8725.book Page 139 Sunday, September 23, 2007 8:57 PM

CHAPTER 7

Take Me to Your Leader

Placing a player character in a convincing world is only part of creating a game. To make a game fun, you need to present the player with a number of challenges. These may come in the form of traps and obstacles, but to really entertain your players you need to have them interact with nonplayer characters (NPCs)—characters that appear to act with a degree of intelligence or awareness in the game. The process of creating these NPCs is called artificial intelligence (AI). In this chapter, we will explore some simple techniques that you can use to give your game characters a life of their own.

Creating Artificial Intelligence for Games

You may have looked in the Pygame documentation for a `pygame.ai` module. There isn't one, because each game can have vastly different requirements when it comes to creating NPCs. The code for an ape that throws barrels at plumbers wouldn't require much work—all the ape needs to determine is whether it should throw the barrel to the left or right, something you could probably simulate in a single line of Python code! Creating a convincing enemy combatant in a futuristic first-person shooter may take a little more effort. The AI player would have to plan routes from one part of the map to another, and at the same time aim weapons and dodge enemy fire. It may also have to make decisions based on the ammo supply and armor inventory. The better it does all of this, the better AI player it will be and the greater the challenge for the player.

Although most AI in games is used to create convincing opponents to play against, it is becoming increasingly popular to use AI techniques for altogether more peaceful purposes. NPCs need not always be enemies that must be dispatched on sight; they may also be characters placed in the game world to add depth to the gameplay. Some NPCs may even be friends of the player that should be protected from harm because they actively assist in the quest. Other games, such as the phenomenally successful The Sims, don't require a player character at all, and are entirely populated with NPCs.

AI is also useful for making the game world more convincing by adding background characters that aren't directly involved in the gameplay (the game equivalent of movie extras). We can apply a few AI techniques to make birds flock together, or crowds of people flee from an out-of-control car in a racing game. It's this kind of attention to detail that truly connects a player to the game world. The trick is to convince the player that the game world would exist even if they weren't currently playing.

AI has a reputation for being difficult, which it doesn't really deserve. Much of the code you create for AI can be reused in various combinations to create a large variety of different

8725.book Page 140 Sunday, September 23, 2007 8:57 PM

types of NPCs. In fact, most games will use the same code for every character in a game and you have to tweak just a few values to modify behavior.

This chapter won't cover a great deal of the theory of artificial intelligence (which could easily consume an entire book). Rather, it will give you a number of techniques that you can apply to many situations in games.

What Is Intelligence?

Intelligence is a difficult thing to define, even for AI programmers. I'm confident that I am intelligent and self-aware, but I can only assume that others are intelligent because they are like me in many ways. Other people talk, move, check their e-mail, and take out their trash like I do—so I *assume* they are intelligent. Similarly, in a game if a character behaves in a way that an intelligent thing would, then the player will assume it is intelligent. The programmer may know that the actions of a character are simply a result of a few pages of computer code, but the player will be oblivious to that fact. As far as the player is concerned, if it walks like a zombie, moans like a zombie, and eats people like a zombie, then it's a zombie!

So intelligence in a game is an illusion (it may be in real life as well). The code to create this illusion doesn't differ a great deal from the code in the previous chapters. You will use the same basic tools of Python strings, lists, dictionaries, and so forth to build classes that are effectively the *brains* of your NPCs. In fact, Python is probably one of the best languages for writing AI because of its large range of built-in objects.

Exploring AI

Artificial intelligence isn't essential to creating an entertaining game. I used to love to play classic platform games where the hero has to leap from platform to platform and brazenly jump on the heads of monsters.

Although the monsters in these games are NPCs, their actions are a little rudimentary to be considered AI. Let's look inside the head of a typical platform game monster (Listing 7-1). This listing is *pseudocode*, which is code that's used to demonstrate a technique but that doesn't actually run.

Listing 7-1. *Pseudocode for a platform monster*

```
self.move_forward()
if self.hit_wall():
    self.change_direction()
```

The particular monster in Listing 7-1 doesn't have any awareness of its surroundings other than being able to detect if it has hit a wall, and it certainly won't react in any way to the player character that is about to land on its head. Generally speaking, a requirement for AI is that the NPC must have awareness of other entities in the games, especially the player character. Let's consider another type of game monster: a fireball-throwing imp from the underworld. The imp has a simple mission in life: to find the player and hurl a fireball in his direction. Listing 7-2 is the pseudocode for the imp's brain.

8725.book Page 141 Sunday, September 23, 2007 8:57 PM

Listing 7-2. *Pseudocode for Imp AI*

```
if self.state == "exploring":
    self.random_heading()
    if self.can_see(player):
        self.state = "seeking"

elif self.state == "seeking":
    self.head_towards("player")
    if self.in_range_of(player):
        self.fire_at(player)
    if not self.can_see(player):
        self.state = "exploring"
```

The imp can be in one of two states: *exploring* or *seeking*. The current state of the imp is stored in the value of `self.state`, and indicates which block of code currently controls the imp's actions. When the imp is exploring (i.e., `self.state == "exploring"`), it will walk aimlessly around the map by picking a random heading. But if it sees the player, it will switch to the second state of `"seeking"`. An imp that is in seeking mode will head toward the player and fire as soon as it is in range. It will keep doing this as long as the player can be seen, but if the cowardly player retreats, the imp will switch back to the exploring state.

Our imp is certainly no deep thinker, but it does have an awareness of its surroundings (i.e., where the player is) and takes actions accordingly. Even with two states, the imp will be intelligent enough to be a stock enemy in a first-person shooter. If we were to add a few more states and define the conditions to switch between them, we could create a more formidable enemy. This is a common technique in game AI and is known as a *state machine*.

■ **Note** This imp is not the smartest of underworld denizens. If the player can no longer be seen, the imp will stop seeking, even if the player has just hidden behind a tree! Fortunately we can build on the state machine to create a smarter class of imp.

Implementing State Machines

The two states for the imp's brain form a very simple state machine. A state generally defines two things:

- What the NPC is doing at that moment
- At what point it should switch to another state

The condition to get from the *exploring* state to the *seeking* state is `self.can_see(player)`—in other words, "Can I (the imp) see the player?" The opposite condition (`not self.can_see(player)`) is used to get back from *seeking* to *exploring*. Figure 7-1 is a diagram of the imp's state machine, which is effectively its brain. The arrows define the links between the states and the conditions that

must be satisfied to switch states. Links in a state machine are always one-way, but there may be another link that returns to the original state. There may also be several intermediate states before returning to the original state, depending on the complexity of the NPC's behavior.

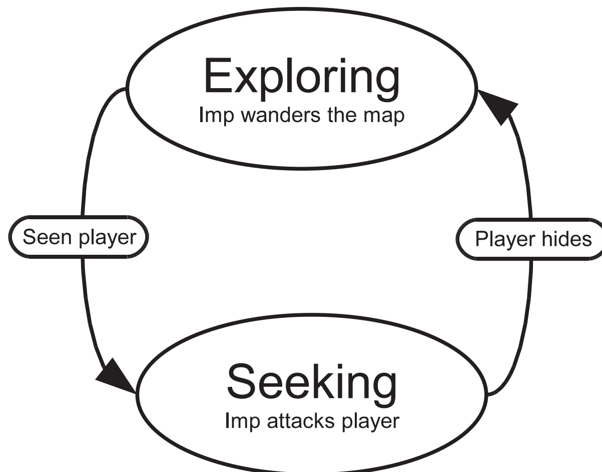


Figure 7-1. *Imp state machine*

In addition to the current behavior and conditions, states may also contain *entry actions* and *exit actions*. An entry action is something that is done prior to entering a new state, and is typically used to perform one-time actions needed by the state to run. For the `seeking` state in the imp's state machine, an entry action might calculate a heading toward the player and play a noise to indicate that it has seen the player—or anything else required to ready the imp for battle. Exit actions, the opposite of entry actions, are performed when leaving a state.

Let's create a slightly more interesting state machine so we can put this into practice. We are going to create a simulation of an ant's nest. Insects are often used when experimenting with AI because they have quite simple behaviors that are easy to model. In our simulation universe, we are going to have three *entities*: leaves, spiders, and the ants themselves. The leaves will grow in random spots on the screen and will be harvested by the ants and returned to the nest. Spiders wander over the screen, and are tolerated by the ants as long as they don't come near the nest. If a spider enters the nest, it will be chased and bitten until it either dies or manages to get far enough away.

■ **Note** Even though we are using an insect theme for this simulation, the AI code we will be writing is applicable to many scenarios. If we were to replace the ants, spiders, and leaves with giant "mech" robots, tanks, and fuel drops, then the simulation would still make sense.

Game Entities

Although we have three different types of entities, it is a good idea to come up with a *base* class for a game entity that contains common properties and actions. That way, we won't need to duplicate code for each of the entities, and we can easily add other entities without much extra work.

An entity will need to store its name (`"ant"`, `"leaf"`, or `"spider"`), as well as its current location, destination, speed, and the image used to represent it on screen. You may find it odd that the `"leaf"` entity will have a destination and speed. We aren't going to have magic walking leaves; we will simply set their speed to zero so that they don't move. That way, we can still treat leaves in the same way as the other entities. In addition to this information, we need to define a few common functions for game entities. We will need a function to render entities to the screen and another to process the entity (i.e., update its position on screen). Listing 7-3 shows the code to create a `GameEntity` class, which will be used as the base for each of the entities.

Listing 7-3. *The Base Class for a Game Entity*

```
class GameEntity(object):

    def __init__(self, world, name, image):

        self.world = world
        self.name = name
        self.image = image
        self.location = Vector2(0, 0)
        self.destination = Vector2(0, 0)
        self.speed = 0.

        self.brain = StateMachine()

        self.id = 0

    def render(self, surface):

        x, y = self.location
        w, h = self.image.get_size()
        surface.blit(self.image, (x-w/2, y-h/2))

    def process(self, time_passed):

        self.brain.think()

        if self.speed > 0 and self.location != self.destination:
```

8725.book Page 144 Sunday, September 23, 2007 8:57 PM


```
            vec_to_destination = self.destination - self.location
            distance_to_destination = vec_to_destination.get_length()
            heading = vec_to_destination.get_normalized()
            travel_distance = min(distance_to_destination, time_passed * self.speed)
            self.location += travel_distance * heading
```

The `GameEntity` class also keeps a reference to a `world`, which is an object we will use to store the positions of all the entities. This `World` object is important because it is how the entity knows about other entities in the simulation. Entities also require an ID to identify it in the world and a `StateMachine` object for its brain (which we will define later).

The `render` function for `GameEntity` simply blits the entities' image to the screen, but first adjusts the coordinates so that the current location is under the center of the image rather than the top left. We do this because the entities will be treated as circles with a point and a radius, which will simplify the math when we need to detect interactions with other entities.

The `process` function of `GameEntity` objects first calls `self.brain.think`, which will run the state machine to control the entity (typically by changing its destination). Only the ant will use a state machine in this simulation, but we *could* add AI to any entity. If we haven't built a state machine for the entity, this call will simply return without doing anything. The rest of the `process` function moves the entity toward its destination, if it is not there already.

Building Worlds

Now that we have created a `GameEntity` class, we need to create a *world* for the entities to live in. There is not much to the world for this simulation—just a nest, represented by a circle in the center of the screen, and a number of game entities of varying types. The `World` class (Listing 7-4) draws the nest and manages its entities.

Listing 7-4. *World Class*

```
class World(object):

    def __init__(self):

        self.entities = {} # Store all the entities
        self.entity_id = 0 # Last entity id assigned
        # Draw the nest (a circle) on the background
        self.background = pygame.surface.Surface(SCREEN_SIZE).convert()
        self.background.fill((255, 255, 255))
        pygame.draw.circle(self.background, (200, 255, 200), NEST_POSITION, ➥
int(NEST_SIZE))

    def add_entity(self, entity):

        # Stores the entity then advances the current id
        self.entities[self.entity_id] = entity
        entity.id = self.entity_id
        self.entity_id += 1
```

8725.book Page 145 Sunday, September 23, 2007 8:57 PM


```
    def remove_entity(self, entity):

        del self.entities[entity.id]

    def get(self, entity_id):

        # Find the entity, given its id (or None if it is not found)
        if entity_id in self.entities:
            return self.entities[entity_id]
        else:
            return None

    def process(self, time_passed):

        # Process every entity in the world
        time_passed_seconds = time_passed / 1000.0
        for entity in self.entities.itervalues():
            entity.process(time_passed_seconds)

    def render(self, surface):

        # Draw the background and all the entities
        surface.blit(self.background, (0, 0))
        for entity in self.entities.values():
            entity.render(surface)

    def get_close_entity(self, name, location, range=100.):

        # Find an entity within range of a location
        location = Vector2(*location)

        for entity in self.entities.values():
            if entity.name == name:
                distance = location.get_distance_to(entity.location)
                if distance < range:
                    return entity
        return None
```

Since we have a number of `GameEntity` objects, it would be perfectly natural to use a Python list object to store them. Although this could work, we would run into problems; when an entity needs to be removed from the world (i.e., it died), we would have to search through the list to find its index, and then call `del` to delete it. Searching through lists can be slow, and would only get slower as the list grows. A better way to store entities is with a Python dictionary, which can efficiently find an entity even if there are many of them.

To store entities in a dictionary, we need a value to use as a key, which could be a string, a number, or another value. Thinking of a name for each ant would be difficult, so we will simply number the ants sequentially: the first ant is #0, the second is #1, and so on. This number is

8725.book Page 146 Sunday, September 23, 2007 8:57 PM

the entity's id, and is stored in every GameEntity object so that we can always locate the object in the dictionary (see Figure 7-2).

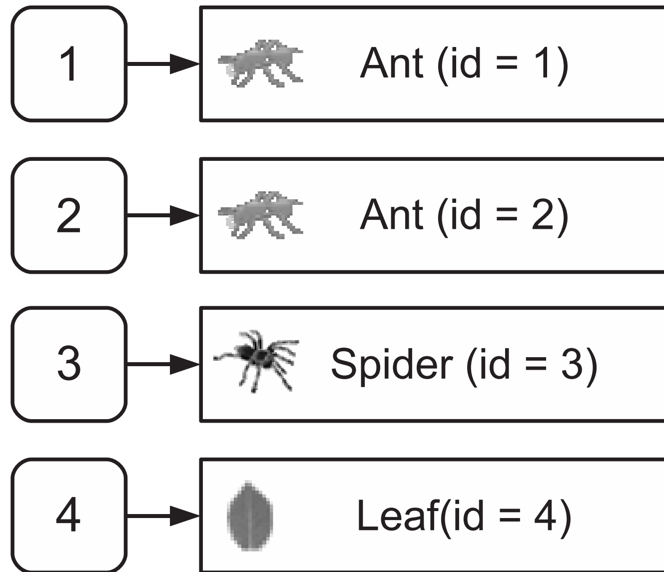


Figure 7-2. *The entities dictionary*

■ **Note** A dictionary that maps incremental numbers to its values is similar to a list, but the keys won't shuffle down if a value is deleted. So Ant #5 will still be Ant #5, even if Ant #4 is removed.

Most of the functions in the World class are responsible for managing the entities in some way. There is an add_entity function to add an entity to the world, a remove_entity function to remove it from the world, and a get function that looks up the entity given its id. If get can't find the id in the entities dictionary, it will return None. This is useful because it will tell us that an entity has been removed (id values are never reused). Consider the situation where a group of ants are in hot pursuit of a spider that has invaded the nest. Each ant object stores the id of the spider it is chasing and will look it up (with get) to retrieve the spider's location. At some point, though, the unfortunate spider will be dispatched and removed from the world. When this happens, any call to the get function with the spider's id will return None, so the ants will know they can stop chasing and return to other duties.

Also in the World class we have a process and a render function. The process function of the World object calls the process function of each entity to give it a chance to update its position. The render function is similar; in addition to drawing the background, it calls the corresponding render function of each entity to draw the appropriate graphic at its location.

8725.book Page 147 Sunday, September 23, 2007 8:57 PM

Finally in the World class there is a function called get_close_entity, which finds an entity that is within a certain distance of a location in the world. This will be used in several places in the simulation.

■ **Note** When implementing an NPC, you should generally limit the information available to it, because like real people NPCs may not necessarily be aware of everything that is going on in the world. We simulate this with the ants, by only letting them *see* objects within a limited distance.

Ant Entity Class

Before we model the brain for the ants, let's look at the Ant class (Listing 7-5). It derives from GameEntity, so that it will have all the capabilities of a GameEntity, together with any additional functions we add to it.

Listing 7-5. *The Ant Entity Class*

```
class Ant(GameEntity):

    def __init__(self, world, image):

        # Call the base class constructor
        GameEntity.__init__(self, world, "ant", image)

        # Create instances of each of the states
        exploring_state = AntStateExploring(self)
        seeking_state = AntStateSeeking(self)
        delivering_state = AntStateDelivering(self)
        hunting_state = AntStateHunting(self)

        # Add the states to the state machine (self.brain)
        self.brain.add_state(exploring_state)
        self.brain.add_state(seeking_state)
        self.brain.add_state(delivering_state)
        self.brain.add_state(hunting_state)

        self.carry_image = None

    def carry(self, image):

        self.carry_image = image

    def drop(self, surface):
```

8725.book Page 148 Sunday, September 23, 2007 8:57 PM


```
        # Blit the 'carry' image to the background and reset it
        if self.carry_image:
            x, y = self.location
            w, h = self.carry_image.get_size()
            surface.blit(self.carry_image, (x-w, y-h/2))
            self.carry_image = None

    def render(self, surface):

        # Call the render function of the base class
        GameEntity.render(self, surface)

        # Extra code to render the 'carry' image
        if self.carry_image:
            x, y = self.location
            w, h = self.carry_image.get_size()
            surface.blit(self.carry_image, (x-w, y-h/2))
```

The constructor for our `Ant` class (`__init__`) first calls the constructor for the base class with the line `GameEntity.__init__(self, world, "ant", image)`. We have to call it this way because if we were to call `self.__init__` Python would call the constructor in `Ant`—and end up in an infinite loop! The remaining code in the ant's constructor creates the state machine (covered in the next section) and also sets a member variable called `carry_image` to `None`. This variable is set by the `carry` function and is used to store the image of an object that the ant is carrying; it could be a leaf or a dead spider. If the `drop` function is called, it will set `carry_image` back to `None`, and it will no longer be drawn.

Because of the ability to *carry* other images, ants have an extra requirement when it comes to rendering the sprite. We want to draw the image the ant is carrying in addition to its own image, so ants have a *specialized* version of `render`, which calls the `render` function in the base class and then renders `carry_image`, if it is not set to `None`.

Building the Brains

Each ant is going to have four states in its state machine, which should be enough to simulate ant-like behavior. The first step in defining the state machine is to work out what each state should do, which are the *actions* for the state (see Table 7-1).

Table 7-1. *Actions for the Ant States*

State	Actions
Exploring	Walk toward a random point in the world.
Seeking	Head toward a leaf.
Delivering	Deliver something to the nest.
Hunting	Chase a spider.

8725.book Page 149 Sunday, September 23, 2007 8:57 PM

We also need to define the links that connect states together. These take the form of a condition and the name of the state to switch to if the condition is met. The exploring state, for example, has two such links (see Table 7-2).

Table 7-2. *Links from Exploring State*

Condition	Destination State
Seen a leaf?	Seeking
Spider attacking base?	Hunting

Once we have defined the links between the states, we have a state machine that can be used as the brain for an entity. Figure 7-3 shows the complete state machine that we will be building for the ant. Drawing a state machine out on paper like this is a great way of visualizing how it all fits together, and will help you when you need to turn it into code.

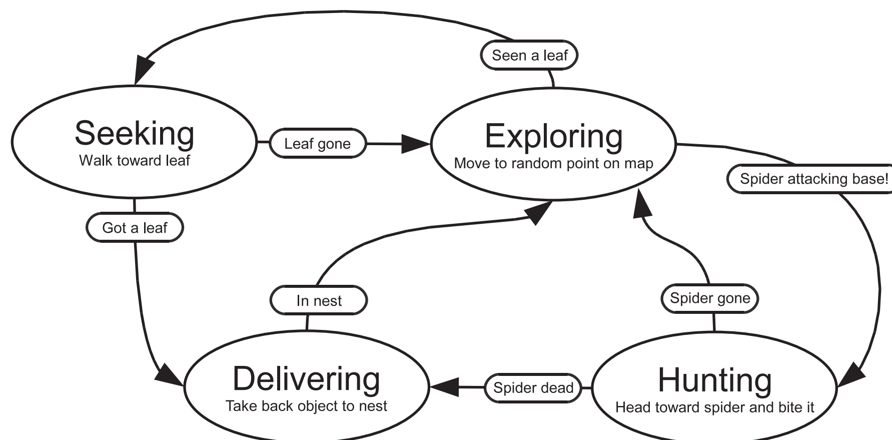


Figure 7-3. *Ant state machine*

Let's put this into practice and create the code for the state machine. We will begin by defining a base class for an individual state (Listing 7-6). Later we will create another class for the state machine as a whole that will manage the states it contains.

The base `State` class doesn't actually do anything other than store the name of the state in the constructor. The remaining functions in `State` do nothing—the `pass` keyword simply tells Python that you intentionally left the function blank. We need these empty functions because not all of the states we will be building will implement all of the functions in the base class. The exploring state, for example, has no exit actions. When we come to implement the `AntStateExploring` class, we can omit the `exit_actions` function because it will safely fall back to the do-nothing version of the function in the base class (`State`).

Listing 7-6. *Base Class for a State*

```
class State(object):

    def __init__(self, name):
        self.name = name

    def do_actions(self):
        pass

    def check_conditions(self):
        pass

    def entry_actions(self):
        pass

    def exit_actions(self):
        pass
```

Before we build the states, we need to build a class that will manage them. The `StateMachine` class (Listing 7-7) stores an instance of each of the states in a dictionary and manages the currently active state. The `think` function runs once per frame, and calls the `do_actions` on the active state—to do whatever the state was designed to do; the exploring state will select random places to walk to, the *seeking* state will move toward the leaf, and so forth. The `think` function also calls the state's `check_conditions` function to check all of the link conditions. If `check_conditions` returns a string, a new active state will be selected and any exit and entry actions will run.

Listing 7-7. *The State Machine Class*

```
class StateMachine(object):

    def __init__(self):

        self.states = {}    # Stores the states
        self.active_state = None    # The currently active state

    def add_state(self, state):

        # Add a state to the internal dictionary
        self.states[state.name] = state

    def think(self):
```

8725.book Page 151 Sunday, September 23, 2007 8:57 PM


```
    # Only continue if there is an active state
        if self.active_state is None:
            return

        # Perform the actions of the active state, and check conditions
        self.active_state.do_actions()

        new_state_name = self.active_state.check_conditions()
        if new_state_name is not None:
            self.set_state(new_state_name)

    def set_state(self, new_state_name):

        # Change states and perform any exit / entry actions
        if self.active_state is not None:
            self.active_state.exit_actions()

        self.active_state = self.states[new_state_name]
        self.active_state.entry_actions()
```

Now that we have a functioning state machine class, we can start implementing each of the individual states by deriving from the `State` class and implementing some of its functions. The first state we will implement is the exploring state, which we will call `AntStateExploring` (see Listing 7-8). The entry actions for this state give the ant a random speed and set its destination to a random point on the screen. The main actions, in the `do_actions` function, select another random destination if the expression `randint(1, 20) == 1` is true, which will happen in about 1 in every 20 calls, since `randint` (in the `random` module) selects a random number that is greater than or equal to the first parameter, and less than or equal to the second. This gives us the antlike random searching behavior we are looking for.

The two outgoing links for the exploring state are implemented in the `check_conditions` function. The first condition looks for a leaf entity that is within 100 pixels from an ant's location (because that's how far our ants can see). If there is a nearby leaf, then `check_conditions` records its `id` and returns the string `seeking`, which will instruct the state machine to switch to the seeking state. The remaining condition will switch to `hunting` if there are any spiders inside the nest *and* within 100 pixels of the ant's location.

■ **Caution** Random numbers are a good way to make your game more fun, because predictable games can get dull after a while. But be careful with random numbers—if something goes wrong, it may be difficult to reproduce the problem!

8725.book Page 152 Sunday, September 23, 2007 8:57 PM

Listing 7-8. *The Exploring State for Ants (AntStateExploring)*

```
class AntStateExploring(State):

    def __init__(self, ant):

        # Call the base class constructor to initialize the State
        State.__init__(self, "exploring")
        # Set the ant that this State will manipulate
        self.ant = ant

    def random_destination(self):

        # Select a point in the screen
        w, h = SCREEN_SIZE
        self.ant.destination = Vector2(randint(0, w), randint(0, h))

    def do_actions(self):

        # Change direction, 1 in 20 calls
        if randint(1, 20) == 1:
            self.random_destination()

    def check_conditions(self):

        # If there is a nearby leaf, switch to seeking state
        leaf = self.ant.world.get_close_entity("leaf", self.ant.location)
        if leaf is not None:
            self.ant.leaf_id = leaf.id
            return "seeking"
        # If there is a nearby spider, switch to hunting state
        spider = self.ant.world.get_close_entity("spider", NEST_POSITION, NEST_SIZE)
        if spider is not None:
            if self.ant.location.get_distance_to(spider.location) < 100.:
                self.ant.spider_id = spider.id
                return "hunting"

        return None

    def entry_actions(self):

        # Start with random speed and heading
        self.ant.speed = 120. + randint(-30, 30)
        self.random_destination()
```

8725.book Page 153 Sunday, September 23, 2007 8:57 PM

As you can see from Listing 7-8, the code for an individual state need not be very complex because the states work together to produce something that is more than the sum of its parts. The other states are similar to `AntStateExploring` in that they pick a destination based on the goal of that state and switch to another state if they have accomplished that goal, or it no longer becomes relevant.

There is not a great deal left to do in the main loop of the game. Once the `World` object has been created, we simply call `process` and `render` once per frame to update and draw everything in the simulation. Also in the main loop are a few lines of code to create leaf entities at random positions in the world and occasionally create spider entities that wander in from the left side of the screen.

Listing 7-9 shows the entire simulation. When you run it, you will see something like Figure 7-4; the ants roam around the screen collecting leaves and killing spiders, which they will pile up in the nest. You can see that the ants satisfy the criteria of being AIs because they are aware of their environment—in a limited sense—and take actions accordingly.

Although there is no *player* character in this simulation, this is the closest we have come to a true game. We have a world, an entity framework, and artificial intelligence. It could be turned into a game with the addition of a player character. You could define a completely new entity for the player, perhaps a praying mantis that has to eat the ants, or add keyboard control to the spider entity and have it collect eggs from the nest. Alternatively, the simulation is a great starting point for a strategy game where groups of ants can be sent to collect leaves or raid neighboring nests. Game developers should be as imaginative as possible!

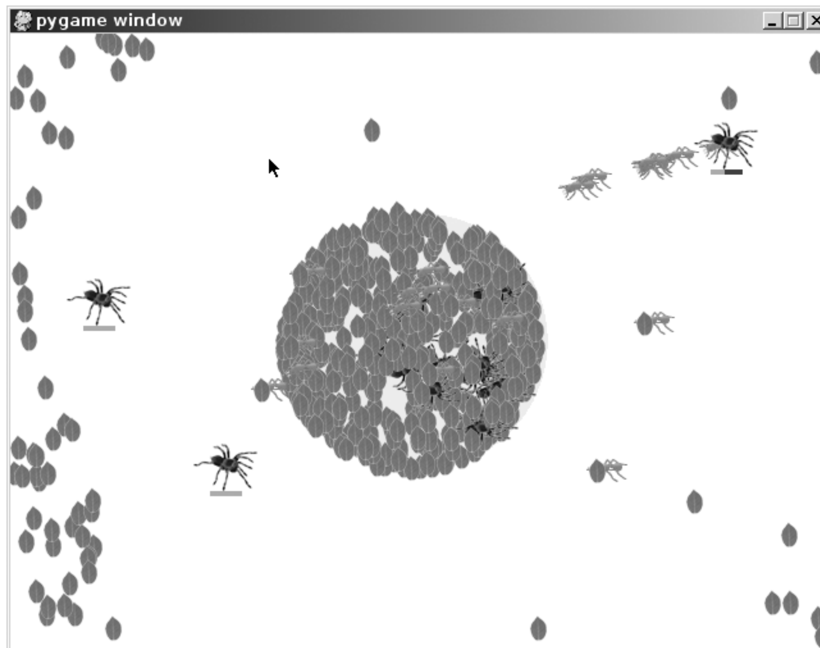


Figure 7-4. *The ant simulation*

Listing 7-9. *The Complete AI Simulation (antstatemachine.py)*

```
# Some constants you can modify
SCREEN_SIZE = (640, 480)
NEST_POSITION = (320, 240)
ANT_COUNT = 20
NEST_SIZE = 100.

import pygame
from pygame.locals import *

from random import randint, choice
from gameobjects.vector2 import Vector2

class State(object):

    def __init__(self, name):
        self.name = name

    def do_actions(self):
        pass

    def check_conditions(self):
        pass

    def entry_actions(self):
        pass

    def exit_actions(self):
        pass


class StateMachine(object):

    def __init__(self):

        self.states = {}
        self.active_state = None

    def add_state(self, state):

        self.states[state.name] = state

    def think(self):

        if self.active_state is None:
            return
```

8725.book Page 155 Sunday, September 23, 2007 8:57 PM


```
        self.active_state.do_actions()

        new_state_name = self.active_state.check_conditions()
        if new_state_name is not None:
            self.set_state(new_state_name)

    def set_state(self, new_state_name):

        if self.active_state is not None:
            self.active_state.exit_actions()

        self.active_state = self.states[new_state_name]
        self.active_state.entry_actions()


class World(object):

    def __init__(self):

        self.entities = {}
        self.entity_id = 0
        self.background = pygame.surface.Surface(SCREEN_SIZE).convert()
        self.background.fill((255, 255, 255))
        pygame.draw.circle(self.background, (200, 255, 200), NEST_POSITION,➥
int(NEST_SIZE))

    def add_entity(self, entity):

        self.entities[self.entity_id] = entity
        entity.id = self.entity_id
        self.entity_id += 1

    def remove_entity(self, entity):

        del self.entities[entity.id]

    def get(self, entity_id):

        if entity_id in self.entities:
            return self.entities[entity_id]
        else:
            return None

    def process(self, time_passed):
```

8725.book Page 156 Sunday, September 23, 2007 8:57 PM


```
        time_passed_seconds = time_passed / 1000.0
        for entity in self.entities.values():
            entity.process(time_passed_seconds)

    def render(self, surface):

        surface.blit(self.background, (0, 0))
        for entity in self.entities.itervalues():
            entity.render(surface)

    def get_close_entity(self, name, location, range=100.):

        location = Vector2(*location)

        for entity in self.entities.itervalues():
            if entity.name == name:
                distance = location.get_distance_to(entity.location)
                if distance < range:
                    return entity
        return None


class GameEntity(object):

    def __init__(self, world, name, image):

        self.world = world
        self.name = name
        self.image = image
        self.location = Vector2(0, 0)
        self.destination = Vector2(0, 0)
        self.speed = 0.

        self.brain = StateMachine()

        self.id = 0

    def render(self, surface):

        x, y = self.location
        w, h = self.image.get_size()
        surface.blit(self.image, (x-w/2, y-h/2))

    def process(self, time_passed):

        self.brain.think()
```

8725.book Page 157 Sunday, September 23, 2007 8:57 PM


```
        if self.speed > 0. and self.location != self.destination:

            vec_to_destination = self.destination - self.location
            distance_to_destination = vec_to_destination.get_length()
            heading = vec_to_destination.get_normalized()
            travel_distance = min(distance_to_destination, time_passed * self.speed)
            self.location += travel_distance * heading


class Leaf(GameEntity):

    def __init__(self, world, image):
        GameEntity.__init__(self, world, "leaf", image)


class Spider(GameEntity):

    def __init__(self, world, image):
        GameEntity.__init__(self, world, "spider", image)

        # Make a 'dead' spider image by turning it upside down
        self.dead_image = pygame.transform.flip(image, 0, 1)

        self.health = 25
        self.speed = 50. + randint(-20, 20)

    def bitten(self):

        # Spider as been bitten
        self.health -= 1
        if self.health <= 0:
            self.speed = 0.
            self.image = self.dead_image
        self.speed = 140.

    def render(self, surface):

        GameEntity.render(self, surface)

        # Draw a health bar
        x, y = self.location
        w, h = self.image.get_size()
        bar_x = x - 12
        bar_y = y + h/2
        surface.fill( (255, 0, 0), (bar_x, bar_y, 25, 4))
        surface.fill( (0, 255, 0), (bar_x, bar_y, self.health, 4))
```

8725.book Page 158 Sunday, September 23, 2007 8:57 PM


```
    def process(self, time_passed):

        x, y = self.location
        if x > SCREEN_SIZE[0] + 2:
            self.world.remove_entity(self)
            return

        GameEntity.process(self, time_passed)


class Ant(GameEntity):

    def __init__(self, world, image):

        GameEntity.__init__(self, world, "ant", image)

        # State classes are defined below
        exploring_state = AntStateExploring(self)
        seeking_state = AntStateSeeking(self)
        delivering_state = AntStateDelivering(self)
        hunting_state = AntStateHunting(self)

        self.brain.add_state(exploring_state)
        self.brain.add_state(seeking_state)
        self.brain.add_state(delivering_state)
        self.brain.add_state(hunting_state)

        self.carry_image = None

    def carry(self, image):

        self.carry_image = image

    def drop(self, surface):

        if self.carry_image:
            x, y = self.location
            w, h = self.carry_image.get_size()
            surface.blit(self.carry_image, (x-w, y-h/2))
            self.carry_image = None

    def render(self, surface):

        GameEntity.render(self, surface)
```

```
        if self.carry_image:
            x, y = self.location
            w, h = self.carry_image.get_size()
            surface.blit(self.carry_image, (x-w, y-h/2))


class AntStateExploring(State):

    def __init__(self, ant):

        State.__init__(self, "exploring")
        self.ant = ant

    def random_destination(self):

        w, h = SCREEN_SIZE
        self.ant.destination = Vector2(randint(0, w), randint(0, h))

    def do_actions(self):

        if randint(1, 20) == 1:
            self.random_destination()

    def check_conditions(self):

        # If ant sees a leaf, go to the seeking state
        leaf = self.ant.world.get_close_entity("leaf", self.ant.location)
        if leaf is not None:
            self.ant.leaf_id = leaf.id
            return "seeking"

        # If the ant sees a spider attacking the base, go to hunting state
        spider = self.ant.world.get_close_entity("spider", NEST_POSITION, NEST_SIZE)
        if spider is not None:
            if self.ant.location.get_distance_to(spider.location) < 100.:
                self.ant.spider_id = spider.id
                return "hunting"

        return None

    def entry_actions(self):

        self.ant.speed = 120. + randint(-30, 30)
        self.random_destination()
```

8725.book Page 160 Sunday, September 23, 2007 8:57 PM


```
class AntStateSeeking(State):

    def __init__(self, ant):

        State.__init__(self, "seeking")
        self.ant = ant
        self.leaf_id = None

    def check_conditions(self):

        # If the leaf is gone, then go back to exploring
        leaf = self.ant.world.get(self.ant.leaf_id)
        if leaf is None:
            return "exploring"

        # If we are next to the leaf, pick it up and deliver it
        if self.ant.location.get_distance_to(leaf.location) < 5.0:

            self.ant.carry(leaf.image)
            self.ant.world.remove_entity(leaf)
            return "delivering"

        return None

    def entry_actions(self):

        # Set the destination to the location of the leaf
        leaf = self.ant.world.get(self.ant.leaf_id)
        if leaf is not None:
            self.ant.destination = leaf.location
            self.ant.speed = 160. + randint(-20, 20)


class AntStateDelivering(State):

    def __init__(self, ant):

        State.__init__(self, "delivering")
        self.ant = ant


    def check_conditions(self):
```

8725.book Page 161 Sunday, September 23, 2007 8:57 PM


```
        # If inside the nest, randomly drop the object
        if Vector2(*NEST_POSITION).get_distance_to(self.ant.location) < NEST_SIZE:
            if (randint(1, 10) == 1):
                self.ant.drop(self.ant.world.background)
                return "exploring"

        return None


    def entry_actions(self):

        # Move to a random point in the nest
        self.ant.speed = 60.
        random_offset = Vector2(randint(-20, 20), randint(-20, 20))
        self.ant.destination = Vector2(*NEST_POSITION) + random_offset


class AntStateHunting(State):

    def __init__(self, ant):

        State.__init__(self, "hunting")
        self.ant = ant
        self.got_kill = False

    def do_actions(self):

        spider = self.ant.world.get(self.ant.spider_id)

        if spider is None:
            return

        self.ant.destination = spider.location

        if self.ant.location.get_distance_to(spider.location) < 15.:

            # Give the spider a fighting chance to avoid being killed!
            if randint(1, 5) == 1:
                spider.bitten()

                # If the spider is dead, move it back to the nest
                if spider.health <= 0:
                    self.ant.carry(spider.image)
                    self.ant.world.remove_entity(spider)
                    self.got_kill = True
```

8725.book Page 162 Sunday, September 23, 2007 8:57 PM

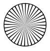

```
    def check_conditions(self):

        if self.got_kill:
            return "delivering"

        spider = self.ant.world.get(self.ant.spider_id)

        # If the spider has been killed then return to exploring state
        if spider is None:
            return "exploring"

        # If the spider gets far enough away, return to exploring state
        if spider.location.get_distance_to(NEST_POSITION) > NEST_SIZE * 3:
            return "exploring"

        return None

    def entry_actions(self):

        self.speed = 160. + randint(0, 50)

    def exit_actions(self):

        self.got_kill = False


def run():

    pygame.init()
    screen = pygame.display.set_mode(SCREEN_SIZE, 0, 32)

    world = World()

    w, h = SCREEN_SIZE

    clock = pygame.time.Clock()

    ant_image = pygame.image.load("ant.png").convert_alpha()
    leaf_image = pygame.image.load("leaf.png").convert_alpha()
    spider_image = pygame.image.load("spider.png").convert_alpha()

    # Add all our ant entities
    for ant_no in xrange(ANT_COUNT):
```

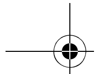
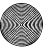
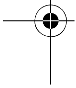
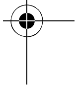
```
        ant = Ant(world, ant_image)
        ant.location = Vector2(randint(0, w), randint(0, h))
        ant.brain.set_state("exploring")
        world.add_entity(ant)


    while True:

        for event in pygame.event.get():
            if event.type == QUIT:
                return

        time_passed = clock.tick(30)

        # Add a leaf entity 1 in 20 frames
        if randint(1, 10) == 1:
            leaf = Leaf(world, leaf_image)
            leaf.location = Vector2(randint(0, w), randint(0, h))
            world.add_entity(leaf)

        # Add a spider entity 1 in 100 frames
        if randint(1, 100) == 1:
            spider = Spider(world, spider_image)
            spider.location = Vector2(-50, randint(0, h))
            spider.destination = Vector2(w+50, randint(0, h))
            world.add_entity(spider)

        world.process(time_passed)
        world.render(screen)

        pygame.display.update()

if __name__ == "__main__":
    run()
```

Summary

Making a nonplayer character behave in a realistic fashion is the goal of artificial intelligence in games. Good AI adds an extra dimension to the game because players will feel that they are in a real world rather than a computer program. Poor AI can destroy the illusion of realism as easily as glitches in the graphics or unrealistic sounds—possibly even more so. A player might be able to believe that a crudely drawn stick figure is a real person, but only as long as it doesn't bump into walls!

The apparent intelligence of an NPC is not always related to the amount of code used to simulate it. Players will tend to attribute intelligence to NPCs that is not really there. In the ant simulation that we created for this chapter, the ants will form an orderly queue when chasing



8725.book Page 164 Sunday, September 23, 2007 8:57 PM





the spider. A friend of mine saw this and remarked that they were cooperating in the hunt—but of course the ants were acting completely independently. Sometimes it can take surprisingly little work to convince the player that something is smart.

State machines are a practical and easy way of implementing game AI because they break down a complex system (i.e., a brain) into smaller chunks that are easy to implement. They aren't difficult to design because we are accustomed to imagining what other people or animals are thinking when they do things. It may not be practical to turn every thought into computer code, but you only need to approximate behavior to simulate it in a game.

The simple state machine framework we created in this chapter can be used in your own games to build convincing AI. As with the ant simulation, start out by defining what the actions for your NPC are and then figure out what would make it switch between these actions. Once you have this laid out on paper (as in Figure 7-3), you can start building the individual states in code.

The next chapter is a gentle introduction to rendering three-dimensional graphics with Pygame.