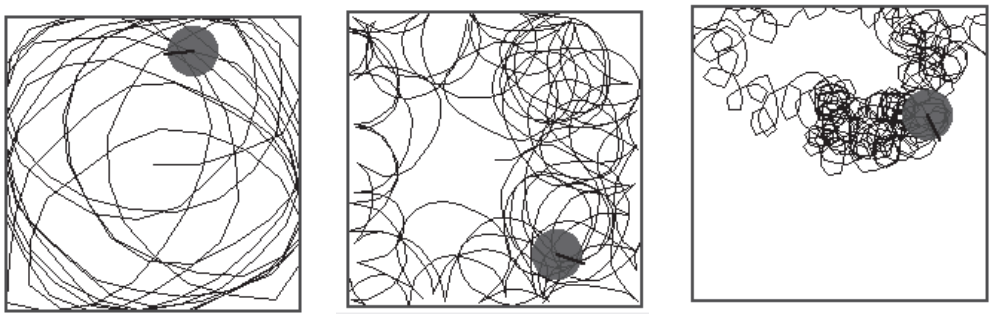
CHAPTER 23

Simulating Animal Behavior



Computers are good for modeling the world in which we live, from plant growth to economic models to the behavior of markets. In this chapter I will show you how to model certain animal behaviors and use simulation to understand the factors that influence them. Together we will model strategies that animals develop to walk, escape, find food, and remain in a friendly environment.

Wandering

Let's start by modeling how an animal might wander. The basic approach to simulating an animal's wandering behavior is to write a loop in which the animal walks and turns somewhat at random. We will use a random number in defining a method `wandering:` that makes the receiver wander by walking a random number of steps, turning through a random angle, and repeating these two moves `aNumber` times. (To obtain a random integer between 1 and 30, send the message `atRandom` to the number `30`.) A possible result of executing the method is illustrated in Figure 23-1.

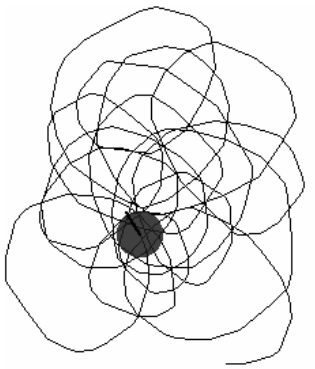


Figure 23-1. *An animal wanders by walking a random number of steps and then turning randomly.*

Method 23-1 presents one way to define the simple behavior described above and illustrated in the figure. Here we simply tell a robot to move randomly a distance between 1 and 30 pixels and turn left through an angle between 1 and 30 degrees. Script 23-1 shows how to invoke this method.

Script 23-1

```
Bot new wandering: 500
```

Method 23-1

```
wandering: n
  "Make the robot walk a random distance and turn through a random angle n times"
  n timesRepeat:
      [ self go: 30 atRandom.
      self turnLeft: 30 atRandom ]
```

Of course, animals do not wander at random. Eons of evolutionary development have created animals that move and turn in response to stimuli in their environment. Perhaps an animal turns and wanders until a certain event occurs. To begin to model such behavior in which the animal loops until some event happens, which we will model by having the user press a mouse button, you could use a conditional loop. The method `wanderingUntilButtonPressed` (Method 23-2) illustrates this point. It allows an animal to wander until the user presses one of the mouse buttons. Since the loop is executed extremely rapidly, it may happen that your computer seems blocked, so just keep holding down the button.

Method 23-2. *An animal wanders randomly until an event occurs.*

```
wanderingUntilButtonPressed
  "Make the robot walk a random distance and turn through a random angle
   until a mouse button is pressed"

  [ self anyButtonPressed ] whileFalse:
      [ self go: 30 atRandom.
      self turnLeft: 30 atRandom ]
```

Separating Influences

Method 23-1 is interesting, but it mixes two aspects of the animal's wandering: the random walking and the random changes of direction. Moreover, every time you want to try a new value of the angle or the number of steps, you have to recompile the method `wandering: n`. Therefore, define a method `wandering: n maxAngle: anAngle` that takes as its second argument the maximum value of the random angle through which the receiver should turn. In this method, let the robot always move forward a fixed distance. Such a method may be invoked as in Script 23-2.

Script 23-2

```
Bot new wandering: 500 maxAngle: 60
```

Try to guess what the animal trace will look like before executing your scripts. Experiment with different angle values. Figures 23-2 and 23-3 show different results with 15, 60, 90, 180, and 360 degrees.

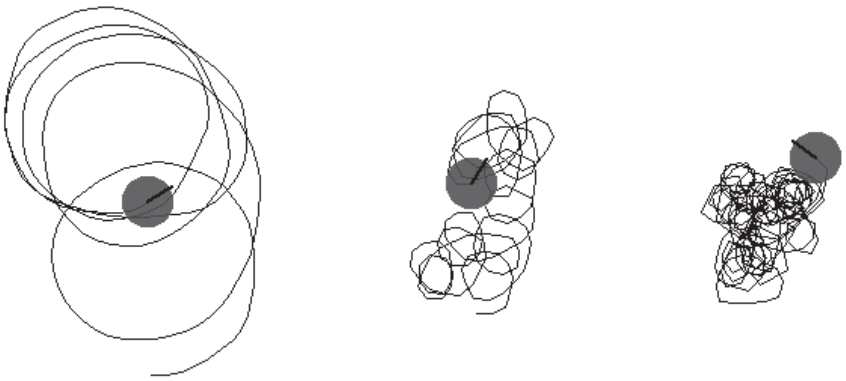


Figure 23-2. *Walking with random angles of maximum 15 (left), 60 (center), and 90 (right) degrees*

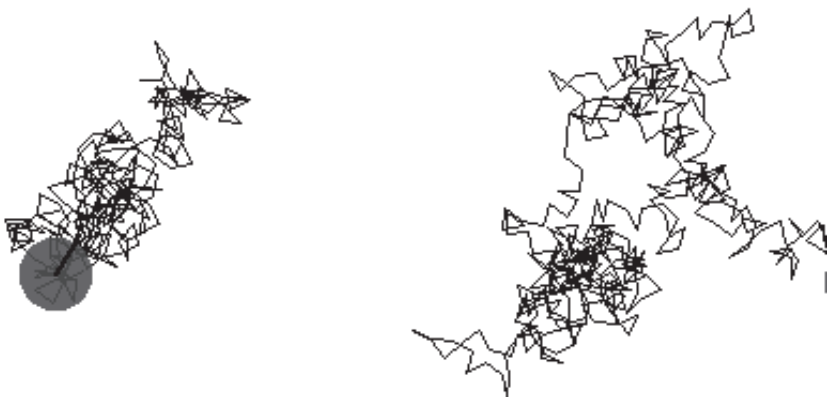


Figure 23-3. *Walking with random angles of maximum 180 (left) and 360 (right) degrees*

Studying the Influence of the Length

We have just been examining the influence of the angle on the shape of the walk. What are your hypotheses about the influence of the length? What should happen if an animal walks a random distance and then turns through a constant angle?

Studying the Influence of the Side to Which the Animal Turns

Up to now, we have always had our animal turn to the left. We can study the influence of the ability to turn to only one side or to two sides. Try to think of a solution to this programming problem before reading my solution. Hint: Note that `atRandom` returns a number between the receiver and 1. Therefore `2 atRandom` returns either 1 or 2.

One possible way to generate a random choice is by introducing a random number (1 or 2) to represent the side chosen, as sketched in Script 23-3. Another idea is to generate a random number twice the maximum angle desired and subtract that angle from the random number. More precisely, to obtain a number between `-45` and `45`, you can generate a random number between 1 and 91 and subtract 46 from that number, as shown in Script 23-4. Figure 23-4 shows what can happen when these strategies are used, and you can see that the path looks more like that of a real animal, a snail, say, than the previous ones.

Figure 23-4. *An animal wanders by walking and then turning randomly, where turning to the right and to the left are equally likely.*

Script 23-3. *A random number (1 or 2) determines whether the animal turns left or right.*

```
...
   left := 2 atRandom.
   left = 1
        ifTrue: [ self turnLeft: ...]
        ifFalse: [ self turnRight: ...]
...
```

Script 23-4. *The maximum turning angle is subtracted from a random number to yield an angle that can be positive or negative.*

```
...
   rdAngle := ((1 + (angle * 2)) atRandom) - (1 + angle).
   self turn: rdAngle.
...
```

Trapped in a Box

Now I would like to constrain the wandering of the animal so that it remains inside a box. This will allow us to study different strategies that bugs seem to follow when they find themselves in such an uncomfortable situation. Doing this is easy. Before telling the animal to move through a certain distance, you have to check whether the location where it would end up is contained in the box. Such a constrained move has already been presented in chapter 22, with Method 22-1, but I will repeat the code in Method 23-3.

Method 23-3

```
go: aDistance ifStayInBox: aRectangle
  "Move the receiver forward only if it stays within aRectangle"
  (aRectangle containsPoint: (self positionIfGo: aDistance))
       ifTrue: [ self go: aDistance ]
```

To create a box, I can create a rectangle, as shown in Script 23-5. This script creates a box 200 pixels on a side around the current position of the animal.

Script 23-5. *Create a rectangle centered on an animal.*

```
| pica rectangle |
pica := Bot new.
rectangle := Rectangle center: pica center extent: 200@200.
```

To improve the visual effect of the simulation and if you want to see the box on the screen, you have to create a "rectangle morph," that is, a graphical object whose shape is a rectangle, as described in Script 23-6. This script first creates a rectangle, then creates a rectangle morph whose bounds are those of the rectangle, with a transparent interior and blue border. Finally, the rectangle morph is displayed by invoking the method `openInWorld`.

Script 23-6. *Visualizing a rectangle centered on an animal*

```
| pica rectangle rm |
pica := Bot new.
rectangle := Rectangle center: pica center extent: 200@200.
rm:= RectangleMorph new.
rm bounds: rectangle.
rm color: Color transparent.
rm setBorderWidth: 2 borderColor: Color blue.
rm openInWorld.
```

Now I will define the method `box: aRectangle` that draws a box representing the rectangle, as you will certainly get considerable use out of it.

Method 23-4. *Draw a visible box on the screen.*

```
box: aRectangle
  "Draw a morph to represent the rectangle"
  | rm |
  rm := RectangleMorph new.
  rm bounds: aRectangle.
  rm color: Color transparent.
  rm setBorderWidth: 2 borderColor: Color blue.
  rm openInWorld
```

Now combine all the pieces to create an animal inside a box and test the method `go: anInteger ifStayInBox: aRectangle`. The next natural question is this: how can we more accurately model the behavior of the trapped animal? Imagine different alternatives. You should experiment with a number of variations.

Following Borders

One approach is to make the animal turn a little bit when it can't move and try again. This is what Method 23-5 specifies. When the robot can move, it just wanders, but if its motion would cause it to bump into a box border, it turns through an angle. This behavior is contained in a loop that makes the robot turn and try again to walk. If the turn is not big enough, it just continues to turn until it can move again.

Scripts 23-6 and 23-7 together produce results similar to those shown in Figure 23-5. Script 23-6 displays a rectangle morph to materialize the box, and Script 23-7 controls the animal's motion.

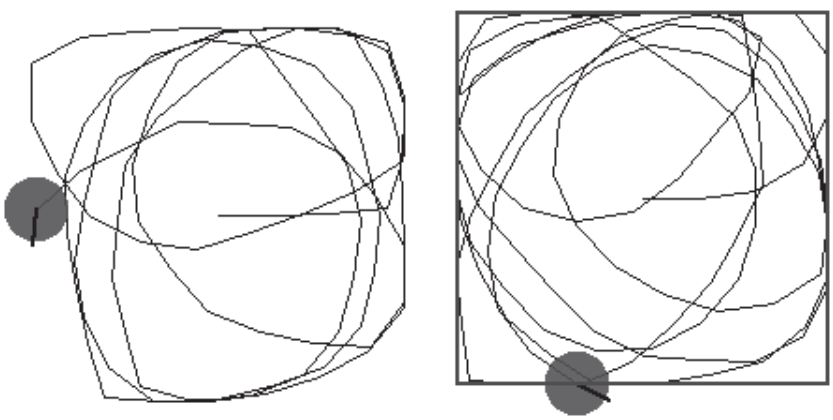


Figure 23-5. *An animal trapped in a box turns until it can move without bumping into the border.*

Script 23-7. *An animal trapped in a box tries to move.*

```
| t rec |
t := Bot new.
rec := (Rectangle center: t center extent: 200@200).
t follow: 500 borderOfBox: rec
```

Method 23-5. *The robot turns if it can't move and tries again.*

```
follow: n borderOfBox: aRectangle
  self box: aRectangle.
  n timesRepeat:
     [ (aRectangle containsPoint: (self positionIfGo: 30))
        ifTrue: [ self go: 30.
                self turnLeft: 30 atRandom ]
        ifFalse: [ self turnLeft: 1 ] ]
```

Flying to the Opposite Border

Another strategy to simulate is that of an insect that tries to fly to the opposite border. I will let you code this behavior, a possible trace of which is shown in the left pane of Figure 23-6.

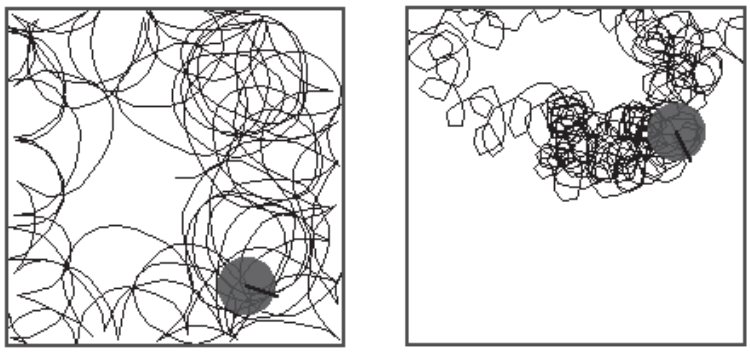


Figure 23-6. *Left: The insect moves in the opposite direction when confronted by a wall. Right: The insect chooses a random direction when confronted by a wall. This picture was obtained by setting the maximum length of the random walk to 10 pixels and turning a maximum of 90 degrees. To escape, it will make a random turn of a maximum of 360 degrees.*

Random Direction

Another choice is for the animal to change its direction at random. I will let you define this behavior, for which a possible trace is shown in the right pane of Figure 23-6.

Introducing an Exit in the Box

Now you can add another rectangle to represent an exit in the box as illustrated in Script 23-8 and displayed in Figure 23-7.

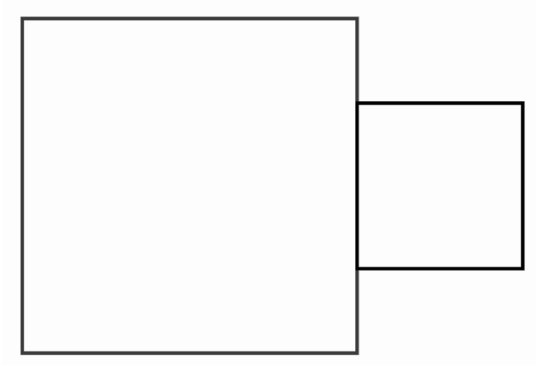


Figure 23-7. *A box with an exit*

Script 23-8. *Visualizing a box containing an exit*

```
| box rm rm2 exit |
box := Rectangle center: World center extent: 200@200.
rm := RectangleMorph new.
rm bounds: box.
rm color: Color transparent.
rm setBorderWidth: 2 borderColor: Color blue.
rm openInWorld.
exit := Rectangle origin: (box topRight + (2@50)) extent: 100@100.
rm2 := RectangleMorph new.
rm2 bounds: exit.
rm2 color: Color transparent.
rm2 setBorderWidth: 2 borderColor: Color black.
rm2 openInWorld.
```

Define a method to avoid having to repeat this code over and over. You might create a method named `escaping: aBox withExit: aExit` that checks first whether the next move would be contained in the rectangle representing the exit, and if this is the case, the method lets the animal escape.

Staying in a Healthy Environment

Have you ever wondered how slime bacteria manage to stay in moist places and avoid dry places? All right, perhaps you have not. In any case, I propose to simulate the strategies that certain basic life forms such as bacteria use to stay in an area where they have a greater chance of survival. I propose that you model two simple strategies discovered in the late 1950s using only changes in speed and direction. Note that when I talk about speed, I mean the distance that the animal can move at each step, which in our case will be the distance in pixels passed to the `go:` method.

The first strategy that the bacteria can follow is to change direction randomly in all circumstances but to change speed depending on the degree of environmental health. This strategy makes sense, because when a bacterium considers a region healthy, it will take more time for it to move a given distance, and hence it will stay longer in a healthy environment than in an unhealthy one. A possible path for a bacterium is shown in Figure 23-8.

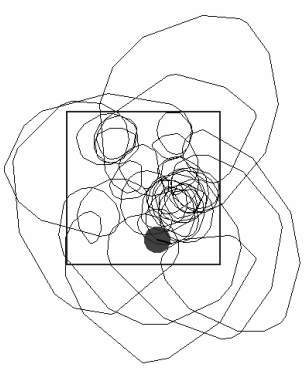


Figure 23-8. *First strategy. The bacterium increases its speed when it finds itself in an unhealthy environment. It changes its direction at a constant rate. Here the speed in a healthy environment is a random number up to 25 pixels,while in an unhealthy environment it is 100. The interior of the rectangle represents the healthy environment.*

The second strategy is the opposite. The bacterium moves at constant speed but changes its direction depending on the health of the environment. It will change its direction on average by a larger angle when it finds a healthy place. Therefore, it has more chance to retrace its steps and remain within a smaller region. Certain very simple bacteria use this strategy to stay in an environment where they can find food. A possible path for a bacterium is shown in Figure 23-9.

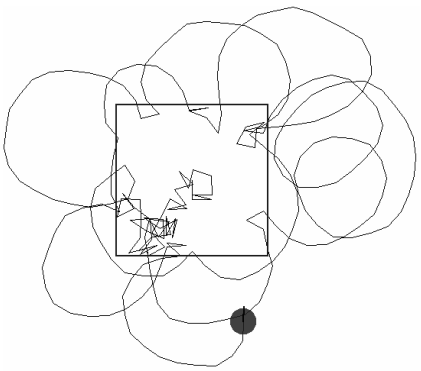


Figure 23-9. *Second strategy. The bacterium increases the range of its directional change in a healthy environment. Here the speed is 25, and the change in direction is at most 36 in an unhealthy environment, and 360 in a healthy one.*

To implement these ideas you just have to imagine that the rectangle we have been using represents a healthy region. For the first strategy, you could define a method named `stayAtConstantAngleNTimes: aNumber in: aRegion`, as shown in Script 23-9. Experiment with different values of the angle through which the bacterium can turn.

Script 23-9.

```
| bacterium |
Bot clearWorld.
bacterium:= Bot new.
bacterium stayAtConstantAngleNTimes: 500
        in: (Rectangle center: bacterium center extent: 200@200)
```

Method 23-6 shows a possible solution.

Method 23-6. *A bacterium changes its speed to remain in a healthy environment.*

```
stayAtConstantAngleNTimes: n in: aRectangle
  "The receiver tries to stay in a healthy environment by changing its speed,
   repeat n times."

  self box: aRectangle.
  n timesRepeat:
     [ (aRectangle containsPoint: self center)
         ifTrue: [ self go: 25 atRandom ]
         ifFalse: [ self go: 100 atRandom ].
      self turn: 25 ]
```

Further Experiments

We could imagine that the animal decreases its speed depending on the distance from the healthy zone. You might also introduce a bit of random behavior in the criterion (speed or turning angle) that otherwise does not change. You also could introduce the possibility for the bacterium to turn both clockwise and counterclockwise, as we already have implemented earlier in this chapter. As Figure 23-10 shows, the second strategy is not particularly efficient; the bacterium has no way of "knowing" whether it is moving in the direction of a good area, so it may end up staying outside of a healthy area for a long time and die as a consequence. Propose some other approaches to solve this aspect of the problem.

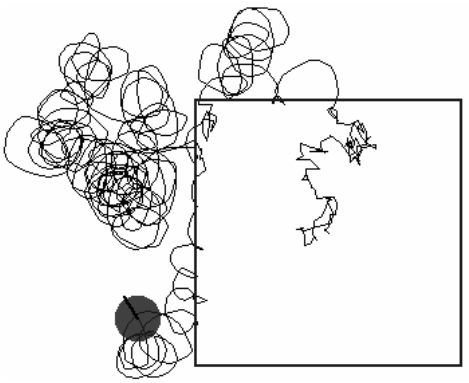


Figure 23-10. *Second strategy. Here the speed is a random number at most 10; the bacterium's change in direction is 36 degrees if the environment in unhealthy, and 360 if it is healthy.*

Finding Food

Now I would like to model different ways that an animal could search for food. A first approach is based on the fact that an animal can locate its food visually. Again you can represent the food area by a rectangle.

Comparing Distance

Imagine that an animal can evaluate the distance from a food source, which we will model as the animal's distance from the center of the rectangle used to represent the area in which food is located. Note that to obtain the distance between two points, you can use the method `dist:`. For example, `100@100 dist: 200@200` returns the distance between the points `100@100` and `200@200`. You can obtain the distance between a robot and a point using the method `distanceFrom: aPoint`.

Implement a method named, for example, `findFoodAreaByDistance: aFoodRectangle` that determines whether by walking one step forward the animal gets closer to the food area. When it is getting closer, it continues to move, but if it is getting farther away, it changes its direction by a certain fixed amount. Script 23-10 shows how the method might be used.

Script 23-10. *Executing* `findFoodAreaByDistance:`

```
| animal food |
animal := Bot new.
food := Rectangle center: 400@250 extent: 30@30.
animal findFoodArea: food
```

The behavior that I have described is somewhat naive, because there is no guarantee that turning through a certain angle will lead to an improved situation. Figures 23-11 and 23-12 present some traces. The right panel of Figure 23-12 shows a situation in which the animal comes within a certain distance of the food and then continually circles about the food rectangle without getting any closer. Try to figure out some solutions to this problem. Method 23-7 presents one possible definition of the method `findFoodAreaByDistance:`.

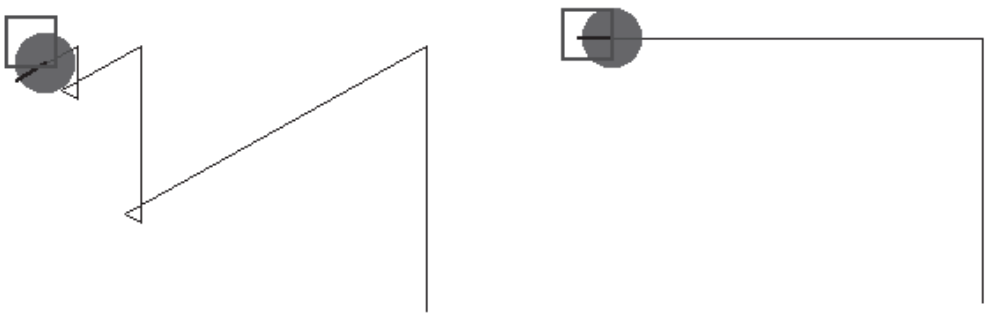


Figure 23-11. *Finding food by comparing distances and simply turning. Left: turn 120 degrees. Right: turn 90 degrees.*

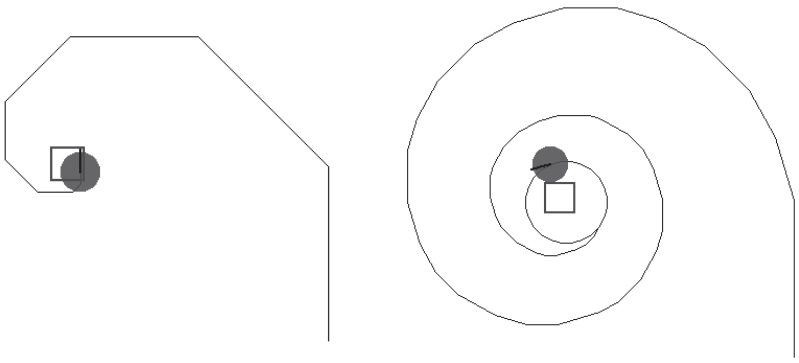


Figure 23-12. *Finding food by comparing distances and simply turning. Left: turn 45 degrees. Right: turn 15 degrees.*

Method 23-7. *An animal approaches a food source by trying to decrease its distance from the food.*

```
findFoodAreaByDistance: foodRectangle

  | food move |
  self box: foodRectangle.
  move := 10.
  food := foodRectangle center.
  [ (foodRectangle containsPoint: self center)
      or: [ self anyButtonPressed ] ] whileFalse:
            [ ((self positionIfGo: move) dist: food) > (self distanceFrom: food)
                  ifTrue: [ self turnLeft: 15 ]
                  ifFalse: [ self go: move ] ]
```

Further Experiments

Here are some ideas for additional experiments. Change the position of the food or the direction of the animal at the beginning of its walk in Script 23-10.

Improve the behavior implemented by Method 23-7 so that once the animal realizes that it is moving in the wrong direction, it will not turn naively in a random direction but will check first to determine whether by turning one way it improves its situation and then changes direction accordingly.

To help you in your experimenting, do not hesitate to define new methods. For example, you could define a method `positionIfGo: aDistance andTurn: anAngle` that returns the position where the receiver would be if it were to turn through `anAngle` and move forward `aDistance` (see Method 23-8).

Method 23-8. *Find the position the receiver would be in if it turned through a certain angle and moved a certain distance.*

```
positionIfGo: aDistance andTurn: anAngle
  "Return the position where the receiver would be if it turned
  through anAngle and moved forward aDistance"

  | position |
  self turn: anAngle.
  position := self positionInDirectionForDistance: aDistance.
  self turn: anAngle negated.
  ^ position
```

We have been talking about the distance between the food and the animal, but it is unlikely that an animal has a way to make such precise measurements. Nevertheless, animals can estimate the distance from a food source in a variety of ways, such as the intensity of the smell of the food. For such an animal, reducing the distance to its food is equivalent to increasing the intensity of the smell.

Taking One's Bearings

In fact, a good way for an animal to be sure of reaching its food is to know exactly where the food is located. In such a case, the best strategy is to "keep your eyes on the prize" by constantly looking at the food and moving in its direction. Implement this approach using the method `pointAt:` and introduce some random movements to make the simulation a bit more realistic.

Now you can introduce the notion of speed and perturbation in the animal's trajectory. Define a method `keepABearing: aRectangle moving: aDistance turning: anAngle` that keeps the animal always pointing to the center of the food area but moving and turning by a constant amount. Figures 23-13 and 23-14 show some results of this approach. With these constraints can you guess which is the more efficient way of reaching food: having a high speed and turning through a large angle or having a low speed and turning through a small angle? Of course, this simulation does not take into account that the food may move too.

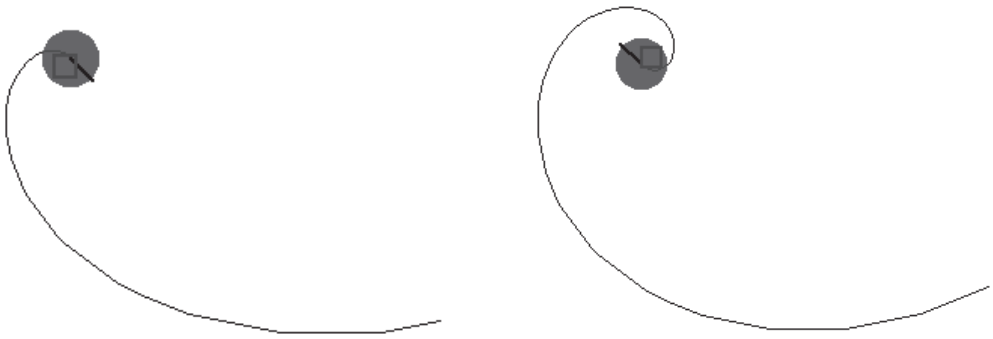


Figure 23-13. *Finding food by always pointing toward it. Left: speed 5 pixels turning 45 degrees. Right: speed 5 pixels turning 60 degrees.*

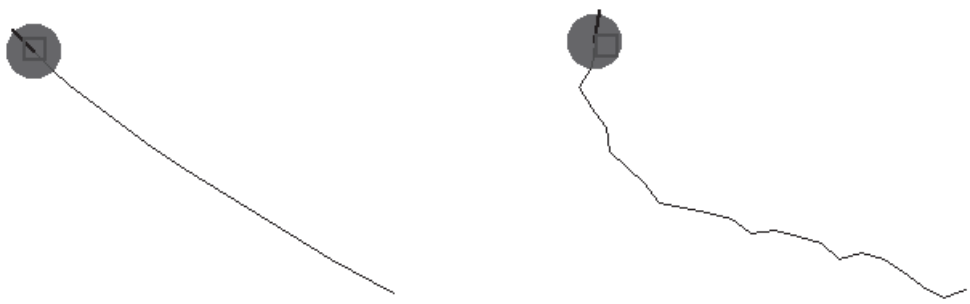


Figure 23-14. *Finding food by always pointing toward it. Left: speed 15 pixels turning 5 degrees. Right: speed 5 pixels turning 60 degrees with random angle.*

The left panel in Figure 23-14 shows that being able to point at the food with a small perturbation is one of the fastest approaches to reaching it. However, the speed has to be reasonable. Carry out some experiments with high speed to see whether this assertion is true. To obtain a more realistic simulation, introduce randomness in the angle and speed that could represent factors such as wind. For example, I introduced some randomness in the right panel in Figure 23-14. In addition to introducing randomness, implement the possibility that the angle may vary in both directions, as we have discussed earlier in this chapter.

Simulating Vision

In the previous experiment, the animal could locate its food without any constraints. Now we would like to be a bit more realistic in our simulation. When an animal identifies its food by eye, it has a restricted angle of vision that prevents it from simply moving straight to the food. Imagine that our animal now has a single eye that is represented by an angle of vision within which the animal can see its food, as shown in Figure 23-15.

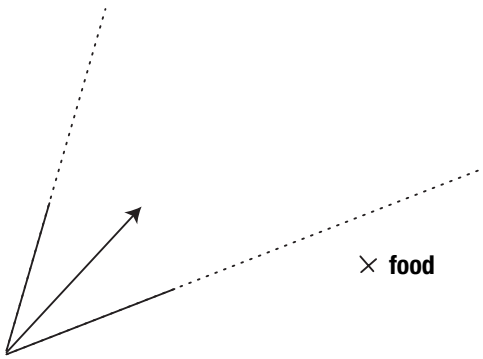


Figure 23-15. *The food lies outside the angle of vision, so the animal cannot see it.*

To define this behavior, I use the method `angleToPointAt:`, which returns the angle through which the animal should turn in order to point toward a given point. You could then decide, for example, that if the animal does not see any food, it turns around, and if it does see food, it moves straight in that direction. You need to be able to determine whether the angle through which the animal needs to turn is smaller than half of its view range. This is what the expression `(self angleToPointAt: aRectangle center) abs < (viewRange /2)` does in Method 23-9. The expression `self angleToPointAt: aRectangle center) abs` returns the angle between the animal's current direction and its food. Now put all of these ideas together and define the method `lookAndFindFoodAt: aRectangle viewRange: viewRange turning: anAngle`, which implements this behavior. Method 23-9 provides a possible solution, but try to invent your own approach. Some traces are shown in Figures 23-16 and 23-17.

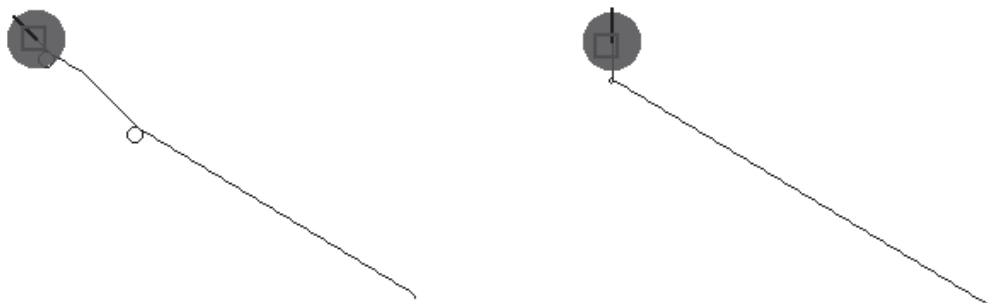


Figure 23-16. *Finding food by turning only when the food is not seen. Left: vision angle 10 degrees, turning angle 15 degrees. Right: vision angle 40 degrees, turning angle 60 degrees.*

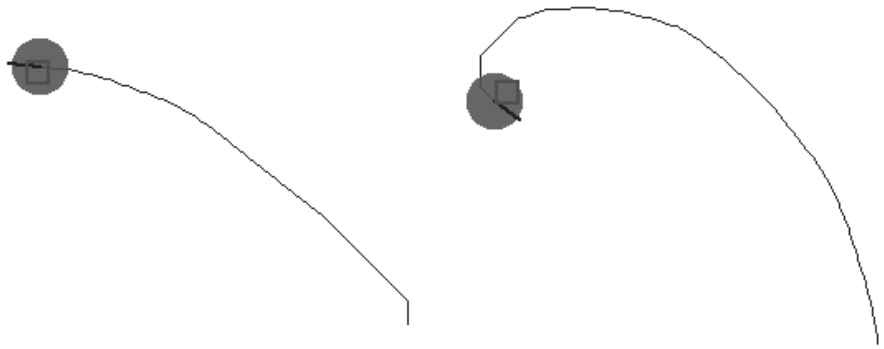


Figure 23-17. *Finding food by turning only when the food is not seen. Left: vision angle 15 degrees, turning angle 2 degrees. Right: vision angle 35 degrees, turning angle 3 degrees.*

This approach is again rather naive, since it may loop (see Figure 23-18, right panel). Indeed, the change in angle does not necessarily lead to a better situation. I also suggest that you make the animal move continuously and not simply turn around when it does not see its food.

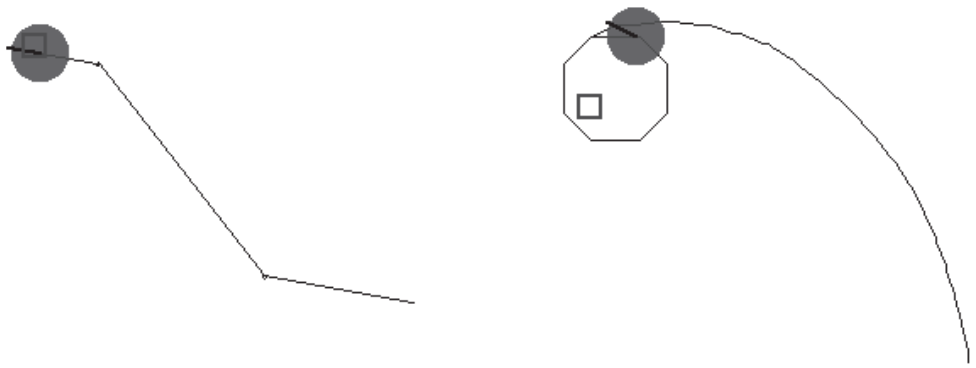


Figure 23-18. *Finding food by turning only when the food is not seen. Left: vision angle 35 degrees, turning angle 80 degrees. Right: vision angle 45 degrees, turning angle 2 degrees.*

Method 23-9

lookAndFindFoodAt: aRectangle viewRange: viewRange turning: anAngle

```
[ (aRectangle containsPoint: self center)
   or: [ self anyButtonPressed ] ] whileFalse:
      [ (self angleToPointAt: aRectangle center) abs < (viewRange /2)
                ifTrue: [ self go: 15 ]
                ifFalse: [ self turn: anAngle ] ]
```

Summary

The approaches to modeling animal behavior presented in this chapter are simple, but already they produce some interesting results. In general, the introduction of perturbation helps to simulate real behavior. For further projects, you might combine several behaviors and try to link certain inputs such as the angle of vision. Many other aspects of animal behavior can be modeled based on the examples presented in this chapter. I wish you lots of fun.