, it is a powerful technique to have in your arsenal and worth the effort to master.

WEB PROGRAM

The Hierarchical State Machine program that is available on the webiste lets you step through a state machine, triggering any transition at each step. It works in the same way as the State Machine program, giving you plenty of feedback on transitions. We hope it will help give a clearer picture, alongside the content of this chapter.

5.3.10 Combining Decision Trees and State Machines

The implementation of transitions bears more than a passing resemblance to the implementation of decision trees. This is no coincidence, but we can take it even further.

Decision trees are an efficient way of matching a series of conditions, and this has application in state machines for matching transitions.

We can combine the two approaches by replacing transitions from a state with a decision tree. The leaves of the tree, rather than being actions as before, are transitions to new states.

A simple state machine might look like Figure 5.20.

The diamond symbol is also part of the UML state chart diagram format, representing a decision. In UML there is no differentiation between decisions and transitions, and the decisions themselves are usually not labeled.

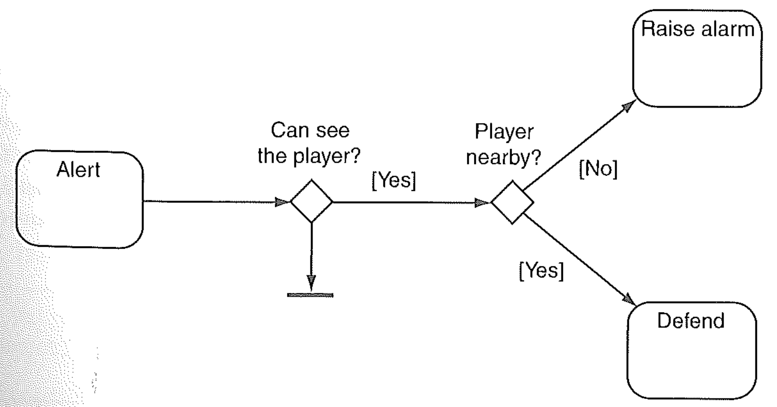

Figure 5.20 State machine with decision tree transitions

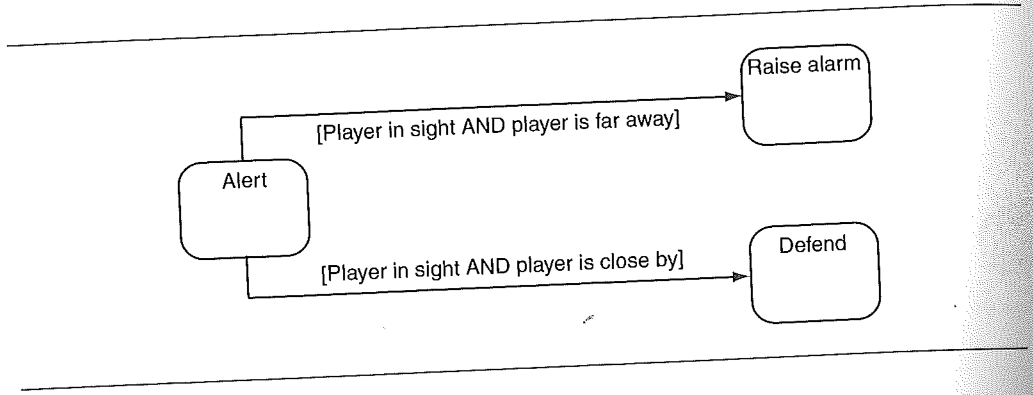
Raise alarm
[Player in sight AND player is far away]
Alert
Defend
[Player in sight AND player is close by]

Figure 5.21 State machine without decision tree transitions

In this book we've labeled the decisions with the test that they perform, which is clearer for our needs.

When in the "Alert" state, a sentry has only one possible transition: via the decision tree. It quickly ascertains whether the sentry can see the player. If the sentry is not able to see the player, then the transition ends and no new state is reached. If the sentry is able to see the player, then the decision tree makes a choice based on the distance of the player. Depending on the result of this choice, two different states may be reached: "Raise Alarm" or "Defend." The latter can only be reached if a further test (distance to the player) passes.

To implement the same state machine without the decision nodes, the state machine in Figure 5.21 would be required. Note that now we have two very complex conditions and both have to evaluate the same information (distance to the player and distance to the alarm point). If the condition involved a time-consuming algorithm (such as the line of sight test in our example), then the decision tree implementation would be significantly faster.

Pseudo-Code

We can incorporate a decision tree into the state machine framework we've developed so far.

The decision tree, as before, consists of `DecisionTreeNodes`. These may be decisions (using the same `Decision` class as before) or `TargetStates` (which replace the `Action` class in the basic decision tree). `TargetStates` hold the state to transition to and can contain actions. As before, if a branch of the decision tree should lead to no result, then we can have some null value at the leaf of the tree.

```
class TargetState (DecisionTreeNode):
  getAction()
  getTargetState()
```

The decision making algorithm needs to change. Rather than testing for `Actions` to return, it now tests for `TargetState` instances:

```
def makeDecision(node):

  # Check if we need to make a decision
  if not node or node is_instance_of TargetState:

    # We've got the target (or a null target); return it
    return node

  else:
    # Make the decision and recurse based on the result
    if node.test():
      return makeDecision(node.trueNode)
    else
      return makeDecision(node.falseNode)
```

We can then build an implementation of the `Transition` interface that supports these decision trees. It has the following algorithm:

```
class DecisionTreeTransition (Transition):

  # Holds the target state at the end of the decision
  # tree, when a decision has been made
  targetState = None

  # Holds the root decision in the tree
  decisionTreeRoot

  def getAction():
    if targetState: return targetState.getAction()
    else: return None

  def getTargetState():
    if targetState: return targetState.getTargetState()
    else: return None

  def isTriggered():

    # Get the result of the decision tree and store it
    targetState = makeDecision(decisionTreeRoot)

    # Return true if the target state points to a
    # destination, otherwise assume that we don't trigger
    return targetState != None
```

5.4 Behavior Trees

Behavior trees have become a popular tool for creating AI characters. **Halo 2** [Bungie Software, 2004] was one of the first high-profile games for which the use of behavior trees was described in detail and since then many more games have followed suit.

They are a synthesis of a number of techniques that have been around in AI for a while: Hierarchical State Machines, Scheduling, Planning, and Action Execution. Their strength comes from their ability to interleave these concerns in a way that is easy to understand and easy for non-programmers to create. Despite their growing ubiquity, however, there are things that are difficult to do well in behavior trees, and they aren't always a good solution for decision making.

Behavior trees have a lot in common with Hierarchical State Machines but, instead of a state, the main building block of a behavior tree is a *task*. A task can be something as simple as looking up the value of a variable in the game state, or executing an animation.

Tasks are composed into sub-trees to represent more complex actions. In turn, these complex actions can again be composed into higher level behaviors. It is this composability that gives behavior trees their power. Because all tasks have a common interface and are largely self-contained, they can be easily built up into hierarchies (i.e., behavior trees) without having to worry about the details of how each sub-task in the hierarchy is implemented.

Types of Task

Tasks in a behavior tree all have the same basic structure. They are given some CPU time to do their thing, and when they are ready they return with a status code indicating either success or failure (a Boolean value would suffice at this stage). Some developers use a larger set of return values, including an *error* status, when something unexpected went wrong, or a *need more time* status for integration with a scheduling system.

While tasks of all kinds can contain arbitrarily complex code, the most flexibility is provided if each task can be broken into the smallest parts that can usefully be composed. This is especially so because, while powerful just as a programming idiom, behavior trees really shine when coupled with a graphical user interface (GUI) to edit the trees. That way, designers, technical artists and level designers can potentially author complex AI behavior.

At this stage, our simple behavior trees will consist of three kinds of tasks: Conditions, Actions, and Composites.

Conditions test some property of the game. There can be tests for proximity (is the character within X units of an enemy?), tests for line of sight, tests on the state of the character (am I healthy?, do I have ammo?), and so on. Each of these kinds of tests needs to be implemented as a separate task, usually with some parameterization so they can be easily reused. Each Condition returns the success status code if the Condition is met and returns failure otherwise.

Actions alter the state of the game. There can be Actions for animation, for character movement, to change the internal state of the character (resting raises health, for example), to play audio samples, to engage the player in dialog, and to engage specialized AI code (such as pathfinding). Just like Conditions, each Action will need to have its own implementation, and there may be a

large number of them in your engine. Most of the time Actions will succeed (if there's a chance they might not, it is better to use Conditions to check for that before the character starts trying to act). It is possible to write Actions that fail if they can't complete, however.

If Conditions and Actions seem familiar from our previous discussion on decision trees and state machines, they should. They occupy a similar role in each technique (and we'll see more techniques with the same features later in this chapter). The key difference in behavior trees, however, is the use of a single common interface for all tasks. This means that arbitrary Conditions, Actions, and groups can be combined together without any of them needing to know what else is in the behavior tree.

Both Conditions and Actions sit at the leaf nodes of the tree. Most of the branches are made up of Composite nodes. As the name suggests, these keep track of a collection of child tasks (Conditions, Actions, or other Composites), and their behavior is based on the behavior of their children. Unlike Actions and Conditions, there are normally only a handful of Composite tasks because with only a handful of different grouping behaviors we can build very sophisticated behaviors.

For our simple behavior tree we'll consider two types of Composite tasks: Selector and Sequence. Both of these run each of their child behaviors in turn. When a child behavior is complete and returns its status code the Composite decides whether to continue through its children or whether to stop there and then and return a value.

A Selector will return immediately with a success status code when one of its children runs successfully. As long as its children are failing, it will keep on trying. If it runs out of children completely, it will return a failure status code.

A Sequence will return immediately with a failure status code when one of its children fails. As long as its children are succeeding, it will keep going. If it runs out of children, it will return in success.

Selectors are used to choose the first of a set of possible actions that is successful. A Selector might represent a character wanting to reach safety. There may be multiple ways to do that (take cover, leave a dangerous area, find backup). The Selector will first try to take cover; if that fails, it will leave the area. If that succeeds, it will stop—there's no point also finding backup, as we've solved the character's goal of reaching safety. If we exhaust all options without success, then the Selector itself has failed.

A Selector task is depicted graphically in Figure 5.22. First the Selector tries a task representing attacking the player; if it succeeds, it is done. If the attack task fails, the Selector node will go on to try a taunting animation instead. As a final fall back, if all else fails, the character can just stare menacingly.

Sequences represent a series of tasks that need to be undertaken. Each of our reaching-safety actions in the previous example may consist of a Sequence. To find cover we'll need to choose a cover point, move to it, and, when we're in range, play a roll animation to arrive behind it. If any of the steps in the sequence fails, then the whole sequence has failed: if we can't reach our desired cover point, then we haven't reached safety. Only if all the tasks in the Sequence are successful can we consider the Sequence as a whole to be successful.

Figure 5.23 shows a simple example of using a Sequence node. In this behavior tree the first child task is a condition that checks if there is a visible enemy. If the first child task fails, then the Sequence task will also immediately fail. If the first child task succeeds then we know there is a

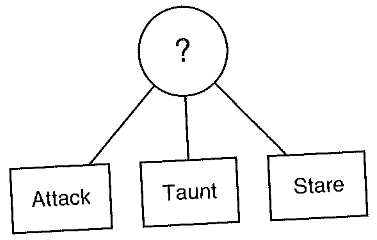

Figure 5.22 Example of a selector node in a behavior tree

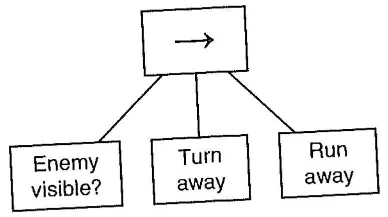

Figure 5.23 Example of a sequence node in a behavior tree

visible enemy, and the Sequence task goes on to execute the next child task, which is to turn away, followed by the running task. The Sequence task will then terminate successfully.

A Simple Example

We can use the tasks in the previous example to build a simple but powerful behavior tree. The behavior tree in this example represents an enemy character trying to enter the room in which the player is standing.

We'll build the tree in stages, to emphasize how the tree can be built up and extended. This process of refining the behavior tree is part of its attraction, as simple behaviors can be roughed in and then refined in response to play testing and additional development resources.

Our first stage, Figure 5.24, shows a behavior tree made up of a single task. It is a move action to be carried out using whatever steering system our engine provides.

To run this task we give it CPU time, and it moves into the room. This was state-of-the-art AI for entering rooms before Half-Life, of course, but wouldn't go down well in a shooter now! The simple example does make a point, however. When you're developing your AI using behavior trees, just a single naive behavior is all you need to get *something* working.

In our case, the enemy is too stupid: the player can simply close the door and confound the incoming enemy.

Move (into room)

Figure 5.24 The simplest behavior tree

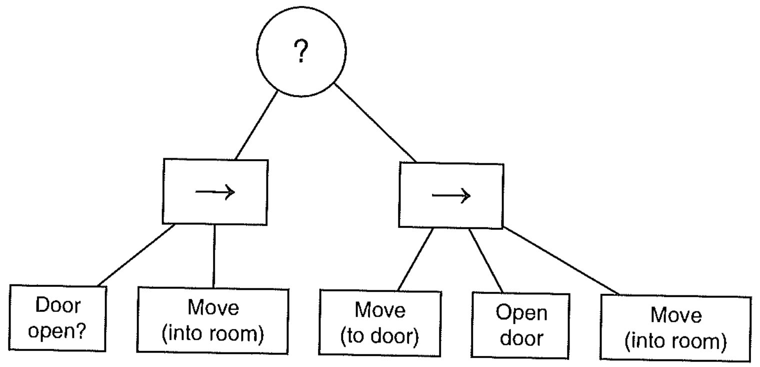

Figure 5.25 A behavior tree with composite nodes

So, we'll need to make the tree a little more complex. In Figure 5.25, the behavior tree is made up of a Selector, which has two different things it can try, each of which is a Sequence. In the first case, it checks to see if the door is open, using a Condition task; then it moves into the room. In the second case, it moves to the door, plays an animation, opens the door, and then moves into the room.

Let's think about how this behavior tree is run. Imagine the door is open. When it is given CPU time, the Selector tries its first child. That child is made up of the Sequence task for moving through the open door. The Condition checks if the door is open. It is, so it returns success. So, the Sequence task moves on to its next child—moving through the door. This, like most actions, always succeeds, so the whole of the Sequence has been successful. Back at the top level, the Selector has received a success status code from the first child it tried, so it doesn't both trying its other child: it immediately returns in success.

What happens when the door is closed? As before the Selector tries its first child. That Sequence tries the Condition. This time, however, the Condition task fails. The Sequence doesn't bother continuing; one failure is enough, so it returns in failure. At the top level, the Selector isn't fazed by a failure; it just moves onto its next child. So, the character moves to the door, opens it, then enters.

This example shows an important feature of behavior trees: a Condition task in a Sequence acts like an IF-statement in a programming language. If the Condition is not met, then the Sequence will not proceed beyond that point. If the Sequence is in turn placed within a Selector, then we get the effect of IF-ELSE-statements: the second child is only tried if the Condition wasn't met for the first child. In pseudo-code the behavior of this tree is:

```
if is_locked(door):
    move_to(door)
    open(door)
    move_to(room)
else:
    move_to(room)
```

The pseudo-code and diagram show that we're using the final move action in both cases. There's nothing wrong with this. Later on in the section we'll look at how to reuse existing sub-trees efficiently. For now it is worth saying that we could refactor our behavior tree to be more like the simpler pseudo-code:

```
if is_locked(door):
    move_to(door)
    open(door)
move_to(room)
```

The result is shown in Figure 5.26. Notice that it is deeper than before; we've had to add another layer to the tree. While some people do like to think about behavior trees in terms of source code, it doesn't necessarily give you any insight in how to create simple or efficient trees.

In our final example in this section we'll deal with the possibility that the player has locked the door. In this case, it won't be enough for the character to just assume that the door can be opened. Instead, it will need to try the door first. Figure 5.27 shows a behavior tree for dealing with this situation. Notice that the Condition used to check if the door is locked doesn't appear at the same point where we check if the door is closed. Most people can't tell if a door is locked just by looking at it, so we want the enemy to go up to the door, try it, and then change behavior if it is locked. In the example, we have the character shoulder charging the door.

We won't walk through the execution of this behavior tree in detail. Feel free to step through it yourself and make sure you understand how it would work if the door is open, if it is closed, and if it is locked.

At this stage we can start to see another common feature of behavior trees. Often they are made up of alternating layers of Sequences and Selectors. As long as the only Composite tasks we have are Sequence and Selector, it will always be possible to write the tree in this way.[1] Even with

1. The reason for this may not immediately be obvious. If you think about a tree in which a Selector has another Selector as a child—its behavior will be exactly the same as if the child's children were inserted in the parent Selector. If one of the grandchildren returns in success, then its parent immediately returns in success, and so does the grandparent. The same is true for Sequence tasks inside other Sequence tasks. This means there is no functional reason for having two levels with the same kind of Composite task. There may, however, be non-functional reasons for using another grouping such as grouping related tasks together to more clearly understand what the overall tree is trying to achieve.

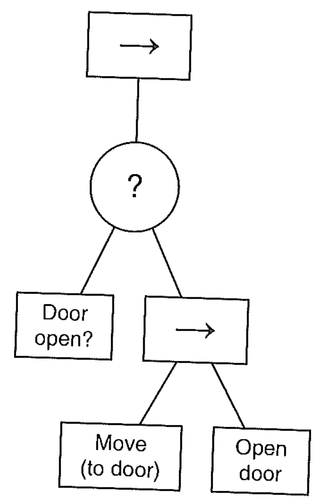

Figure 5.26 A more complicated refactored tree

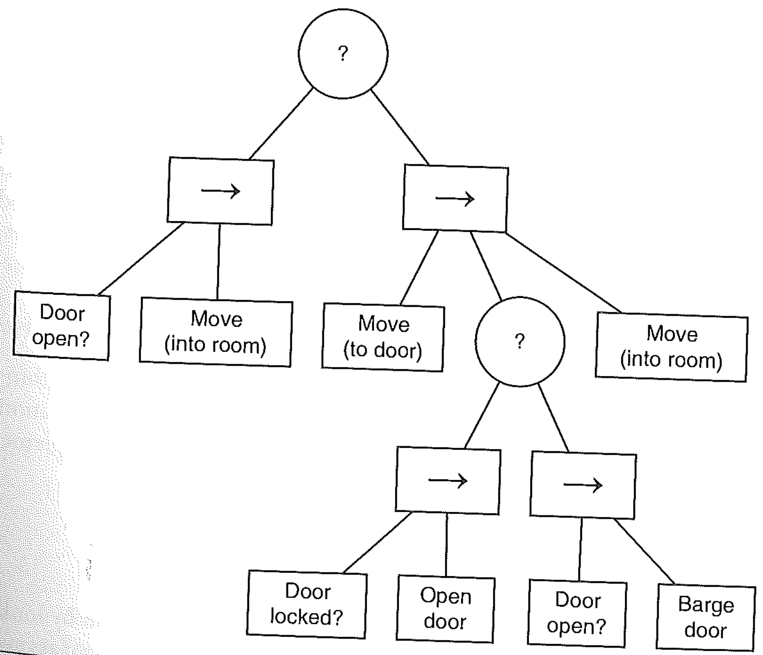

Figure 5.27 A behavior tree for a minimally acceptable enemy

the other kinds of Composite tasks we'll see later in the section, Sequence and Selector are still the most common, so this alternating structure is quite common.

We're probably just about at the point where our enemy's room-entering behavior would be acceptable in a current generation game. There's plenty more we can do here. We could add additional checks to see if there are windows to smash through. We could add behaviors to allow the character to use grenades to blow the door, we could have it pick up objects to barge the door, and we could have it pretend to leave and lie in wait for the player to emerge.

Whatever we end up doing, the process of extending the behavior tree is exactly as we've shown it here, leaving the character AI playable at each intermediate stage.

Behavior Trees and Reactive Planning

Behavior trees implement a very simple form of planning, sometimes called *reactive planning*. Selectors allow the character to try things, and fall back to other behaviors if they fail. This isn't a very sophisticated form of planning: the only way characters can think ahead is if you manually add the correct conditions to their behavior tree. Nevertheless, even this rudimentary planning can give a good boost to the believability of your characters.

The behavior tree represents all possible Actions that your character can take. The route from the top level to each leaf represents one course of action,[2] and the behavior tree algorithm searches among those courses of action in a left-to-right manner. In other words, it performs a depth-first search.

There is nothing about behavior trees or depth-first reactive planning that is unique, of course; we could do the same thing using other techniques, but typically they are much harder. The behavior of trying doors and barging through them if they are locked, for example, can be implemented using a finite state machine. But most people would find it quite unintuitive to create. You'd have to encode the fall-back behavior explicitly in the rules for state transitions. It would be fairly easy to write a script for this particular effect, but we'll soon see behavior trees that are difficult to turn into scripts without writing lots of infrastructure code to support the way behavior trees naturally work.

5.4.1 Implementing Behavior Trees

Behavior trees are made up of independent tasks, each with its own algorithm and implementation. All of them conform to a basic interface which allows them to call one another without knowing how they are implemented. In this section, we'll look at a simple implementation based on the tasks we've introduced above.

5.4.2 Pseudo-Code

Behavior trees are easy to understand at the code level. We'll begin by looking at a possible base class for a task that all nodes in the tree can inherit from. The base class specifies a method used

2. Strictly this only applies to each leaf in a Selector and the last leaves in each Sequence.

to run the task. The method should return a status code showing whether it succeeded or failed. In this implementation we will use the simplest approach and use the Boolean values True and False. The implementation of that method is normally not defined in the base class (i.e., it is a pure virtual function):

```
class Task:
    # Holds a list of the children (if any) of this task
    children

    # Always terminates with either success (True) or
    # failure (False)
    def run()
```

Here is an example of a simple task that asserts there is an enemy nearby:

```
class EnemyNear (Task):
    def run():
        if distanceToEnemy < 10:
            return True

        # Task failure, there is no enemy nearby
        return False
```

Another example of a simple task could be to play an animation:

```
class PlayAnimation (Task):
    animation_id
    speed

    def Attack(animation_id, loop=False, speed=1.0):
        this.animation = animation
        this.speed = speed

    def run():
        if animationEngine.ready():
            animationEngine.play(animation, speed)
            return True

        # Task failure, the animation could not be played.
        # The parent node will worry about the consequences
        return False
```

This task is parameterized to play one particular animation, and it checks to see if the animation engine is available before it does so.

One reason the animation engine might not be ready is if it was already busy playing a different animation. In a real game we'd want more control than this over the animation (we could still play a head-movement animation while the character was running, for example). We'll look at a more comprehensive way to implement resource-checking later in this section.

The Selector task can be implemented simply:

```
class Selector (Task):
    def run():
        for c in children:
            if c.run():
                return True

        return False
```

The Sequence node is implemented similarly:

```
class Sequence (Task):
    def run():
        for c in children:
            if not c.run():
                return False

        return True
```

Performance

The performance of a behavior tree depends on the tasks within it. A tree made up of just Selector and Sequence nodes and leaf tasks (Conditions and Actions) that are O(1) in performance and memory will be O(n) in memory and O(log n) in speed, where n is the number of nodes in the tree.

Implementation Notes

In the pseudo-code we've used Boolean values to represent the success and failure return values for tasks. In practice, it is a good idea to use a more flexible return type than Boolean values (an enum in C-based languages is ideal), because you may find yourself needing more than two return values, and it can be a serious drag to work through tens of task class implementations changing the return values.

Non-Deterministic Composite Tasks

Before we leave Selectors and Sequences for a while, it is worth looking at some simple variations of them that can make your AI more interesting and varied. The implementations above run each of their children in a strict order. The order is defined in advance by the person defining the tree. This is necessary in many cases: in our simple example above we absolutely have to check if the door is open before trying to move through it. Swapping that order would look very odd. Similarly for Selectors, there's no point trying to barge through the door if it is already open, we need to try the easy and obvious solutions first.

In some cases, however, this can lead to predictable AIs who always try the same things in the same order. In many Sequences there are some Actions that don't need to be in a particular order. If our room-entering enemy decided to smoke the player out, they might need to get matches and gasoline, but it wouldn't matter in which order as long as both matches and gasoline were in place before they tried to start the fire. If the player saw this behavior several times, it would be nice if the different characters acting this way didn't always get the components in the same order.

For Selectors, the situation can be even more obvious. Let's say that our enemy guard has five ways to gain entry. They can walk through the open door, open a closed door, barge through a locked door, smoke the player out, or smash through the window. We would want the first two of these to always be attempted in order, but if we put the remaining three in a regular Selector then the player would know what type of forced entry is coming first. If the forced entry actions normally worked (e.g., the door couldn't be reinforced, the fire couldn't be extinguished, the window couldn't be barricaded), then the player would never see anything but the first strategy in the list—wasting the AI effort of the tree builder.

These kinds of constraints are called "partial-order" constraints in the AI literature. Some parts may be strictly ordered, and others can be processed in any order. To support this in our behavior tree we use variations of Selectors and Sequences that can run their children in a random order.

The simplest would be a Selector that repeatedly tries a single child:

```
class RandomSelector (Task):
    children
    def run():
        while True:
            child = random.choice(children)
            if child.run(): return True
               return False
```

This gives us randomness but has two problems: it may try the same child more than once, even several times in a row, and it will never give up, even if all its children repeatedly fail. For these reasons, this simple implementation isn't widely useful, but it can still be used, especially in combination with the parallel task we'll meet later in this section.

A better approach would be to walk through all the children in some random order. We can do this for either Selectors or Sequences. Using a suitable random shuffling procedure, we can implement this as:

```
class NonDeterministicSelector (Task):

    children

    def run():
        shuffled = random.shuffle(children)
        for child in shuffled:
            if child.run(): return true
        return false
```

and

```
class NonDeterministicSequence (Task):

    children

    def run():
        shuffled = random.shuffle(children)
        for child in shuffled:
            if not child.run(): return false
        return true
```

In each case, just add a shuffling step before running the children. This keeps the randomness but guarantees that all the children will be run and that the node will terminate when all the children have been exhausted.

Many standard libraries do have a random shuffle routine for their vector or list data types. If yours doesn't it is fairly easy to implement Durstenfeld's shuffle algorithm:

```
def shuffle(original):
    list = original.copy()
    n = list.length
    while n > 1:
        k = random.integer_less_than(n)
        n--;
        list[n], list[k] = list[k], list[n]btPartia
    return list
```

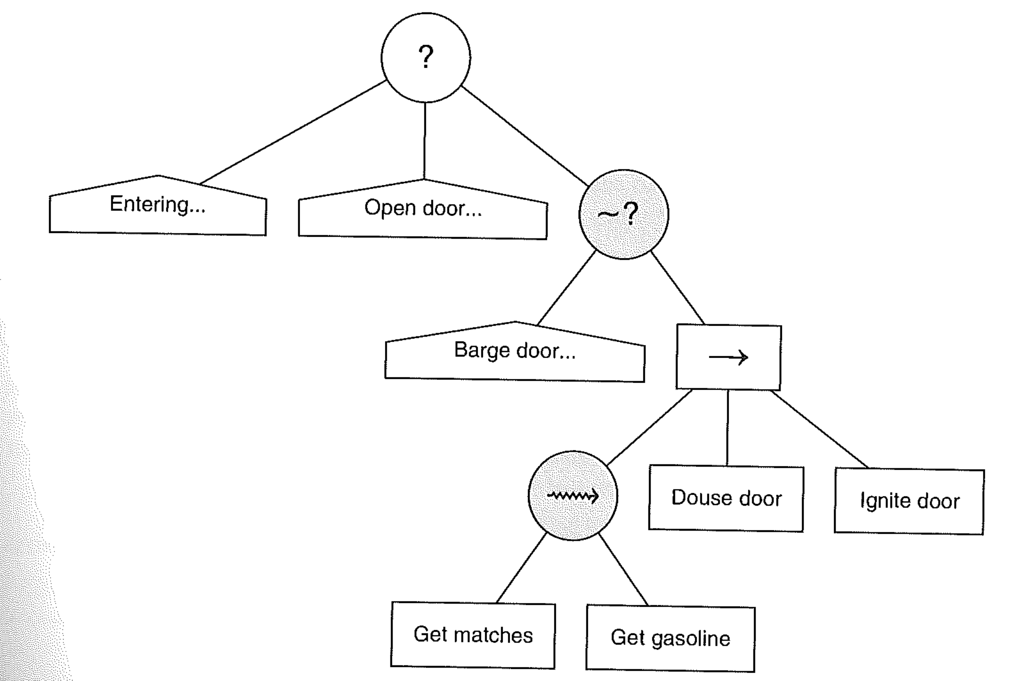

Figure 5.28 Example behavior tree with partial ordering

An implementation of this is included on the website.

So we have fully ordered Composites, and we have non-deterministic Composites. To make a partially ordered AI strategy we put them together into a behavior tree. Figure 5.28 shows the tree for the previous example: an enemy AI trying to enter a room. Non-deterministic nodes are shown with a wave in their symbol and are shaded gray.

Although the figure only shows the low-level details for the strategy to smoke the player out, each strategy will have a similar form, being made up of fixed-order Composite tasks. This is very common; non-deterministic tasks usually sit within a framework of fixed-order tasks, both above and below.

5.4.3 Decorators

So far we've met three families of tasks in a behavior tree: Conditions, Actions, and Composites. There is a fourth that is significant: Decorators.

The name "decorator" is taken from object-oriented software engineering. The decorator pattern refers to a class that wraps another class, modifying its behavior. If the decorator has the same interface as the class it wraps, then the rest of the software doesn't need to know if it is dealing with the original class or the decorator.

In the context of a behavior tree, a Decorator is a type of task that has one single child task and modifies its behavior in some way. You could think of it like a Composite task with a single child. Unlike the handful of Composite tasks we'll meet, however, there are many different types of useful Decorators.

One simple and very common category of Decorators makes a decision whether to allow their child behavior to run or not (they are sometimes called "filters"). If they allow the child behavior to run, then whatever status code it returns is used as the result of the filter. If they don't allow the child behavior to run, then they normally return in failure, so a Selector can choose an alternative action.

There are several standard filters that are useful. For example, we can limit the number of times a task can be run:

```
class Limit (Decorator)
    runLimit
    runSoFar = 0

    def run():
        if runSoFar >= runLimit:
            return False

        runSoFar++
        return child.run()
```

which could be used to make sure that a character doesn't keep trying to barge through a door that the player has reinforced.

We can use a Decorator to keep running a task until it fails:

```
class UntilFail (Decorator):
    def run():
        while True:
            result = child.run()

            if not result: break

        return True
```

We can combine this Decorator with other tasks to build up a behavior tree like the one in Figure 5.29. The code to create this behavior tree will be a sequence of calls to the task constructors that will look something like:

```
ex = Selector(Sequence(Visible,
                       UntilFail(Sequence(Conscious,
                                          Hit,
                                          Pause,
                                          Hit)),
                       Restrain),
              Selector(Sequence(Audible,
                                Creep),
                       Move))
```

The basic behavior of this tree is similar to before. The Selector node at the root, labeled (a) in the figure, will initially try its first child task. This first child is a Sequence node, labeled (b). If there is no visible enemy, then the Sequence node (b) will immediately fail and the Selector node (a) at the root will try its second child.

The second child of the root node is another Selector node, labeled (c). Its first child (d) will succeed if there is an audible enemy, in which case the character will creep. Sequence node (d) will then terminate successfully, causing Selector node (c) to also terminate successfully. This, in turn, will cause the root node (a) to terminate successfully.

So far, we haven't reached the Decorator, so the behavior is exactly what we've seen before.

In the case where there is a visible enemy, Sequence node (b) will continue to run its children, arriving at the decorator. The Decorator will execute Sequence node (e) until it fails. Node (e) can

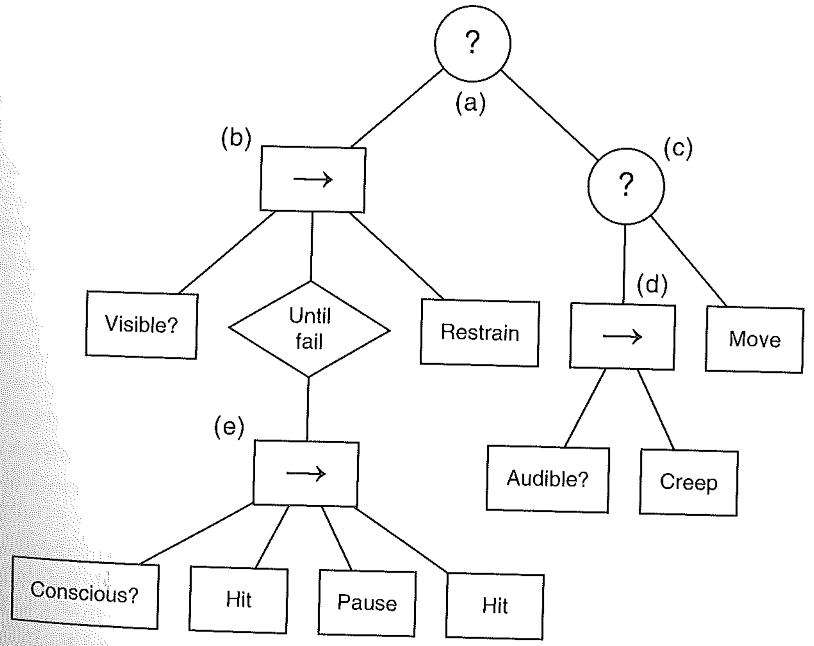

Figure 5.29 Example behavior tree

only fail when the character is no longer conscious, so the character will continually hit the enemy until it loses consciousness, after which the Selector node will terminate successfully. Sequence node (b) will then finally execute the task to tie up the unconscious enemy. Node (b) will now terminate successfully, followed by the immediate successful termination of the root node (a).

Notice that the Sequence node (e) includes a fixed repetition of hit, pause, hit. So, if the enemy happens to lose consciousness after the first hit in the sequence, then the character will still hit the subdued enemy one last time. This may give the impression of a character with a brutal personality. It is precisely this level of fine-grained control over potentially important details that is another key reason for the appeal of behavior trees.

In addition to filters that modify when and how often to call tasks, other Decorators can usefully modify the status code returned by a task:

```
class Inverter (Decorator):
    def run()
        return not child.run()
```

We've given just a few simple Decorators here. There are many more we could implement and we'll see some more below. Each of the Decorators above have inherited from a base class "Decorator". The base class is simply designed to manage its child task. In terms of our simple implementation this would be

```
class Decorator (Task):
    # Stores the child this task is decorating.
    child
```

Despite the simplicity it is a good implementation decision to keep this code in a common base class. When you come to build a practical behavior tree implementation you'll need to decide when child tasks can be set and by whom. Having the child – task management code in one place is useful. The same advice goes for Composite tasks—it is wise to have a common base class below both Selector and Sequence.

Guarding Resources with Decorators

Before we leave Decorators there is one important Decorator type that isn't as trivial to implement as the example above. We've already seen why we need it when we implemented the `PlayAnimation` task above.

Often, parts of a behavior tree need to have access to some limited resource. In the example this was the skeleton of the character. The animation engine can only play one animation on each part of the skeleton at any time. If the character's hands are moving through the reload animation, they can't be asked to wave. There are other code resources that can be scarce. We might have a limited number of pathfinding instances available. Once they are all spoken for, other characters can't use them and should choose behaviors that avoid cluing the player into the limitation.

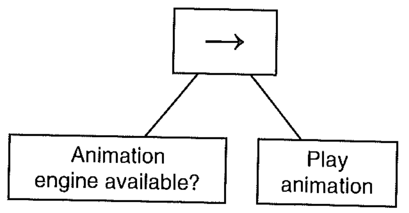

Figure 5.30 Guarding a resource using a Condition and Selector

There are other cases where resources are limited in purely game terms. There's nothing to stop us playing two audio samples at the same time, but it would be odd if they were both supposed to be exclamations from the same character. Similarly, if one character is using a wall-mounted health station, no other character should be able to use it. The same goes for cover points in a shooter, although we might be able to fit a maximum of two or three characters in some cover points and only one in others.

In each of these cases, we need to make sure that a resource is available before we run some action. We could do this in three ways:

1. By hard-coding the test in the behavior, as we did with `PlayAnimation`
2. By creating a Condition task to perform the test and using a Sequence
3. By using a Decorator to guard the resource

The first approach we've seen. The second would be to build a behavior tree that looks something like Figure 5.30. Here, the Sequence first tries the Condition. If that fails, then the whole Sequence fails. If it succeeds, then the animation action is called.

This is a completely acceptable approach, but it relies on the designer of the behavior tree creating the correct structure each time. When there are lots of resources to check, this can be overly laborious.

The third option, building a Decorator, is somewhat less error prone and more elegant.

The version of the Decorator we're going to create will use a mechanism called a *semaphore*. Semaphores are associated with parallel or multithreaded programming (and it is no coincidence that we're interested in them, as we'll see in the next section). They were originally invented by Edsger Dijkstra, of the Dijkstra algorithm fame.

Semaphores are a mechanism for ensuring that a limited resource is not over subscribed. Unlike our `PlayAnimation` example, semaphores can cope with resources that aren't limited to one single user at a time. We might have a pool of ten pathfinders, for example, meaning at most ten characters can be pathfinding at a time. Semaphores work by keeping a tally of the number of resources there are available and the number of current users. Before using the resource, a piece of code must ask the semaphore if it can "acquire" it. When the code is done it should notify the semaphore that it can be "released."

To be properly thread safe, semaphores need some infrastructure, usually depending on low-level operating system primitives for locking. Most programming languages have good libraries for semaphores, so you're unlikely to need to implement one yourself. We'll assume that semaphores are provided for us and have the following interface:

```
class Semaphore:
    # Creates a semaphore for a resource
    # with the given maximum number of users.
    def Semaphore(maximum_users)

    # Returns true if the acquisition is
    # successful, and false otherwise.
    def acquire()

    # Has no return value.
    def release()
```

With a semaphore implementation we can create our Decorator as follows:

```
class SemaphoreGuard (Decorator):

    # Holds the semaphore that we're using to
    # guard a resource.
    semaphore

    def SemaphoreGuard(semaphore):
        this.semaphore = semaphore

    def run():
        if semaphore.acquire()
            result = child.run()
            semaphore.release()
            return result
        else:
            return False
```

The Decorator returns its failure status code when it cannot acquire the semaphore. This allows a select task higher up the tree to find a different action that doesn't involve the contested resource.

Notice that the guard doesn't need to have any knowledge of the actual resource it is guarding. It just needs the semaphore. This means with this one single class, and the ability to create semaphores, we can guard any kind of resource, whether it is an animation engine, a health station, or a pathfinding pool.

In this implementation we expect the semaphore to be used in more than one guard Decorator at more than one point in the tree (or in the trees for several characters if it represents some shared resource like a cover point).

To make it easy to create and access semaphores in several Decorators, it is common to see a factory that can create them by name:

```
semaphore_hashtable = {}
def getSemaphore(name, maximum_users):
    if not semaphore_hashtable.has(name):
        semaphore_hashtable[name] =
            Semaphore(maximum_users)
    return semaphore_hashtable.get(name)
```

It is easy then for designers and level creators to create new semaphore guards by simply specifying a unique name for them. Another approach would be to pass in a name to the `SemaphoreGuard` constructor, and have it look up or create the semaphore from that name.

This Decorator gives us a powerful way of making sure that a resource isn't over-subscribed. But, so far this situation isn't very likely to arise. We've assumed that our tasks run until they return a result, so only one task gets to be running at a time. This is a major limitation, and one that would cripple our implementation. To lift it we'll need to talk about concurrency, parallel programming, and timing.

5.4.4 Concurrency and Timing

So far in this chapter we've managed to avoid the issue of running multiple behaviors at the same time. Decision trees are intended to run quickly—giving a result that can be acted upon. State machines are long-running processes, but their state is explicit, so it is easy to run them for a short time each frame (processing any transitions that are needed).

Behavior trees are different. We may have Actions in our behavior tree that take time to complete. Moving to a door, playing a door opening animation, and barging through the locked door all take time. When our game comes back to the AI on subsequent frames, how will it know what to do? We certainly don't want to start from the top of the tree again, as we might have left off midway through an elaborate sequence.

The short answer is that behavior trees as we have seen them so far are just about useless. They simply don't work unless we can assume some sort of concurrency: the ability of multiple bits of code to be running at the same time.

One approach to implementing this concurrency is to imagine each behavior tree is running in its own thread. That way an Action can take seconds to carry out: the thread just sleeps while it is happening and wakes again to return True back to whatever task was above it in the tree.

A more difficult approach is to merge behavior trees with the kind of cooperative multitasking and scheduling algorithms we will look at in Chapter 9. In practice, it can be highly wasteful to run lots of threads at the same time, and even on multi-core machines we might need to use a cooperative multitasking approach, with one thread running on each core and any number of lightweight or software threads running on each.

Although this is the most common practical implementation, we won't go into detail here. The specifics depend greatly on the platform you are targeting, and even the simplest approaches contain considerably more code for managing the details of thread management than the behavior tree algorithm.

WEB PROGRAM

The website contains an implementation of behavior trees using cooperative multitasking in ActionScript 3 for the Adobe Flash platform. Flash doesn't support native threads, so there is no alternative but to write behavior trees in this way.

To avoid this complexity we'll act as if the problem didn't exist; we'll act as if we have a multithreaded implementation with as many threads as we need.

Waiting

In a previous example we met a Pause task that allowed a character to wait a moment between Actions to strike the player. This is a very common and useful task. We can implement it by simply putting the current thread to sleep for a while:

```
class Wait (Task):
    duration

    def run():
        sleep(duration)
        return result
```

There are more complex things we can do with waiting, of course. We can use it to time out a long-running task and return a value prematurely. We could create a version of our Limit task that prevents an Action being run again within a certain time frame or one that waits a random amount of time before returning to give variation in our character's behavior.

This is just the start of the tasks we could create using timing information. None of these ideas is particularly challenging to implement, but we will not provide pseudo-code here. Some are given in the source code on the website.

The Parallel Task

In our new concurrent world, we can make use of a third Composite task. It is called "Parallel," and along with Selector and Sequence it forms the backbone of almost all behavior trees.

The Parallel task acts in a similar way to the Sequence task. It has a set of child tasks, and it runs them until one of them fails. At that point, the Parallel task as a whole fails. If all of the child tasks complete successfully, the Parallel task returns with success. In this way, it is identical to the Sequence task and its non-deterministic variations.

The difference is the way it runs those tasks. Rather than running them one at a time, it runs them all simultaneously. We can think of it as creating a bunch of new threads, one per child, and setting the child tasks off together.

When one of the child tasks ends in failure, Parallel will terminate all of the other child threads that are still running. Just unilaterally terminating the threads could cause problems, leaving the game inconsistent or failing to free resources (such as acquired semaphores). The termination procedure is usually implemented as a request rather than a direct termination of the thread. In order for this to work, all the tasks in the behavior tree also need to be able to receive a termination request and clean up after themselves accordingly.

In systems we've developed, tasks have an additional method for this:

```
class Task:
    def run()
    def terminate()
```

and the code on the website uses the same pattern. In a fully concurrent system, this terminate method will normally set a flag, and the run method is responsible for periodically checking if this flag is set and shutting down if it is. The code below simplifies this process, placing the actual termination code in the terminate method.[3]

With a suitable thread handling API, our Parallel task might look like:

```
class Parallel (Task):
    children

    # Holds all the children currently running.
    runningChildren

    # Holds the final result for our run method.
    result

    def run():
        result = undefined

        # Start all our children running
        for child in children:
            thread = new Thread()
            thread.start(runChild, child)

        # Wait until we have a result to return
```

3. This isn't the best approach in practice because the termination code will rely on the current state of the run method and should therefore be run in the same thread. The terminate method, on the other hand, will be called from our Parallel thread, so should do as little as possible to change the state of its child tasks. Setting a Boolean flag is the bare minimum, so that is the best approach.

```
        while result == undefined:
            sleep()
        return result

    def runChild(child):
        runningChildren.add(child)
        returned = child.run()
        runningChildren.remove(child)

        if returned == False:
            terminate()
            result = False

        else if runningChildren.length == 0:
            result = True

    def terminate():
        for child in runningChildren:
            child.terminate()
```

In the run method, we create one new thread for each child. We're assuming the thread's start method takes a first argument that is a function to run and additional arguments that are fed to that function. The threading libraries in a number of languages work that way. In languages such as Java where functions can't be passed to other functions, you'll need to create another class (an inner class, probably) that implements the correct interface.

After creating the threads the run method then keeps sleeping, waking only to see if the result variable has been set. Many threading systems provide more efficient ways to wait on a variable change using condition variables or by allowing one thread to manually wake another (our child threads could manually wake the parent thread when they change the value of the result). Check your system documentation for more details.

The runChild method is called from our newly created thread and is responsible for calling the child task's run method to get it to do its thing. Before starting the child, it registers itself with the list of running children. If the Parallel task gets terminated, it can terminate the correct set of still-running threads. Finally runChild checks to see if the whole Parallel task should return False, or if not whether this child is the last to finish and the Parallel should return True. If neither of these conditions holds, then the result variable will be left unchanged, and the while loop in the Parallel's run method will keep sleeping.

Policies for Parallel

We'll see Parallel in use in a moment. First, it is worth saying that here we've assumed one particular *policy* for Parallel. A policy, in this case, is how the Parallel task decides when and what to return. In our policy we return failure as soon as one child fails, and we return success when all children succeed. As mentioned above, this is the same policy as the Sequence task. Although this is the most common policy, it isn't the only one.

We could also configure Parallel to have the policy of the Selector task so it returns success when its first child succeeds and failure only when all have failed. We could also use hybrid policies, where it returns success or failure after some specific number or proportion of its children have succeeded or failed.

It is much easier to brainstorm possible task variations than it is to find a set of useful tasks that designers and level designers intuitively understand and that can give rise to entertaining behaviors. Having too many tasks or too heavily parameterized tasks is not good for productivity. We've tried in this book to stick to the most common and most useful variations, but you will come across others in studios, books, and conferences.

Using Parallel

The Parallel task is most obviously used for sets of Actions that can occur at the same time. We might, for example, use Parallel to have our character roll into cover at the same time as shouting an insult and changing primary weapon. These three Actions don't conflict (they wouldn't use the same semaphore, for example), and so we could carry them out simultaneously. This is a quite low-level use of parallel—it sits low down in the tree controlling a small sub-tree.

At a higher level, we can use Parallel to control the behavior of a group of characters, such as a fire team in a military shooter. While each member of the group gets its own behavior tree for its individual Actions (shooting, taking cover, reloading, animating, and playing audio, for example), these group Actions are contained in Parallel blocks within a higher level Selector that chooses the group's behavior. If one of the team members can't possibly carry out their role in the strategy, then the Parallel will return in failure and the Selector will have to choose another option. This is shown abstractly in Figure 5.31. The sub-trees for each character would be complex in their own right, so we haven't shown them in detail here.

Both groups of uses discussed above use Parallel to combine Action tasks. It is also possible to use Parallel to combine Condition tasks. This is particularly useful if you have certain Condition tests that take time and resources to complete. By starting a group of Condition tests together,

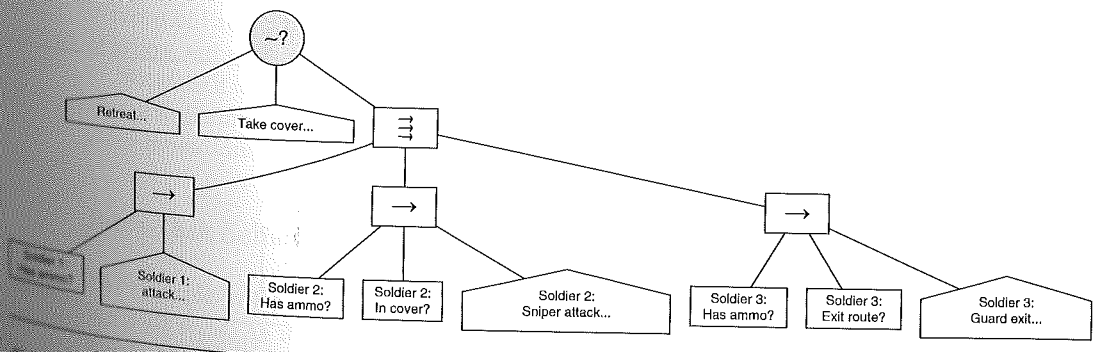

Figure 5.31 Using Parallel to implement group behavior

failures in any of them will immediately terminate the others, reducing the resources needed to complete the full package of tests.

We can do something similar with Sequences, of course, putting the quick Condition tests first to act as early outs before committing resources to more complex tests (this is a good approach for complex geometry tests such as sight testing). Often, though, we might have a series of complex tests with no clear way to determine ahead of time which is most likely to fail. In that case, placing the Conditions in a Parallel task allows any of them to fail first and interrupt the others.

The Parallel Task for Condition Checking

One final common use of the Parallel task is continually check whether certain Conditions are met while carrying out an Action. For example, we might want an ally AI character to manipulate a computer bank to open a door for the player to progress. The character is happy to continue its manipulation as long as the play guards the entrance from enemies. We could use a Parallel task to attempt an implementation as shown in Figures 5.32 and 5.33.

In both figures the Condition checks if the player is in the correct location. In Figure 5.32, we use Sequence, as before, to make sure the AI only carries out their Actions if the player is in position. The problem with this implementation is that the player can move immediately when the character begins work. In Figure 5.33, the Condition is constantly being checked. If it ever fails (because the player moves), then the character will stop what it is doing. We could embed this tree in a Selector that has the character encouraging the player to return to his post.

To make sure the Condition is repeatedly checked we have used the `UntilFail` Decorator to continually perform the checking, returning only if the Decorator fails. Based on our implementation of Parallel above, there is still a problem in Figure 5.33 which we don't have the tools to solve yet. We'll return to it shortly. As an exercise, can you follow the execution sequence of the tree and see what the problem is?

Using Parallel blocks to make sure that Conditions hold is an important use-case in behavior trees. With it we can get much of the power of a state machine, and in particular the state machine's ability to switch tasks when important events occur and new opportunities arise. Rather than events triggering transitions between states, we can use sub-trees as states and have them running in parallel with a set of conditions. In the case of a state machine, when the condition is met, the

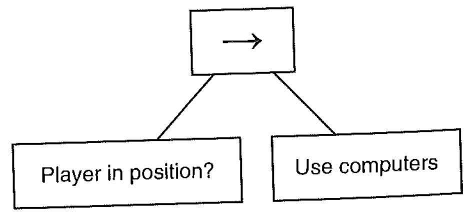

Figure 5.32 Using Sequence to enforce a Condition

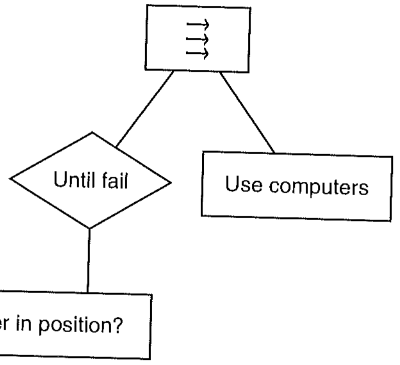

Figure 5.33 Using Parallel to keep track of Conditions

transition is triggered. With a behavior tree the behavior runs as long as the Condition is met. A state-machine-like behavior is shown using a state machine in Figure 5.34.

This is a simplified tree for the janitor robot we met earlier in the chapter. Here it has two sets of behaviors: it can be in tidy-up mode, as long there is trash to tidy, or it can be in recharging mode. Notice that each "state" is represented by a sub-tree headed by a Parallel node. The Condition for each tree is the opposite of what you'd expect for a state machine: they list the Conditions needed to stay in the state, which is the logical complement of all the conditions for all the state machine transitions.

The top Repeat and Select nodes keep the robot continually doing something. We're assuming the repeat Decorator will never return, either in success or failure. So the robot keeps trying either of its behaviors, switching between them as the criteria are met.

At this level the Conditions aren't too complex, but for more states the Conditions needed to hold the character in a state would rapidly get unwieldy. This is particularly the case if your agents need a couple of levels of alarm behaviors—behaviors that interrupt others to take immediate, reactive action to some important event in the game. It becomes counter-intuitive to code these in terms of Parallel tasks and Conditions, because we tend to think of the event *causing* a change of action, rather than the lack of the event allowing the lack of a change of action.

So, while it is technically possible to build behavior trees that show state-machine-like behavior, we can sometimes only do so by creating unintuitive trees. We'll return to this issue when we look at the limitations of behavior trees at the end of this section.

Intra-Task Behavior

The example in Figure 5.33 showed a difficulty that often arises with using Parallel alongside behavior trees. As it stands, the tree shown would never return as long as the player didn't move out of position. The character would perform its actions, then stand around waiting for

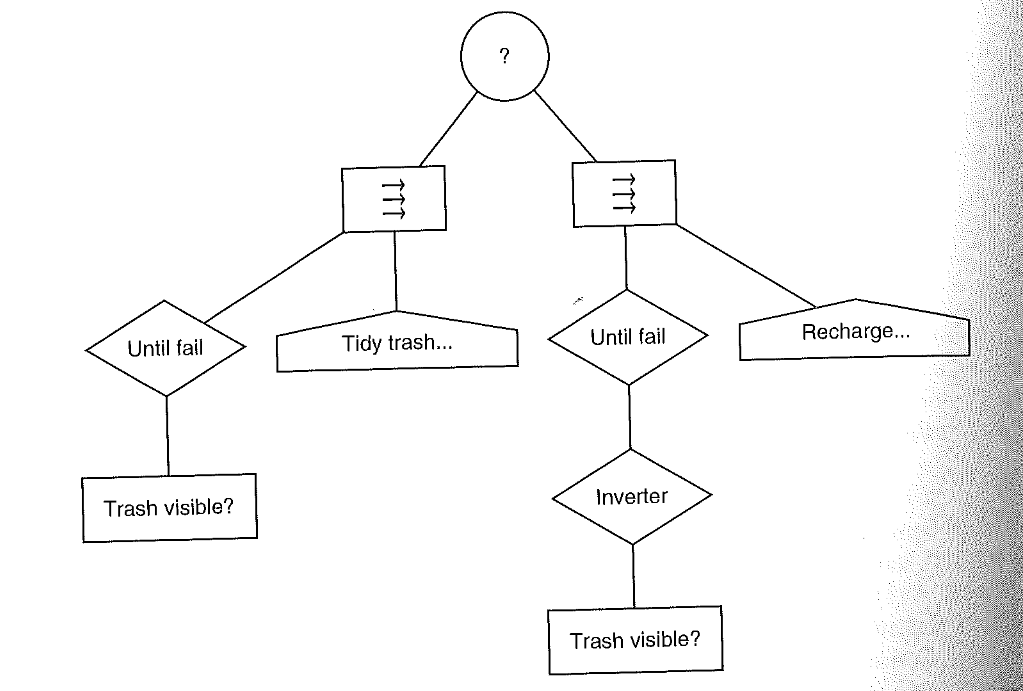

Figure 5.34 A behavior tree version of a state machine

the `UntilFail` Decorator to finish, which, of course, it won't do as long as the player stays put. We could add an Action to the end of the Sequence where the character tells the player to head for the door, or we could add a task that returns False. Both of these would certainly terminate the Parallel task, but it would terminate in failure, and any nodes above it in the tree wouldn't know if it had failed after completion or not.

To solve this issue we need behaviors to be able to affect one another directly. We need to have the Sequence end with an Action that disables the `UntilFail` behavior and has it return True. Then, the whole Action can complete.

We can do this using two new tasks. The first is a Decorator. It simply lets its child node run normally. If the child returns a result, it passes that result on up the tree. But, if the child is still working, it can be asked to terminate itself, whereupon it returns a predetermined result. We will need to use concurrency again to implement this.[4] We could define this as:

4. Some programming languages provide "continuations"—the ability to jump back to arbitrary pieces of code and to return from one function from inside another. If they sound difficult to manage, it's because they are. Unfortunately, a lot of the thread-based machinations in this section are basically trying to do the job that continuations could do natively. In a language with continuations, the Interrupter class would be much simpler.

```
class Interrupter (Decorator):
    # Is our child running?
    isRunning

    # Holds the final result for our run method.
    result

    def run():
        result = undefined

        # Start all child
        thread = new Thread()
        thread.start(runChild, child)

        # Wait until we have a result to return
        while result == undefined:
            sleep()
        return result

    def runChild(child):
        isRunning = True
        result = child.run()
        isRunning = False

    def terminate():
        if isRunning: child.terminate()

    def setResult(desiredResult):
        result = desiredResult
```

If this task looks familiar, that's because it shares the same logic as Parallel. It is the equivalent of Parallel for a single child, with the addition of a single method that can be called to set the result from an external source, which is our second task. When it is called, it simply sets a result in an external Interrupter, then returns with success.

```
class PerformInterruption (Task):
    # The interrupter we'll be interrupting
    interrupter

    # The result we want to insert.
    desiredResult
    def run():
        interrupter.setResult(desiredResult)
        return True
```

Together, these two tasks give us the ability to communicate between any two points in the tree. Effectively they break the strict hierarchy and allow tasks to interact horizontally.

With these two tasks, we can rebuild the tree for our computer-using AI character to look like Figure 5.35.

In practice there are a number of other ways in which pairs of behaviors can collaborate, but they will often have this same pattern: a Decorator and an Action. We could have a Decorator that can stop its child from being run, to be enabled and disabled by another Action task. We could have a Decorator that limits the number of times a task can be repeated but that can be reset by another task. We could have a Decorator that holds onto the return value of its child and only returns to its parent when another task tells it to. There are almost unlimited options, and behavior tree systems can easily bloat until they have very large numbers of available tasks, only a handful of which designers actually use.

Eventually this simple kind of inter-behavior communication will not be enough. Certain behavior trees are only possible when tasks have the ability to have richer conversations with one another.

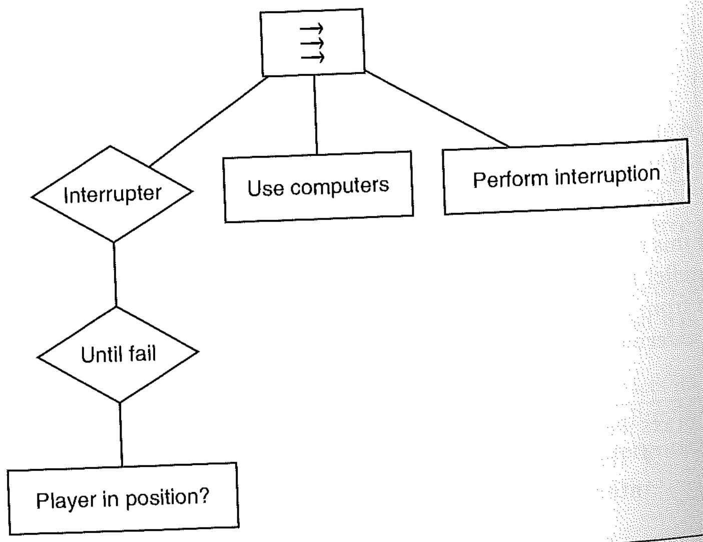

Figure 5.35 Using Parallel and Interrupter to keep track of Conditions

5.4.5 Adding Data to Behavior Trees

To move beyond the very simplest inter-behavior communication we need to allow tasks in our behavior tree to share data with one another. If you try to implement an AI using the behavior tree implementations we've seen so far you'll quickly encounter the problem of a lack of data. In our example of an enemy trying to enter a room, there was no indication of which room the character was trying to enter. We could just build big behavior trees with separate branches for each area of our level, but this would obviously be wasteful.

In a real behavior tree implementation, tasks need to know what to work on. You can think of a task as a sub-routine or function in a programming language. We might have a sub-tree that represents smoking the player out of a room, for example. If this were a sub-routine it would take an argument to control which room to smoke:

```
def smoke_out(room):
    matches = fetch_matches()
    gas = fetch_gasoline()
    douse_door(room.door, gas)
    ignite(room.door, matches)
```

In our behavior tree we need some similar mechanism to allow one sub-tree to be used in many related scenarios. Of course, the power of sub-routines is not just that they take parameters, but also that we can reuse them again and again in multiple contexts (we could use the "ignite" action to set fire to anything and use it from within lots of strategies). We'll return to the issue of reusing behavior trees as sub-routines later. For now, we'll concentrate on how they get their data.

Although we want data to pass between behavior trees, we don't want to break their elegant and consistent API. We certainly don't want to pass data into tasks as parameters to their run method. This would mean that each task needs to know what arguments its child tasks take and how to find these data.

We could parameterize the tasks at the point where they are created, since at least some part of the program will always need to know what nodes are being created, but in most implementations this won't work, either. Behavior nodes get assembled into a tree typically when the level loads (again, we'll finesse this structure soon). We aren't normally building the tree dynamically as it runs. Even implementations that do allow some dynamic tree building still rely on most of the tree being specified before the behavior begins.

The most sensible approach is to decouple the data that behaviors need from the tasks themselves. We will do this by using an external data store for all the data that the behavior tree needs. We'll call this data store a *blackboard*. Later in this chapter, in the section on blackboard architectures, we'll see a representation of such a data structure and some broader implications for its use. For now it is simply important to know that the blackboard can store any kind of data and that interested tasks can query it for the data they need.

Using this external blackboard, we can write tasks that are still independent of one another but can communicate when needed.

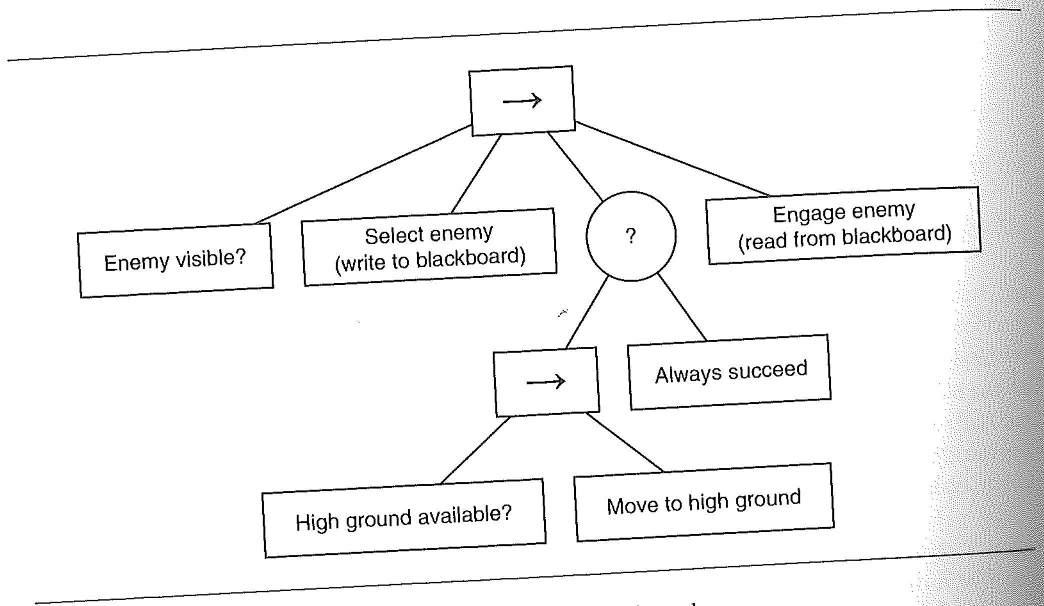

Figure 5.36 A behavior tree communicating via blackboard

In a squad-based game, for example, we might have a collaborative AI that can autonomously engage the enemy. We could write one task to select an enemy (based on proximity or a tactical analysis, for example) and another task or sub-tree to engage that enemy. The task that selects the enemy writes down the selection it has made onto the blackboard. The task or tasks that engage the enemy query the blackboard for a current enemy. The behavior tree might look like Figure 5.36. The enemy detector could write:

```
target: enemy-10f9
```

to the blackboard. The Move and Shoot At tasks would ask the blackboard for its current "target" values and use these to parameterize their behavior. The tasks should be written so that, if the blackboard had no target, then the task fails, and the behavior tree can look for something else to do.

In pseudo-code this might look like:

```
class MoveTo (Task):
    # The blackboard we're using
    blackboard

    def run():
        target = blackboard.get('target')
        if target:
            character = blackboard.get('character')
            steering.arrive(character, target)

            return True
        else:
            return False
```

where the enemy detector might look like:

```
class SelectTarget (Task):

    blackboard

    def run():
        character = blackboard.get('character')
        candidates = enemies_visible_to(character)
        if candidates.length > 0:
            target = biggest_threat(candidates, character)
            blackboard.set('target', target)
            return True
        else:
            return False
```

In both these cases we've assumed that the task can find which character it is controlling by looking that information up in the blackboard. In most games we'll want some behavior trees to be used by many characters, so each will require its own blackboard.

Some implementations associate blackboards with specific sub-trees rather than having just one for the whole tree. This allows sub-trees to have their own private data-storage area. It is shared between nodes in that sub-tree, but not between sub-trees. This can be implemented using a particular Decorator whose job is to create a fresh blackboard before it runs its child:

```
class BlackboardManager (Decorator):
    blackboard = null

    def run():
        blackboard = new Blackboard()
        result = child.run()
        delete blackboard
        return result
```

Using this approach gives us a hierarchy of blackboards. When a task comes to look up some data, we want to start looking in the nearest blackboard, then in the blackboard above that, and so on until we find a result or reach the last blackboard in the chain:

```
class Blackboard:
    # The blackboard to fall back to
    parent
```

```

data

def get(name):
    if name in data:
        return data[name]
    else if parent:
        return parent.get(name)
    else return null
```

Having blackboards fall back in this way allows blackboards to work in the same way that a programming language does. In programming languages this kind of structure would be called a "scope chain."[5]

The final element missing from our implementation is a mechanism for behavior trees to find their nearest blackboard. The easiest way to achieve this is to pass the blackboard down the tree as an argument to the run method. But didn't we say that we didn't want to change the interface? Well, yes, but what we wanted to avoid was having different interfaces for different tasks, so tasks would have to know what parameters to pass. By making all tasks accept a blackboard as their only parameter, we retain the anonymity of our tasks.

The task API now looks like this:

```
class Task:
    def run(blackbaord)
    def terminate()
```

and our `BlackboardManager` task can then simply introduce a new blackboard to its child, making the blackboard fall back to the one it was given:

```
class BlackboardManager (Decorator):
    def run(blackboard):
        new_bb = new Blackboard()
        new_bb.parent = blackboard
        result = child.run()
        free new_bb
        return result
```

5. It is worth noting that the scope chain we're building here is called a *dynamic scope chain*. In programming languages, dynamic scopes were the original way that scope chains were implemented, but it rapidly became obvious that they caused serious problems and were very difficult to write maintainable code for. Modern languages have all now moved over to static scope chains. For behavior trees, however, dynamic scope isn't a big issue and is probably more intuitive. We're not aware of any developers who have thought in such formal terms about data sharing, however, so we're not aware of anyone who has practical experience of both approaches.

Another approach to implementing hierarchies of blackboards is to allow tasks to query the task above them in the tree. This query moves up the tree recursively until it reaches a `BlackboardManager` task that can provide the blackboard. This approach keeps the original no-argument API for our task's run method, but adds a lot of extra code complexity.

Some developers use completely different approaches. Some in-house technology we know already have mechanisms in their scheduling system for passing around data along with bits of code to run. These systems can be repurposed to provide the blackboard data for a behavior tree, giving them automatic access to the data-debugging tools built into the game engine. It would be a duplication of effort to implement either scheme above in this case.

Whichever scheme you implement, blackboard data allow you to have communication between parts of your tree of any complexity. In the section on concurrency, above, we had pairs of tasks where one task calls methods on another. This simple approach to communication is fine in the absence of a richer data-exchange mechanism but should probably not be used if you are going to give your behavior tree tasks access to a full blackboard.

In that case, it is better to have them communicate by writing and reading from the blackboard rather than calling methods. Having all your tasks communicate in this way allows you to easily write new tasks to use existing data in novel ways, making it quicker to grow the functionality of your implementation.

5.4.6 Reusing Trees

In the final part of this section we'll look in more detail at how behavior trees get to be constructed in the first place, how we can reuse them for multiple characters, and how we can use sub-trees multiple times in different contexts. These are three separate but important elements to consider. They have related solutions, but we'll consider each in turn.

Instantiating Trees

Chances are, if you've taken a course on object-oriented programming, you were taught the dichotomy between instances of things and classes of things. We might have a class of soda machines, but the particular soda machine in the office lobby is an instance of that class. Classes are abstract concepts; instances are the concrete reality. This works for many situations, but not all. In particular, in game development, we regularly see situations where there are three, not two, levels of abstraction. So far in this chapter we've been ignoring this distinction, but if we want to reliably instantiate and reuse behavior trees we have to face it now.

At the first level we have the classes we've been defining in pseudo-code. They represent abstract ideas about how to achieve some task. We might have a task for playing an animation, for example, or a condition that checks whether a character is within range of an attack.

At the second level we have instances of these classes arranged in a behavior tree. The examples we've seen so far consist of instances of each task class at a particular part of the tree. So, in the behavior tree example of Figure 5.29, we have two Hit tasks. These are two instances of the Hit class. Each instance has some parameterization: the `PlayAnimation` task gets told what animation to play, the `EnemyNear` condition gets given a radius, and so on.

But now we're meeting the third level. A behavior tree is a way of defining a set of behaviors, but those behaviors can belong to any number of characters in the game at the same or different times. The behavior tree needs to be instantiated for a particular character at a particular time.

This three layers of abstraction don't map easily onto most regular class-based languages, and you'll need to do some work to make this seamless. There are a few approaches:

1. Use a language that supports more than two layers of abstraction.
2. Use a cloning operation to instantiate trees for characters.
3. Create a new intermediate format for the middle layer of abstraction.
4. Use behavior tree tasks that don't keep local state and use separate state objects.

The first approach is probably not practical. There is another way of doing object orientation (OO) that doesn't use classes. It is called *prototype-based* object orientation, and it allows you to have any number of different layers of abstraction. Despite being strictly more powerful than class-based OO, it was discovered much later, and unfortunately has had a hard time breaking into developers' mindsets. The only widespread language to support it is JavaScript.[6]

The second approach is the easiest to understand and implement. The idea is that, at the second layer of abstraction, we build a behavior tree from the individual task classes we've defined. We then use that behavior tree as an "archetype"; we keep it in a safe place and never use it to run any behaviors on. Any time we need an instance of that behavior tree we take a copy of the archetype and use the copy. That way we are getting all of the configuration of the tree, but we're getting our own copy. One method of achieving this is to have each task have a clone method that makes a copy of itself. We can then ask the top task in the tree for a clone of itself and have it recursively build us a copy. This presents a very simple API but can cause problems with fragmented memory. The code on the website uses this approach, as does the pseudo-code examples below. We've chosen this for simplicity only, not to suggest it is the right way to do this. In some languages, "deep-copy" operations are provided by the built-in libraries that can do this for us. Even if we don't have a deep copy, writing one can potentially give better memory coherence to the trees it creates.

WEB PROGRAM

Approach three is useful when the specification for the behavior tree is held in some data format. This is common—the AI author uses some editing tool that outputs some data structure saying what nodes should be in the behavior tree and what properties they should have. If we have this specification for a tree we don't need to keep a whole tree around as an archetype; we can just

6. The story of prototype-based OO in JavaScript isn't a pretty one. Programmers taught to think in class-based OO can find it hard to adjust, and the web is littered with people making pronouncements about how JavaScript's object-oriented model is "broken." This has been so damaging to JavaScript's reputation that the most recent versions of the JavaScript specification have retrofitted the class-based model. ActionScript 3, which is an implementation of that recent specification, leans heavily this way, and Adobe's libraries for Flash and Flex effectively lock you into Java-style class-based programming, wasting one of the most powerful and flexible aspects of the language.

store the specification, and build an instance of it each time it is needed. Here, the only classes in our system are the original task classes, and the only instances are the final behavior trees. We've effectively added a new kind of intermediate layer of abstraction in the form of our custom data structure, which can be instantiated when needed.

Approach four is somewhat more complicated to implement but has been reported by some developers. The idea is that we write all our tasks so they never hold any state related to a specific use of that task for a specific character. They can hold any data at the middle level of abstraction: things that are the same for all characters at all times, but specific to that behavior tree. So, a Composite node can hold the list of children it is managing, for example (as long as we don't allow children to be dynamically added or removed at runtime). But, our Parallel node can't keep track of the children that are currently running. The current list of active children will vary from time to time and from character to character. These data do need to be stored somewhere, however, otherwise the behavior tree couldn't function. So this approach uses a separate data structure, similar to our blackboard, and requires all character-specific data to be stored there. This approach treats our second layer of abstraction as the instances and adds a new kind of data structure to represent the third layer of abstraction. It is the most efficient, but it also requires a lot of bookkeeping work.

This three-layer problem isn't unique to behavior trees, of course. It arises any time we have some base classes of objects that are then configured, and the configurations are then instantiated. Allowing the configuration of game entities by non-programmers is so ubiquitous in large-scale game development (usually it is called "data-driven" development) that this problem keeps coming up, so much so that it is possible that whatever game engine you're working with already has some tools built in to cope with this situation, and the choice of approach we've outlined above becomes moot—you go with whatever the engine provides. If you are the first person on your project to hit the problem, it is worth really taking time to consider the options and build a system that will work for everyone else, too.

Reusing Whole Trees

With a suitable mechanism to instantiate behavior trees, we can build a system where many characters can use the same behavior.

During development, the AI authors create the behavior trees they want for the game and assign each one a unique name. A factory function can then be asked for a behavior tree matching a name at any time.

We might have a definition for our generic enemy character:

```
Enemy Character (goon):
    model = ''enemy34.model''
    texture = ''enemy34-urban.tex''
    weapon = pistol-4
    behavior = goon-behavior
```

When we create a new goon, the game requests a fresh goon behavior tree. Using the cloning approach to instantiating behavior trees, we might have code that looks like:

```
def createBehaviorTree(type):
    archetype = behavior_tree_library[type]
    return archetype.clone()
```

Clearly not onerous code! In this example, we're assuming the behavior tree library will be filled with the archetypes for all the behavior trees that we might need. This would normally be done during the loading of the level, making sure that only the trees that might be needed in that level are loaded and instantiated into archetypes.

Reusing Sub-trees

With our behavior library in place, we can use it for more than simply creating whole trees for characters. We can also use it to store named sub-trees that we intend to use in multiple contexts. Take the example shown in Figure 5.37.

This shows two separate behavior trees. Notice that each of them has a sub-tree that is designed to engage an enemy. If we had tens of behavior trees for tens of different kinds of character, it would be incredibly wasteful to have to specify and duplicate these sub-trees. It would be great to reuse them. By reusing them we'd also be able to come along later and fix bugs or add more sophisticated functionality and know that every character in the game instantly benefits from the update.

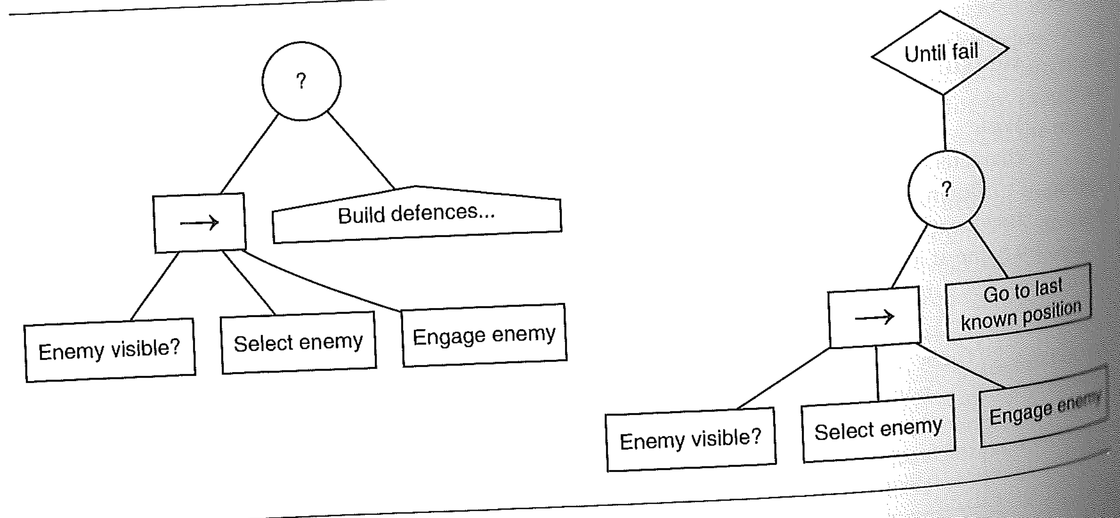

Figure 5.37 Common sub-trees across characters

We can certainly store partial sub-trees in our behavior tree library. Because every tree has a single root task, and because every task looks just the same, our library doesn't care whether it is storing sub-trees or whole trees. The added complication for sub-trees is how to get them out of the library and embedded in the full tree.

The simplest solution is to do this lookup when you create a new instance of your behavior tree. To do this you add a new "reference" task in your behavior tree that tells the game to go and find a named sub-tree in the library. This task is never run—it exists just to tell the instantiation mechanism to insert another sub-tree at this point.

For example, this class is trivial to implement using recursive cloning:

```
class SubtreeReference (Task):

    # What named subtree are we referring to.
    reference_name

    def run():
        throw Error("This task isn't meant to be run!")

    def clone():
        return createBehaviorTree(reference_name)
```

In this approach our archetype behavior tree contains these reference nodes, but as soon as we instantiate our full tree it replaces itself with a copy of the sub-tree, built by the library.

Notice that the sub-tree is instantiated when the behavior tree is created, ready for a character's use. In memory-constrained platforms, or for games with thousands of AI characters, it may be worth holding off on creating the sub-tree until it is needed, saving memory in cases where parts of a large behavior tree are rarely used. This may be particularly the case where the behavior tree has a lot of branches for special cases: how to use a particular rare weapon, for example, or what to do if the player mounts some particularly clever ambush attempt. These highly specific sub-trees don't need to be created for every character, wasting memory; instead, they can be created on demand if the rare situation arises.

We can implement this using a Decorator. The Decorator starts without a child but creates that child when it is first needed:

```
class SubtreeLookupDecorator (Decorator):

    subtree_name

    def SubtreeLookupDecorator(subtree_name):
        this.subtree_name = subtree_name
        this.child = null

    def run():
        if child == null:
            child = createBehaviorTree(subtree_name)
        return child.run()
```

Obviously we could extend this further to delete the child and free the memory after it has been used, if we really want to keep the behavior tree as small as possible.

With the techniques we've now met, we have the tools to build a comprehensive behavior tree system with whole trees and specific components that can be reused by lots of characters in the game. There is a lot more we can do with behavior trees, in addition to tens of interesting tasks we could write and lots of interesting behaviors we could build. Behavior trees are certainly an exciting technology, but they don't solve all of our problems.

5.4.7 Limitations of Behavior Trees

Over the last five years, behavior trees have come from nowhere to become something of the flavor of the month in game AI. There are some commentators who see them as a solution to almost every problem you can imagine in game AI. It is worth being a little cautious. These fads do come and go. Understanding what behavior trees are bad at is as important as understanding where they excel.

We've already seen a key limitation of behavior trees. They are reasonably clunky when representing the kind of state-based behavior that we met in the previous section. If your character transitions between types of behavior based on the success or failure of actions, however (so they get mad when they can't do something, for example), then behavior trees work fine. But it is much harder if you have a character who needs to respond to external events—interrupting a patrol route to suddenly go into hiding or to raise an alarm, for example—or a character than needs to switch strategies when its ammo is looking low. Notice that we're not claiming those behaviors can't be implemented in behavior trees, just that it would be cumbersome to do so.

Because behavior trees make it more difficult to think and design in terms of states, AI based solely on behavior trees tends to avoid these kinds of behavior. If you look at a behavior tree created by an artist or level designer, they tend to avoid noticeable changes of character disposition or alarm behavior. This is a shame, since those cues are simple and powerful and help raise the level of the AI.

We can build a hybrid system, of course, where characters have multiple behavior trees and use a state machine to determine which behavior tree they are currently running. Using the approach of having behavior tree libraries that we saw above, this provides the best of both worlds. Unfortunately, it also adds considerable extra burden to the AI authors and toolchain developers, since they now need to support two kinds of authoring: state machines and behavior trees.

An alternative approach would be to create tasks in the behavior tree that behave like state machines—detecting important events and terminating the current sub-tree to begin another. This merely moves the authoring difficulty, however, as we still need to build a system for AI authors to parameterize these relatively complex tasks.

Behavior trees on their own have been a big win for game AI, and developers will still be exploring their potential for a few years. As long as they are pushing forward the state of the art, we suspect that there will not be a strong consensus on how best to avoid these limitations, with developers experimenting with their own approaches.

5.5 Fuzzy Logic

So far the decisions we've made have been very cut and dried. Conditions and decisions have been true or false, and we haven't questioned the dividing line. Fuzzy logic is a set of mathematical techniques designed to cope with gray areas.

Imagine we're writing AI for a character moving through a dangerous environment. In a finite state machine approach, we could choose two states: "Cautious" and "Confident." When the character is cautious, it sneaks slowly along, keeping an eye out for trouble. When the character is confident, it walks normally. As the character moves through the level, it will switch between the two states. This may appear odd. We might think of the character getting gradually braver, but this isn't shown until suddenly it stops creeping and walks along as if nothing had ever happened.

Fuzzy logic allows us to blur the line between cautious and confident, giving us a whole spectrum of confidence levels. With fuzzy logic we can still make decisions like "walk slowly when cautious," but both "slowly" and "cautious" can include a range of degrees.

5.5.1 A Warning

Fuzzy logic is relatively popular in the games industry and is used in several games. For that reason, we have decided to include a section on it in this book. However, you should be aware that fuzzy logic has, for valid reasons, been largely discredited within the mainstream academic AI community.

You can read more details in Russell and Norvig [2002] but the executive summary is that it is always better to use probability to represent any kind of uncertainity. The slightly longer version is that it has been proven (a long time ago, as it turns out) that if you play any kind of betting game then any player who is not basing their decisions on probability theory can expect to eventually lose his money. The reason is that flaws in any other theory of uncertainty, besides probability theory, can potentially be exploited by an opponent.

Part of the reason why fuzzy logic ever became popular was the perception that using probabilistic methods can be slow. With the advent of Bayes nets and other graphical modeling techniques, this is no longer such an issue. While we won't explicitly cover Bayes nets in this book, we will look at various other related approaches such as Markov systems.

5.5.2 Introduction to Fuzzy Logic

This section will give a quick overview of the fuzzy logic needed to understand the techniques in this chapter. Fuzzy logic itself is a huge subject, with many subtle features, and we don't have the changing every frame, it may not be sensible to keep a complete record of every value it has ever taken.

Any piece of data in the database can then be queried, and the expert system shell will return the audit trail of how the data got there and how the current value came to be set. This information can be recursive. If the data we are interested in came from a rule, we can ask where the matches came from that triggered that rule. This process can continue until we are left with just the items of data that were added by the game (or were there from the start).

In an expert system this is used to justify the decisions that the system makes. If the expert system is controlling a factory and chooses to shut down a production line, then the justification system can give the reasons for its decision.

In a game context, we don't need to justify decisions to the player, but during testing, it is often very useful to have a mechanism for justifying the behavior of a character. Rule-based systems can be so much more complicated than the previous decision making techniques in this chapter. Finding out the detailed and long-term causes of a strange-looking behavior can save days of debugging.

We've built an expert system shell specifically for inclusion in a game. We added a justification system late in the development cycle after a bout of hair-pulling problems. The difference in debugging power was dramatic. A sample portion of the output is shown below (the full output was around 200 lines long). Unfortunately, because the code was developed commercially, we are not able to include this application on the website.

```
Carnage XS. V104 2002-9-12.
JUSTIFICATION FOR <Action: Grenade (2.2,0.5,2.1)>

<Action: grenade ?target>
 FROM RULE: flush-nest
 BINDINGS: target = (2.2,0.5,2.1)
 CONDITIONS:

  <Visible: heavy-weapon <ptr008F8850> at (2.2,0.5,2.1)>
   FROM RULE: covered-by-heavy-weapon
   BINDINGS: ?weapon = <ptr008F8850>
   CONDITIONS:
    <Ontology: machine-gun <ptr008F8850>>
     FROM FACT: <Ontology: machine-gun <ptr008F8850>>
    <Location: <ptr008F8850> at (312.23, 0.54, 12.10)>
     FROM FACT: <Location: <ptr008F8850> at (2.2,0.5,2.1)>

  <Visible: enemy-units in group>
  ...
```

To make sure the final game wasn't using lots of memory to store the firing data, the justifica-tion code was conditionally compiled so it didn't end up in the final product.

5.8.10 Where Next

The rule-based systems in this section represent the most complex non-learning decision makers we'll cover in this book. A full Rete-implementation with justification and rule set support is a formidable programming task that can support incredible sophistication of behavior. It can support more advanced AI than any seen in current-generation games (providing someone was capable of writing enough good rules). Likewise, GOAP is well ahead of the current state of the art and is the cutting edge of AI being explored in several big studios.

The remainder of this chapter looks at the problem of making decisions from some different angles, examining ways to combine different decision makers together, to script behaviors directly from code, and to execute actions that are requested from any decision making algorithm.

5.9 Blackboard Architectures

A blackboard system isn't a decision making tool in its own right. It is a mechanism for coordinating the actions of several decision makers.

The individual decision making systems can be implemented in any way: from a decision tree to an expert system or even to learning tools such as the neural networks we'll meet in Chapter 7. It is this flexibility that makes blackboard architectures appealing.

In the AI literature, blackboard systems are often large and unwieldy, requiring lots of management code and complicated data structures. For this reason they have something of a bad reputation among game AI programmers. At the same time, many developers implement AI systems that use the same techniques without associating them with the term "blackboard architecture."

5.9.1 The Problem

We would like to be able to coordinate the decision making of several different techniques. Each technique may be able to make suggestions as to what to do next, but the final decision can only be made if they cooperate.

We may have a decision making technique specializing in targeting enemy tanks, for example. It can't do its stuff until a tank has been selected to fire at. A different kind of AI is used to select a firing target, but that bit of AI can't do the firing itself. Similarly, even when the target tank is selected, we may not be in a position where firing is possible. The targeting AI needs to wait until a route-planning AI can move to a suitable firing point.

We could simply put each bit of AI in a chain. The target selector AI chooses a target, the movement AI moves into a firing position, and the ballistics AI calculates the firing solution. This approach is very common but doesn't allow for information to pass in the opposite direction. If the ballistics AI calculates that it cannot make an accurate shot, then the targeting AI may need to calculate a new solution. On the other hand, if the ballistics AI can work out a shot, then there is no need to even consider the movement AI. Obviously, whatever objects are in the way do not affect the shell's trajectory.

We would like a mechanism whereby each AI can communicate freely without requiring all the communication channels to be set up explicitly.

5.9.2 The Algorithm

The basic structure of a blackboard system has three parts: a set of different decision making tools (called *experts* in blackboard-speak), a blackboard, and an arbiter. This is illustrated in Figure 5.54.

The blackboard is an area of memory that any expert may use to read from and write to. Each expert needs to read and write in roughly the same language, although there will usually be messages on the blackboard that not everyone can understand.

Each expert looks at the blackboard and decides if there's anything on it that they can use. If there is, they ask to be allowed to have the chalk and board eraser for a while. When they get control they can do some thinking, remove information from the blackboard, and write new information, as they see fit. After a short time, the expert will relinquish control and allow other experts to have a go.

The arbiter picks which expert gets control at each go. Experts need to have some mechanism of indicating that they have something interesting to say. The arbiter chooses one at a time and gives it control. Often, none or only one expert wants to take control, and the arbiter is not required.

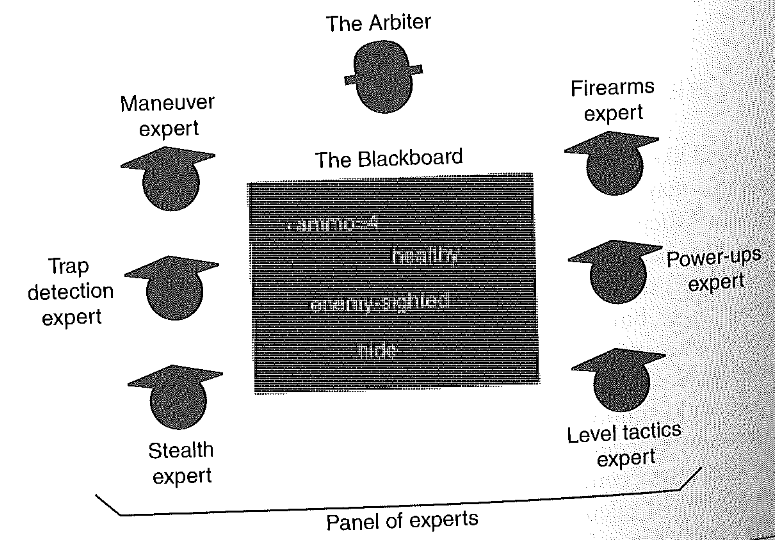

Figure 5.54 Blackboard architecture

The algorithm works in iterations:

1. Experts look at the board and indicate their interest.
2. The arbiter selects an expert to have control.
3. The expert does some work, possibly modifying the blackboard.
4. The expert voluntarily relinquishes control.

The algorithm used by the arbiter can vary from implementation to implementation. The simple and common approach we will use asks the experts to indicate how useful they think they can be in the form of a numeric insistence value. The arbiter can then simply pick the expert with the highest insistence value. In the case of a tie, a random expert is selected.

Extracting an Action

Suggested actions can be written to the blackboard by experts in the same way that they write any other information. At the end of an iteration (or multiple iterations if the system is running for longer), actions placed on the blackboard can be removed and carried out using the action execution techniques at the end of this chapter.

Often, an action can be suggested on the blackboard before it has been properly thought through. In our tank example, the targeting expert may post a "fire at tank 15" action on the board. If the algorithm stopped at that point, the action would be carried out without the ballistic and movement experts having had a chance to agree.

A simple solution is to store the potential action along with a set of agreement flags. An action on the blackboard is only carried out if all relevant experts have agreed to it. This does not have to be every expert in the system, just those who would be capable of finding a reason not to carry the action out.

In our example the "fire at tank 15" action would have one agreement slot: that of the ballistics expert. Only if the ballistics expert has given the go ahead would the action be carried out. The ballistics expert may refuse to give the go ahead and instead either delete the action or add a new action "move into firing position for tank 15." With the "fire at tank 15" action still on the blackboard, the ballistics expert can wait to agree to it until the firing position is reached.

5.9.3 Pseudo-Code

The `blackboardIteration` function below takes as input a blackboard and a set of experts. It returns a list of actions on the blackboard that have been passed for execution. The function acts as the arbiter, following the highest insistence algorithm given above.

```
def blackboardIteration(blackboard, experts):

  # Go through each expert for their insistence
```

```
    bestExpert = None
    highestInsistence = 0
    for expert in experts:
      # Ask for the expert's insistence
      insistence = expert.getInsistence(blackboard)

      # Check against the highest value so far
      if insistence > highestInsistence:
        highestInsistence = insistence
        bestExpert = expert

    # Make sure somebody insisted
    if bestExpert:

      # Give control to the most insistent expert
      bestExpert.run(blackboard)

    # Return all passed actions from the blackboard
    return blackboard.passedActions
```

5.9.4 Data Structures and Interfaces

The `blackboardIteration` function relies on three data structures: a blackboard consisting of entries and a list of experts.

The `Blackboard` has the following structure:

```
class Blackboard:
  entries
  passedActions
```

It has two components: a list of blackboard entries and a list of ready-to-execute actions. The list of blackboard entries isn't used in the arbitration code above and is discussed in more detail later in the section on blackboard language. The actions list contains actions that are ready to execute (i.e., they have been agreed upon by every expert whose permission is required). It can be seen as a special section of the blackboard: a to-do list where only agreed-upon actions are placed.

More complex blackboard systems also add meta-data to the blackboard that controls its execution, keeps track of performance, or provides debugging information. Just as for rule-based systems, we can also add data to hold an audit trail for entries: which expert added them and when.

Other blackboard systems hold actions as just another entry on the blackboard itself, without a special section. For simplicity, we've elected to use a separate list; it is the responsibility of each expert to write to the "actions" section when an action is ready to be executed and to keep unconfirmed actions off the list. This makes it much faster to execute actions. We can simply work through this list rather than searching the main blackboard for items that represent confirmed actions.

Experts can be implemented in any way required. For the purpose of being managed by the arbiter in our code, they need to conform to the following interface:

```
class Expert:
  def getInsistence(blackboard)
  def run(blackboard)
```

The `getInsistence` function returns an insistence value (greater than zero) if the expert thinks it can do something with the blackboard. In order to decide on this, it will usually need to have a look at the contents of the blackboard. Because this function is called for each expert, the blackboard should not be changed at all from this function. It would be possible, for example, for an expert to return some instance, only to have the interesting stuff removed from the blackboard by another expert. When the original expert is given control, it has nothing to do.

The `getInsistence` function should also run as quickly as possible. If the expert takes a long time to decide if it can be useful, then it should always claim to be useful. It can spend the time working out the details when it gets control. In our tanks example, the firing solution expert may take a while to decide if there is a way to fire. In this case, the expert simply looks on the blackboard for a target, and if it sees one, it claims to be useful. It may turn out later that there is no way to actually hit this target, but that processing is best done in the `run` function when the expert has control.

The `run` function is called when the arbiter gives the expert control. It should carry out the processing it needs, read and write to the blackboard as it sees fit, and return. In general, it is better for an expert to take as little time as possible to run. If an expert requires lots of time, then it can benefit from stopping in the middle of its calculations and returning a very high insistence on the next iteration. This way the expert gets its time split into slices, allowing the rest of the game to be processed. Chapter 9 has more details on this kind of scheduling and time slicing.

The Blackboard Language

So far we haven't paid any attention to the structure of data on the blackboard. More so than any of the other techniques in this chapter, the format of the blackboard will depend on the application. Blackboard architectures can be used for steering characters, for example, in which case the blackboard will contain three-dimensional (3D) locations, combinations of maneuvers, or animations. Used as a decision making architecture, it might contain information about the game state, the position of enemies or resources, and the internal state of a character.

There are general features to bear in mind, however, that go some way toward a generic blackboard language. Because the aim is to allow different bits of code to talk to each other seamlessly, information on the blackboard needs at least three components: value, type identification, and semantic identification.

The value of a piece of data is self-explanatory. The blackboard will typically have to cope with a wide range of different data types, however, including structures. It might contain health values expressed as an integer and positions expressed as a 3D vector, for example.

Because the data can be in a range of types, its content needs to be identified. This can be a simple type code. It is designed to allow an expert to use the appropriate type for the data (in C/C++ this is normally done by typecasting the value to the appropriate type). Blackboard entries could achieve this by being polymorphic: using a generic `Datum` base class with sub-classes for `FloatDatum`, `Vector3DDatum`, and so on, or with runtime-type information (RTTI) in a language such as C++, or the sub-classes containing a type identifier. It is more common, however, to explicitly create a set of type codes to identify the data, whether or not RTTI is used.

The type identifier tells an expert what format the data are in, but it doesn't help the expert understand what to do with it. Some kind of semantic identification is also needed. The semantic identifier tells each expert what the value *means*. In production blackboard systems this is commonly implemented as a string (representing the name of the data). In a game, using lots of string comparisons can slow down execution, so some kind of magic number is often used.

A blackboard item may therefore look like the following:

```
struct BlackboardDatum:
  id
  type
  value
```

The whole blackboard consists of a list of such instances.

In this approach complex data structures are represented in the same way as built-in types. All the data for a character (its health, ammo, weapon, equipment, and so on) could be represented in one entry on the blackboard or as a whole set of independent values.

We could make the system more general by adopting an approach similar to the one we used in the rule-based system. Adopting a hierarchical data representation allows us to effectively expand complex data types and allows experts to understand parts of them without having to be hard-coded to manipulate the type. In languages such as Java, where code can examine the structure of a type, this would be less important. In C++, it can provide a lot of flexibility. An expert could look for just the information on a weapon, for example, without caring if the weapon is on the ground, in a character's hand, or currently being constructed.

While many blackboard architectures in non-game AI follow this approach, using nested data to represent their content, we have not seen it used in games. Hierarchical data tend to be associated with rule-based systems and flat lists of labeled data with blackboard systems (although the two approaches overlap, as we'll see below).

5.9.5 Performance

The blackboard arbiter uses no memory and runs in O(n) time, where n is the number of experts. Often, each expert needs to scan through the blackboard to find an entry that it might be interested in. If the list of entries is stored as a simple list, this takes O(m) time for each expert, where m is the number of entries in the blackboard. This can be reduced to almost O(1) time if the blackboard entries are stored in some kind of hash. The hash must support lookup based on the semantics of the data, so an expert can quickly tell if something interesting is present.

The majority of the time spent in the `blackboardIteration` function should be spent in the `run` function of the expert who gains control. Unless a huge number of experts is used (or they are searching through a large linear blackboard), the performance of each `run` function is the most important factor in the overall efficiency of the algorithm.

5.9.6 Other Things Are Blackboard Systems

When we described the blackboard system, we said it had three parts: a blackboard containing data, a set of experts (implemented in any way) that read and write to the blackboard, and an arbiter to control which expert gets control. It is not alone in having these components, however.

Rule-Based Systems

Rule-based systems have each of these three elements: their database contains data, each rule is like an expert—it can read from and write to the database, and there is an arbiter that controls which rule gets to fire. The triggering of rules is akin to experts registering their interest, and the arbiter will then work in the same way in both cases.

This similarity is no coincidence. Blackboard architectures were first put forward as a kind of generalization of rule-based systems: a generalization in which the rules could have any kind of trigger and any kind of rule.

A side effect of this is that if you intend to use both a blackboard system and a rule-based system in your game, you may need to implement only the blackboard system. You can then create "experts" that are simply rules: the blackboard system will be able to manage them.

The blackboard language will have to be able to support the kind of rule-based matching you intend to perform, of course. But, if you are planning to implement the data format needed in the rule-based system we discussed earlier, then it will be available for use in more flexible blackboard applications.

If your rule-based system is likely to be fairly stable, and you are using the Rete matching algorithm, then the correspondence will break down. Because the blackboard architecture is a super-set of the rule-based system, it cannot benefit from optimizations specific to rule handling.

Finite State Machines

Less obviously, finite state machines are also a subset of the blackboard architecture (actually they are a subset of a rule-based system and, therefore, of a blackboard architecture). The blackboard is replaced by the single state. Experts are replaced by transitions, determining whether to act based on external factors, and rewriting the sole item on the blackboard when they do. In the state machines in this chapter we have not mentioned an arbiter. We simply assumed that the first triggered transition would fire. This is simply the first-applicable arbitration algorithm.

Other arbitration strategies are possible in any state machine. We can use dynamic priorities, randomized algorithms, or any kind of ordering. They aren't normally used because the state machine is designed to be simple; if a state machine doesn't support the behavior you are looking for, it is unlikely that arbitration will be the problem.

State machines, rule-based systems, and blackboard architectures form a hierarchy of increasing representational power and sophistication. State machines are fast, easy to implement, and restrictive, while blackboard architectures can often appear far too general to be practical. The general rule, as we saw in the introduction, is to use the simplest technique that supports the behavior you are looking for.

5.10 Scripting

A significant proportion of the decision making in games uses none of the techniques described so far in this chapter. In the early and mid-1990s, most AI was hard-coded using custom written code to make decisions. This is fast and works well for small development teams when the programmer is also likely to be designing the behaviors for game characters. It is still the dominant model for platforms with modest development needs (i.e., last-generation handheld consoles prior to PSP, PDAs, and mobile phones).

As production became more complex, there arose a need to separate the content (the behavior designs) from the engine. Level designers were empowered to design the broad behaviors of characters. Many developers moved to use the other techniques in this chapter. Others continued to program their behaviors in a full programming language but moved to a scripting language separate from the main game code. Scripts can be treated as data files, and if the scripting language is simple enough level designers or technical artists can create the behaviors.

An unexpected side effect of scripting language support is the ability for players to create their own character behavior and to extend the game. Modding is an important financial force in PC games (it can extend their full-price shelf life beyond the eight weeks typical of other titles), so much so that most triple-A titles have some kind of scripting system included. On consoles the economics is less clear cut. Most of the companies we worked with who had their own internal game engine had some form of scripting language support.

While we are unconvinced about the use of scripts to run top-notch character AI, they have several important applications: in scripting the triggers and behavior of game levels (which keys open which doors, for example), for programming the user interface, and for rapidly prototyping character AI.

This section provides a brief primer for supporting a scripting language powerful enough to run AI in your game. It is intentionally shallow and designed to give you enough information to either get started or decide it isn't worth the effort. Several excellent websites are available comparing existing languages, and a handful of texts cover implementing your own language from scratch.

5.10.1 Language Facilities

There are a few facilities that a game will always require of its scripting language. The choice of language often boils down to trade-offs between these concerns.

Speed

Scripting languages for games need to run as quickly as possible. If you intend to use a lot of scripts for character behaviors and events in the game level, then the scripts will need to execute as part of the main game loop. This means that slow-running scripts will eat into the time you need to render the scene, run the physics engine, or prepare audio.

Most languages can be anytime algorithms, running over multiple frames (see Chapter 9 for details). This takes the pressure off the speed to some extent, but it can't solve the problem entirely.

Compilation and Interpretation

Scripting languages are broadly interpreted, byte-compiled, or fully compiled, although there are many flavors of each technique.

Interpreted languages are taken in as text. The interpreter looks at each line, works out what it means, and carries out the action it specifies.

Byte-compiled languages are converted from text to an internal format, called *byte code*. This byte code is typically much more compact than the text format. Because the byte code is in a format optimized for execution, it can be run much faster.

Byte-compiled languages need a compilation step; they take longer to get started, but then run faster. The more expensive compilation step can be performed as the level loads but is usually performed before the game ships.

The most common game scripting languages are all byte-compiled. Some, like Lua, offer the ability to detach the compiler and not distribute it with the final game. In this way all the scripts can be compiled before the game goes to master, and only the compiled versions need to be included with the game. This removes the ability for users to write their own script, however.

Fully compiled languages create machine code. This normally has to be linked into the main game code, which can defeat the point of having a separate scripting language. We do know of one developer, however, with a very neat runtime-linking system that can compile and link machine code from scripts at runtime. In general, however, the scope for massive problems with this approach is huge. We'd advise you to save your hair and go for something more tried and tested.

Extensibility and Integration

Your scripting language needs to have access to significant functions in your game. A script that controls a character, for example, needs to be able to interrogate the game to find out what it can see and then let the game know what it wants to do as a result.

The set of functions it needs to access is rarely known when the scripting language is implemented or chosen. It is important to have a language that can easily call functions or use classes in your main game code. Equally, it is important for the programmers to be able to expose new functions or classes easily when the script authors request it.

Some languages (Lua being the best example) put a very thin layer between the script and the rest of the program. This makes it very easy to manipulate game data from within scripts, without having a whole set of complicated translations.

Re-Entrancy

It is often useful for scripts to be re-entrant. They can run for a while, and when their time budget runs out they can be put on hold. When a script next gets some time to run, it can pick up where it left off.

It is often helpful to let the script yield control when it reaches a natural lull. Then a scheduling algorithm can give it more time, if it has it available, or else it moves on. A script controlling a character, for example, might have five different stages (examine situation, check health, decide movement, plan route, and execute movement). These can all be put in one script that yields between each section. Then each will get run every five frames, and the burden of the AI is distributed.

Not all scripts should be interrupted and resumed. A script that monitors a rapidly changing game event may need to run from its start at every frame (otherwise, it may be working on incorrect information). More sophisticated re-entrancy should allow the script writer to mark sections as uninterruptible.

These subtleties are not present in most off-the-shelf languages, but can be a massive boon if you decide to write your own.

5.10.2 Embedding

Embedding is related to extensibility. An embedded language is designed to be incorporated into another program. When you run a scripting language from your workstation, you normally run a dedicated program to interpret the source code file. In a game, the scripting system needs to be controlled from within the main program. The game decides which scripts need to be run and should be able to tell the scripting language to process them.

5.10.3 Choosing a Language

A huge range of scripting languages is available, and many of them are released under licences that are suitable for inclusion in a game. Traditionally, most scripting languages in games have been created by developers specifically for their needs. In the last few years there has been a growing interest in off-the-shelf languages.

Some commercial game engines include scripting language support (**Unreal** and **Quake** by id Software, for example). Other than these complete solutions, most existing languages used

in games were not originally designed for this purpose. They have associated advantages and disadvantages that need to be evaluated before you make a choice.

Advantages

Off-the-shelf languages tend to be more complete and robust than a language you write yourself. If you choose a fairly mature language, like those described below, you are benefiting from a lot of development time, debugging effort, and optimization that has been done by other people.

When you have deployed an off-the-shelf language, the development doesn't stop. A community of developers is likely to be continuing work on the language, improving it and removing bugs. Many open source languages provide web forums where problems can be discussed, bugs can be reported, and code samples can be downloaded. This ongoing support can be invaluable in making sure your scripting system is robust and as bug free as possible.

Many games, especially on the PC, are written with the intention of allowing consumers to edit their behavior. Customers building new objects, levels, or whole mods can prolong a game's shelf life. Using a scripting language that is common allows users to learn the language easily using tutorials, sample code, and command line interpreters that can be downloaded from the web. Most languages have newsgroups or web forums where customers can get advice without calling the publisher's help line.

Disadvantages

When you create your own scripting language, you can make sure it does exactly what you want it to. Because games are sensitive to memory and speed limitations, you can put only the features you need into the language. As we've seen with re-entrancy, you can also add features that are specific to game applications and that wouldn't normally be included in a general purpose language.

By the same token, when things go wrong with the language, your staff knows how it is built and can usually find the bug and create a workaround faster.

Whenever you include third-party code into your game, you are losing some control over it. In most cases, the advantages outweigh the lack of flexibility, but for some projects control is a must.

Open-Source Languages

Many popular game scripting languages are released under open-source licences.

Open-source software is released under a licence that gives users rights to include it in their own software without paying a fee. Some open-source licences require that the user release the newly created product open source. These are obviously not suitable for commercial games.

Open-source software, as its name suggests, also allows access to see and change the source code. This makes it easy to attract studios by giving you the freedom to pull out any extraneous or inefficient code. Some open-source licences, even those that allow you to use the language in commercial products, require that you release any modifications to the language itself. This may be an issue for your project.

Whether or not a scripting language is open source, there are legal implications of using the language in your project. Before using any outside technology in a product you intend to distribute (whether commercial or not), you should always consult a good intellectual property lawyer. This book cannot properly advise you on the legal implications of using a third-party language. The following comments are intended as an indication of the kinds of things that might cause concern. There are many others.

With nobody selling you the software, nobody is responsible if the software goes wrong. This could be a minor annoyance if a difficult-to-find bug arises during development. It could be a major legal problem, however, if your software causes your customer's PC to wipe its hard drive. With most open-source software, you are responsible for the behavior of the product.

When you licence technology from a company, the company normally acts as an insulation layer between you and being sued for breach of copyright or breach of patent. A researcher, for example, who develops and patents a new technique has rights to its commercialization. If the same technique is implemented in a piece of software, without the researcher's permission, he may have cause to take legal action. When you buy software from a company, it takes responsibility for the software's content. So, if the researcher comes after you, the company that sold you the software is usually liable for the breach (it depends on the contract you sign).

When you use open-source software, nobody is licencing the software to you, and because you didn't write it, you don't know if part of it was stolen or copied. Unless you are very careful, you will not know if it breaks any patents or other intellectual property rights. The upshot is that you could be liable for the breach.

You need to make sure you understand the legal implications of using "free" software. It is not always the cheapest or best choice, even though the up-front costs are very low. Consult a lawyer before you make the commitment.

5.10.4 A Language Selection

Everyone has a favorite language, and trying to back a single pre-built scripting language is impossible. Read any programming language newsgroup to find endless "my language is better than yours" flame wars.

Even so, it is a good idea to understand which languages are the usual suspects and what their strengths and weaknesses are. Bear in mind that it is usually possible to hack, restructure, or rewrite existing languages to get around their obvious failings. Many (probably most) commercial games developers using scripting languages do this. The languages described below are discussed in their out-of-the-box forms.

We'll look at three languages in the order we would personally recommend them: Lua, Scheme, and Python.

Lua

Lua is a simple procedural language built from the ground up as an embedding language. The design of the language was motivated by extensibility. Unlike most embedded languages, this isn't

limited to adding new functions or data types in C or C++. The way the Lua language works can also be tweaked.

Lua has a small number of core libraries that provide basic functionality. Its relatively featureless core is part of the attraction, however. In games you are unlikely to need libraries to process anything but maths and logic. The small core is easy to learn and very flexible.

Lua does not support re-entrant functions. The whole interpreter (strictly the "state" object, which encapsulates the state of the interpreter) is a C++ object and is completely re-entrant. Using multiple state objects can provide some re-entrancy support, at the cost of memory and lack of communication between them.

Lua has the notion of "events" and "tags." Events occur at certain points in a script's execution: when two values are added together, when a function is called, when a hash table is queried, or when the garbage collector is run, for example. Routines in C++ or Lua can be registered against these events. These "tag" routines are called when the event occurs, allowing the default behavior of Lua to be changed. This deep level of behavior modification makes Lua one of the most adjustable languages you can find.

The event and tag mechanism is used to provide rudimentary object-oriented support (Lua isn't strictly object oriented, but you can adjust its behavior to get as close as you like to it), but it can also be used to expose complex C++ types to Lua or for tersely implementing memory management.

Another Lua feature beloved by C++ programmers is the "userdata" data type. Lua supports common data types, such as floats, ints, and strings. In addition, it supports a generic "userdata" with an associated sub-type (the "tag"). By default, Lua doesn't know how to do anything with userdata, but by using tag methods, any desired behavior can be added. Userdata is commonly used to hold a C++ instance pointer. This native handling of pointers can cause problems, but often means that far less interface code is needed to make Lua work with game objects.

For a scripting language, Lua is at the fast end of the scale. It has a very simple execution model that at peak is fast. Combined with the ability to call C or C++ functions without lots of interface code, this means that real-world performance is impressive.

The syntax for Lua is recognizable for C and Pascal programmers. It is not the easiest language to learn for artists and level designers, but its relative lack of syntax features means it is achievable for keen employees.

Despite its documentation being poorer than for the other two main languages here, Lua is the most widely used pre-built scripting language in games. The high-profile switch of Lucas Arts from its internal SCUMM language to Lua motivated a swathe of developers to investigate its capabilities.

We started using Lua several years ago, and it is easy to see why it is rapidly becoming the *de facto* standard for game scripting. To find out more, the best source of information is the Lua book *Programming in Lua* [Ierusalimschy, 2006], which is also available free online.

Scheme and Variations

Scheme is a scripting language derived from LISP, an old language that was used to build most of the classic AI systems prior to the 1990s (and many since, but without the same dominance).

The first thing to notice about Scheme is its syntax. For programmers not used to LISP, Scheme can be difficult to understand.

Brackets enclose function calls (and almost everything is a function call) and all other code blocks. This means that they can become very nested. Good code indentation helps, but an editor that can check enclosing brackets is a must for serious development. For each set of brackets, the first element defines what the block does; it may be an arithmetic function:

```
(+ a 0.5)
```

or a flow control statement:

```
(if (> a 1.0) (set! a 1.0))
```

This is easy for the computer to understand but runs counter to our natural language. Non-programmers and those used to C-like languages can find it hard to think in Scheme for a while.

Unlike Lua and Python, there are literally hundreds of versions of Scheme, not to mention other LISP variants suitable for use as an embedded language. Each variant has its own trade-offs, which make it difficult to make generalizations about speed or memory use. At their best, however (minischeme and tinyscheme come to mind), they can be very, very small (minischeme is less than 2500 lines of C code for the complete system, although it lacks some of the more exotic features of a full scheme implementation) and superbly easy to tweak. The fastest implementations can be as fast as any other scripting language, and compilation can typically be much more efficient than other languages (because the LISP syntax was originally designed for easy parsing).

Where Scheme really shines, however, is its flexibility. There is no distinction in the language between code and data, which makes it easy to pass around scripts within Scheme, modify them, and then execute them later. It is no coincidence that most notable AI programs using the techniques in this book were originally written in LISP.

We have used Scheme a lot, enough to be able to see past its awkward syntax (many of us had to learn LISP as an AI undergraduate). Professionally, we have never used Scheme unmodified in a game (although we know at least one studio that has), but we have built more languages based on Scheme than on any other language (six to date and one more on the way). If you plan to roll your own language, we would strongly recommend you first learn Scheme and read through a couple of simple implementations. It will probably open your eyes as to how easy a language can be to create.

Python

Python is an easy-to-learn, object-oriented scripting language with excellent extensibility and embedding support. It provides excellent support for mixed language programming, including the ability to transparently call C and C++ from Python. Python has support for re-entrant functions as part of the core language from version 2.2 onward (called *Generators*).

Python has a huge range of libraries available for it and has a very large base of users. Python users have a reputation for helpfulness, and the `comp.lang.python` newsgroup is an excellent source of troubleshooting and advice.

Python's major disadvantages are speed and size. Although significant advances in execution speed have been made over the last few years, it can still be slow. Python relies on hash table lookup (by string) for many of its fundamental operations (function calls, variable access, object-oriented programming). This adds lots of overhead.

While good programming practice can alleviate much of the speed problem, Python also has a reputation for being large. Because it has much more functionality than Lua, it is larger when linked into the game executable.

Python 2.X and further Python 2.3 releases added a lot of functionality to the language. Each additional release fulfilled more of Python's promise as a software engineering tool, but by the same token made it less attractive as an embedded language for games. Earlier versions of Python were much better in this regard, and developers working with Python often prefer previous releases.

Python often appears strange to C or C++ programmers, because it uses indentation to group statements, just like the pseudo-code in this book.

This same feature makes it easier to learn for non-programmers who don't have brackets to forget and who don't go through the normal learning phase of not indenting their code.

Python is renowned for being a very readable language. Even relatively novice programmers can quickly see what a script does. More recent additions to the Python syntax have damaged this reputation greatly, but it still seems to be somewhat above its competitors.

Of the scripting languages we have worked with, Python has been the easiest for level designers and artists to learn. On a previous project we needed to use this feature but were frustrated by the speed and size issues. Our solution was to roll our own language (see the section below) but use Python syntax.

Other Options

There is a whole host of other possible languages. In our experience each is either completely unused in games (to the best of our knowledge) or has significant weaknesses that make it a difficult choice over its competitors. To our knowledge, none of the languages in this section has seen commercial use as an in-game scripting tool. As usual, however, a team with a specific bias and a passion for one particular language can work around these limitations and get a usable result.

Tcl

Tcl is a very well-used embeddable language. It was designed to be an integration language, linking multiple systems written in different languages. Tcl stands for Tool Control Language.

Most of Tcl's processing is based on strings, which can make execution very slow. Another major drawback is its bizarre syntax, which takes some getting used to, and unlike Scheme it doesn't hold the promise of extra functionality in the end. Inconsistencies in the syntax (such as argument passing by value or by name) are more serious flaws for the casual learner.

Java

Java is becoming ubiquitous in many programming domains. Because it is a compiled language, however, its use as a scripting language is restricted. By the same token, however, it can be fast. Using JIT compiling (the byte code gets turned into native machine code before execution), it can approach C++ for speed. The execution environment is very large, however, and there is a sizeable memory footprint.

It is the integration issues that are most serious, however. The Java Native Interface (that links Java and C++ code) was designed for extending Java, rather than embedding it. It can therefore be difficult to manage.

JavaScript

JavaScript is a scripting language designed for web pages. It really has nothing to do with Java, other than its C++-like syntax.

There isn't one standard JavaScript implementation, so developers who claim to use JavaScript are most likely rolling their own language based on the JavaScript syntax.

The major advantage of JavaScript is that it is known by many designers who have worked on the web. Although its syntax loses lots of the elegance of Java, it is reasonably usable.

Ruby

Ruby is a very modern language with the same elegance of design found in Python, but its support for object-oriented idioms is more ingrained. It has some neat features that make it able to manipulate its own code very efficiently. This can be helpful when scripts have to call and modify the behavior of other scripts. It is not highly re-entrant from the C++ side, but it is very easy to create sophisticated re-entrancy from within Ruby.

It is very easy to integrate with C code (not as easy as Lua, but easier than Python, for example). Ruby is only beginning to take off, however, and hasn't reached the audience of the other languages in this chapter. It hasn't been used (modified or otherwise) in any game we have heard about. One weakness is its lack of documentation, although that may change rapidly as it gains wider use. It's a language we have resolved to follow closely for the next few years.

5.10.5 Rolling Your Own

Most game scripting languages are custom written for the job at hand. While this is a long and complex procedure for a single game, the added control can be beneficial in the long run. Studios developing a whole series of games using the same engine will effectively spread the development effort and cost over multiple titles.

Regardless of the look and capabilities of the final language, scripts will pass through the same process on their way to being executed: all scripting languages must provide the same basic set of elements. Because these elements are so ubiquitous, tools have been developed and refined to make it easy to build them.

There is no way we can give a complete guide to building your own scripting language in this book. There are many other books on language construction (although, surprisingly, there aren't any good books we know of on creating a scripting, rather than a fully compiled, language). This section looks at the elements of scripting language construction from a very high level, as an aid to understanding rather than implementation.

The Stages of Language Processing

Starting out as text in a text file, a script typically passes through four stages: tokenization, parsing, compiling, and interpretation.

The four stages form a pipeline, each modifying its input to convert it into a format more easily manipulated. The stages may not happen one after another. All steps can be interlinked, or sets of stages can form separate phases. The script may be tokenized, parsed, and compiled offline, for example, for interpretation later.

Tokenizing

Tokenizing identifies elements in the text. A text file is just a sequence of characters (in the sense of ASCII characters!). The tokenizer works out which bytes belong together and what kind of group they form.

A string of the form:

```
a = 3.2;
```

can be split into six tokens:

`a`	text
`<space>`	whitespace
`=`	equality operator
`<space>`	whitespace
`3.2`	floating point number
`;`	end of statement identifier

Notice that the tokenizer doesn't work out how these fit together into meaningful chunks; that is the job of the parser.

The input to the tokenizer is a sequence of characters. The output is a sequence of tokens.

Parsing

The meaning of a program is very hierarchical: a variable name may be found inside an assignment statement, found inside an IF-statement, which is inside a function body, inside a class definition, inside a namespace declaration, for example. The parser takes the sequence of tokens, identifies the role each plays in the program, and identifies the overall hierarchical structure of the program.

The line of code:

```
if (a < b) return;
```

converted into the token sequence:

```
keyword(if), whitespace, open-brackets, name(a), operator(<),
name(b), close-brackets, whitespace, keyword(return),
end-of-statement
```

is converted by the parser into a structure such as that shown in Figure 5.55.

This hierarchical structure is known as the *parse tree*, or sometimes a *syntax tree* or *abstract syntax tree* (AST for short). Parse trees in a full language may be more complex, adding additional layers for different types of symbol or for grouping statements together. Typically, the parser will output additional data along with the tree, most notably a symbol table that identifies what variable or function names have been used in the code. This is not essential. Some languages

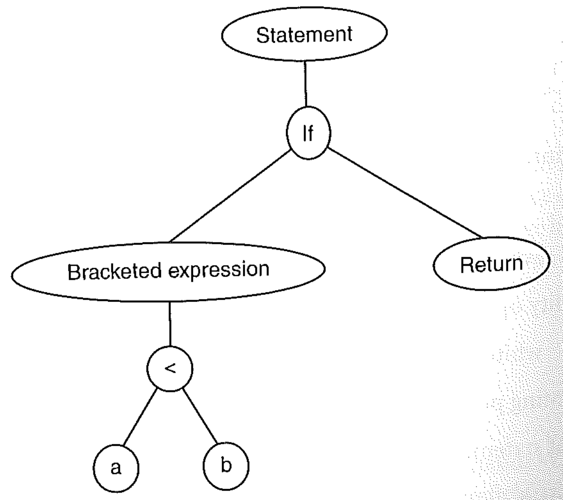

Figure 5.55 A parse tree

look up variable names dynamically when they run in the interpreter (Python does this, for example).

Syntax errors in the code show up during parsing because they make it impossible for the parser to build an output.

The parser doesn't work out how the program should be run; that is the job of the compiler.

Compiling

The compiler turns the parse tree into byte code that can be run by the interpreter. Byte code is typically sequential binary data.

Non-optimizing compilers typically output byte code as a literal translation of the parse tree. So a code such as:

```
a = 3;
if (a < 0) return 1;
else return 0;
```

could get compiled into:

```
  load 3
  set-value-of a
  get-value-of a
  compare-with-zero
  if-greater-jump-to LABEL
  load 1
  return
LABEL:
  load 0
  return
```

Optimizing compilers try to understand the program and make use of prior knowledge to make the generated code faster. An optimizing compiler may notice that a must be 3 when the IF-statement above is encountered. It can therefore generate:

```
load 3
set-value-of a
load 0
return
```

Building an efficient compiler is well beyond the scope of this book. Simple compilers are not difficult to build, but don't underestimate the effort and experience needed to build a good solution. There are many hundreds of home-brewed languages out there with pathetic compilers. We've seen it very many times.

Tokenizing, parsing, and compiling are often done offline and are usually called "compiling," even though the process includes all three stages. The generated byte code can then be stored and interpreted at runtime. The parser and compiler can be large, and it makes sense not to have the overhead of these modules in the final game.

Interpreting

The final stage of the pipeline runs the byte code. In a compiler for a language such as C or C++, the final product will be machine instructions that can be directly run by the processor. In a scripting language, you often need to provide services (such as re-entrancy and secure execution) that are not easily achieved with machine language.

The final byte code is run on a "virtual machine." This is effectively an emulator for a machine that has never existed in hardware.

You decide the instructions that the machine can execute, and these are the byte code instructions. In the previous example,

```
load <value>
set-value-of <variable>
get-value-of <variable>
compare-with-zero
if-greater-jump-to <location>
return
```

are all byte codes.

Your byte code instructions don't have to be limited to those that might be seen in real hardware, either. For example, there may be a byte code for "turn the data into a set of game coordinates": the kind of instruction that makes your compiler easier to create but that no real hardware would ever need.

Most virtual machines consist of a big switch statement in C: each byte code has a short bit of C code that gets executed when the byte code is reached in the interpreter. So the "add" byte code has a bit of C/C++ code that performs the addition operation. Our conversion example may have two or three lines of C++ to perform the required conversion and copy the results back into the appropriate place.

Just-in-Time Compiling

Because of the highly sequential nature of byte code, it is possible to write a virtual machine that is very fast at running it. Even though it is still interpreted, it is many times faster than interpreting the source language a line at a time.

It is possible to remove the interpretation step entirely, however, by adding an additional compilation step. Some byte code can be compiled into the machine language of the target hardware. When this is done in the virtual machine, just before execution, it is called *just-in-time* (JIT) compiling. This is not common in game scripting languages but is a mainstay of languages such as Java and Microsoft's .NET byte code.

Tools: A Quick Look at Lex and Yacc

Lex and Yacc are the two principal tools used in building tokenizers and parsers, respectively. Each has many different implementations and is provided with most UNIX distributions (versions are available for other platforms, too). The Linux variants we have most often used are Flex and Bison.

To create a tokenizer with Lex, you tell it what makes up different tokens in your language. What constitutes a number, for example (even this differs from language to language—compare 0.4f to 1.2e−9). It produces C code that will convert the text stream from your program into a stream of token codes and token data. The software it generates is almost certainly better and faster than that you could write yourself.

Yacc builds parsers. It takes a representation of the grammar of your language—what tokens make sense together and what large structures can be made up of smaller ones, for example. This grammar is given in a set of rules that show how larger structures are made from simpler ones or from tokens, for example:

```
assignment: NAME '=' expression;
```

This rule tells Yacc that when it finds a `NAME` token, followed by an equals sign, followed by a structure it knows as an expression (for which there will be a recognizer rule elsewhere), then it knows it has an assignment.

Yacc also generates C code. In most cases, the resulting software is as good as or better than you would create manually, unless you are experienced with writing parsers. Unlike Lex, the final code can often be further optimized if speed is absolutely critical. Fortunately, for game scripting the code can usually be compiled when the game is not being played, so the slight inefficiency is not important.

Both Lex and Yacc allow you to add your own C code to the tokenizing or parsing software. There isn't a *de facto* standard tool for doing the compiling, however. Depending on the way the language will behave, this will vary widely. It is very common to have Yacc build an AST for the compiler to work on, however, and there are various tools to do this, each with their own particular output format.

Many Yacc-based compilers don't need to create a syntax tree. They can create byte code output from within the rules using C code written into the Yacc file. As soon as an assignment is found, for example, its byte code is output. It is very difficult to create optimizing compilers this way, however. So if you intend to create a professional solution, it is worth heading directly for a parse tree of some kind.

5.10.6 Scripting Languages and Other AI

If you make the effort to build a scripting language into your game, chances are it will run most of your AI. Most of the other techniques in this book will not need to be coded into your game. This can seem appealing at first, but you should still have a general purpose language for the heavy lifting.

It is a successful approach taken by many commercial studios. Typically, some extra AI is provided (normally a pathfinding engine, for example) for very processor intensive needs.

But in our opinion it misses the point of established AI techniques. They exist because they are elegant solutions to behavior problems, not because programming in C++ is inconvenient. Even if you go to a scripting language, you have to think about the algorithms used in the character scripts. Writing *ad hoc* code in scripts can rapidly become as difficult to debug as writing it in C++ (more so in fact, since scripting languages often have much less mature debugging tools).

Several developers we know have fallen into this trap, assuming that a scripting language means they don't need to think about the way characters are implemented. Even if you are using a scripting language, we'd advise you to think about the architecture and algorithms you use in those scripts. It may be that the script can implement one of the other techniques in this chapter, or it may be that a separate dedicated C++ implementation would be more practical alongside or instead of the scripting language.

5.11 Action Execution

Throughout this chapter we've talked about actions as if it were clear what they were. Everything from decision trees to rule-based systems generates actions, and we've avoided being clear on what format they might take.

Many developers don't work with actions as a distinct concept. The result of each decision making technique is simply a snippet of code that calls some function, tweaks some state variable, or asks a different bit of the game (AI, physics, rendering, whatever) to perform some task.

On the other hand, it can be beneficial to handle a character's actions through a central piece of code. It makes the capabilities of a character explicit, makes the game more flexible (you can add and remove new types of actions easily), and can aid hugely in debugging the AI. This calls for a distinct concept for actions, with a distinct algorithm to manage and run them.

This section looks at actions in general and how they can be scheduled and executed through a general action manager. The discussion about how different types of actions are executed is relevant, even to projects that don't use a central execution manager.

5.11.1 Types of Action

We can divide the kind of actions that result from AI decisions into four flavors: state change actions, animations, movement, and AI requests.

State change actions are the simplest kind of action, simply changing some piece of the game state. It is often not directly visible to the player. A character may change the firing mode of its weapon, for example, or use one of its health packs. In most games, these changes only have associated animations or visual feedback when the player carries them out. For other characters, they simply involve a change in a variable somewhere in the game's state.

Animations are the most primitive kind of visual feedback. This might be a particle effect when a character casts a spell or a quick shuffle of the hands to indicate a weapon reload. Often, combat is simply a matter of animation, whether it be the recoil from a gun, sheltering behind a raised shield, or a lengthy combo sword attack.

Animations may be more spectacular. We might request an in-engine cutscene, sending the camera along some predefined track and coordinating the movement of many characters.

Actions may also require the character to make some **movement** through the game level. Although it isn't always clear where an animation leaves off and movement begins, we are thinking about larger scale movement here. A decision maker that tells a character to run for cover, to collect a nearby power-up, or to chase after an enemy is producing a movement action.

In Chapter 3 on movement algorithms we saw the kind of AI that converts this kind of high-level movement request (sometimes called *staging*) into primitive actions. These primitive actions (e.g., apply such-and-such a force in such-and-such a direction) can then be passed to the game physics, or an animation controller, to execute.

Although these movement algorithms are typically considered part of AI, we're treating them here as if they are just a single action that can be executed. In a game, they will be executed by calling the appropriate algorithms and passing the results onto the physics or animation layer. In other words, they will usually be implemented in terms of the next type of action.

In **AI requests** for complex characters, a high-level decision maker may be tasked with deciding which lower level decision maker to use. The AI controlling one team in a real-time strategy game, for example, may decide that it is time to build. A different AI may actually decide which building gets to be constructed. In squad-based games, several tiers of AI are possible, with the output from one level guiding the next level (we will cover specific tactical and strategic AI techniques in Chapter 6).

A Single Action

The action that is output from a decision making tool may combine any or all of these flavors. In fact, most actions have at least two of these components to them.

Reloading a weapon involves both a state change (replenishing ammo in the gun from the overall total belonging to the character) and an animation (the hand shuffle). Running for cover may involve an AI request (to a pathfinder), movement (following the path), and animation (waving hands over head in panic). Deciding to build something may involve more AI (choosing what to build) and animation (the construction yard's chimney starts smoking).

Actions involving any animation or movement take time. State changes may be immediate, and an AI request can be honored straight away, but most actions will take some time to complete.

A general action manager will need to cope with actions that take time; we can't simply complete the action in an instant.

Many developers engineer their AI so that the decision maker keeps scheduling the same action every frame (or every time it is called) until the action is completed. This has the advantage that the action can be interrupted at any time (see the next section), but it means that the decision making system is being constantly processed and may have to be more complex than necessary.

Take, for example, a state machine with a sleeping and on-guard state. When the character wakes up, it will need to carry out a "wake-up" action, probably involving an animation and maybe some movement. Similarly, when a character decides to take a nap, it will need a "go-to-sleep" action. If we need to continually wake up or go to sleep every frame, then the state machine will actually require four states, shown in Figure 5.56.

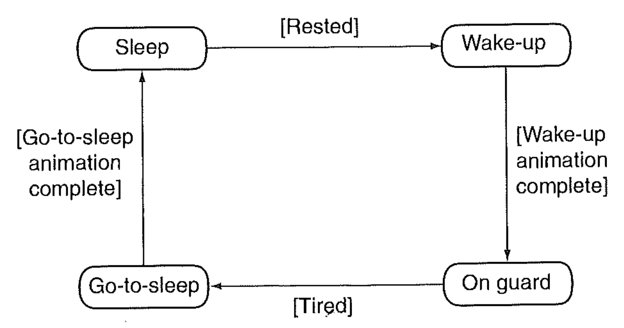
Sleep
[Rested]
Wake-up
[Go-to-sleep animation complete]
[Wake-up animation complete]
Go-to-sleep
[Tired]
On guard

Figure 5.56 State machine with transition states

This isn't a problem when we only have two states to transition between, but allowing transitions between five states would involve 40 additional transition states. It soon scales out of control.

If we can support actions with some duration, then the wake-up action will simply be requested on exiting the sleep state. The go-to-sleep action will likewise be carried out when entering the sleep state. In this case, the state machine doesn't need continual processing. After it signals that it is waking up, we can wait until the character has played its waking animation before moving onto the next thing.

Interrupting Actions

Because actions take time, they may start off being sensible things to do, but may end up being stupid long before the action is complete. If a character is sent wandering off toward a power-up, it will appear silly if it carries on toward the power-up after a squad of enemy units spring their ambush: it's time to stop going after the power-up and run away.

If a decision making system decides that an important action is needed, then it should be able to trump other actions currently being carried out. Most games allow such emergencies to even break the consistency of animation. While in normal circumstances a whole animation is played, if the character needs to do a rapid volte-face, the animation can be interrupted for another (possibly with a couple of frames of blending between them to avoid a distinct jump).

Our action manager should allow actions with higher importance to interrupt the execution of others.

Compound Actions

It is rare for a character in a game to be doing only one thing at a time. The action of a character is typically layered. They might be playing a "make-obscene-gesture" animation, while moving around the level, while pursuing an enemy, while using a health pack.

It is a common implementation strategy to split these actions up so they are generated by different decision making processes. We might use one simple decision maker to monitor health levels and schedule the use of a health pack when things look dangerous. We might use another decision maker to choose which enemy to pursue. This could then hand-off to a pathfinding routine to work out a pursuit route. In turn, this might use some other AI to work out how to follow the route and yet another piece of code to schedule the correct animations.

In this scenario, each decision making system is outputting one action of a very particular form. The action manager needs to accumulate all these actions and determine which ones can be layered.

An alternative is to have decision makers that output compound actions. In a strategy game, for example, we may need to coordinate several actions in order to be successful. The decision making system might decide to launch a small attack at a strong point in the enemy defense, while making a full-strength flanking assault. Both actions need to be carried out together. It would be difficult to coordinate separate decision making routines to get this effect.

In these cases the action returned from a decision making system will need to be made up of several atomic actions, all of which are to be executed at the same time.

This is an obvious requirement, but one close to our hearts. One AI system we know of ignored the need for compound actions until late in the development schedule. Eventually, the decision making tool (including its loading and saving formats, the connections into the rest of the game code, and the bindings to the scripting language and other tools) needed rewriting, a major headache we could have avoided.

Scripted Actions

Developers and (more commonly) games journalists occasionally talk about "scripted AI" in a way that has nothing to do with scripting languages. Scripted AI in this context usually means a set of pre-programmed actions that will always be carried out in sequence by a character. There is no decision making involved; the script is always run from the start.

For example, a scientist character may be placed in a room. When the player enters the room, the script starts running. The character rushes to a computer bank, starts the self-destruct sequence, and then runs for the door and escapes.

Scripting the behavior in this way allows developers to give the impression of better AI than would be possible if the character needed to make its own decisions. A character can be scripted to act spitefully, recklessly, or secretly, all without any AI effort.

This kind of scripted behavior is less common in current games because the player often has the potential to disrupt the action. In our example, if the player immediately runs for the door and stands there, the scientist may not be able to escape, but the script won't allow the scientist to react sensibly to the blockage. For this reason, these kinds of scripted actions are often limited to in-game cutscenes in recent games.

Scripted behavior has been used for many years in a different guise without removing the need for decision making.

Primitive actions (such as move to a point, play an animation, or shoot) can be combined into short scripts that can then be treated as a single action. A decision making system can decide to carry out a decision script that will then sequence a number of primitive actions.