11 Artificial Life

In This Chapter

- Introduction
- Simulating Food Chains
- Food Chain Model
- Competition
- Sample Iteration
- Source Discussion
- Sample Results
- Interesting Strategies
- Adjusting the Parameters
- Summary
- References
- Resources

Artificial life (Alife) is a term coined by Chris Langton [Langton03] to describe a wide variety of computational mechanisms used to model natural systems. Alife has been used to model agents that trade resources in artificial economies, insect ecologies, animal behavior, and entities negotiating with one another to study models in game theory. In this chapter, we'll investigate Alife and then implement a simulation that demonstrates agents in a food chain competing in an artificial environment.

INTRODUCTION

Although Alife is a large discipline with various concerns, we'll focus here on what is called *synthetic ethology*, which is defined most succinctly by Bruce MacLennan:

> Synthetic ethology is an approach to the study of animal behavior in which simple, synthetic organisms are allowed to behave and evolve in a synthetic world. Because both the organisms and their worlds are synthetic, they can be constructed for specific purposes, particularly for testing specific hypotheses. [MacLennan03]

Artificial life can then be described as the theory and practice for biological-system modeling and simulation. One hope of researchers working with Alife is that by modeling biological systems, we can better understand why and how they work. Through the models, researchers can manipulate their environments to play "what if" games to learn how systems and environments respond to change.

SIMULATING FOOD CHAINS

A food chain describes the hierarchy of living organisms in an ecosystem. For instance, consider a simple abstracted food chain comprising three entities. At the bottom of the food chain is the plant. It derives its energy from the environment (rain, soil, and sun). The next level is herbivores—a herbivore consumes plant life to survive. Finally, at the top are carnivores. Carnivores, within this simulation, consume herbivores to survive. Ignoring the effect of dead herbivores and carnivores on the environment, the food chain can be illustrated as shown in Figure 11.1.

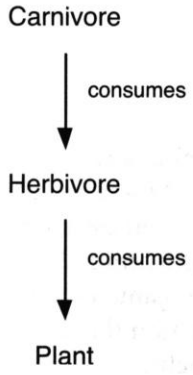


FIGURE 11.1 Simple food chain.

By viewing Figure 11.1 as a dependency graph, it should be clear that a delicate balance exists between the entities. What happens if the abundance of plant life

diminishes through a drought or other natural or artificial event? The lack of plant life affects the sustainability of herbivores in the environment, resulting in a decrease in their population. This effect cascades up through the food chain, ultimately affecting the carnivore population at the top. This balance can be modeled and studied within the domain of Alife and synthetic ethology.

FOOD CHAIN MODEL

To model a simple food chain, a number of aspects of the simulation must be defined, including the environment (the physical space in which the agents interact), the agents themselves (and their perception and actuation within the environment), and a set of laws that describes how and when interaction takes place. These elements will be described in the following sections.

Overview

Based on our discussion of the simple food chain, our simulation will consist of an environment and three types of entities. A plant is a stationary food source (for herbivores) that exists within the environment. Herbivores are migratory agents that sense their environment and can eat plants. The predators of herbivores are carnivores, which are the other migratory agent in the environment. Carnivores can eat only herbivores, and herbivores can eat only plants. If either agent lives for a certain amount of time in the environment without eating, the agent dies of starvation. When an agent consumes enough food, it is permitted to reproduce, which creates a new agent (of the particular type) in the environment. Through reproduction, evolution occurs through the mutation of the agents' brains (simple neural network).

It's important to note that the agents initially have no idea how to survive in the environment. They don't know that eating will allow them to live longer. Nor do they know that they must avoid their predators. All of these details must be experienced and learned by the agents through a simple form of evolution.

The following sections will discuss the elements of the simulation in detail.

Environment

Agents live in a grid world whose edges are connected in a toroid fashion. When an agent moves beyond the edge of a particular dimension, it reappears on the other side (see Figure 11.2).

Plants occupy unique cells in the environment, but it is possible for a single cell to be occupied by one or more agents (herbivore and/or carnivore).

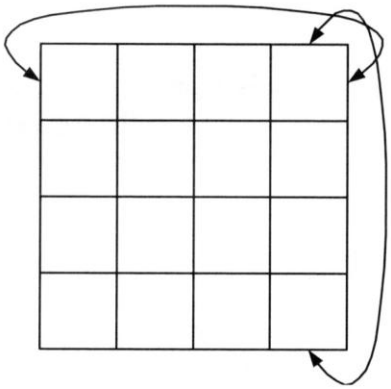


FIGURE 11.2 The toroid grid world for the food chain simulation.

Anatomy of an Agent

An agent is a generic entity in the simulation. It is defined as being a particular type (herbivore or carnivore), but the way it perceives the environment and how it acts on it are the same (see Figure 11.3). The agent can be thought of as a simple system with a set of inputs (its perception of the world), a reaction depending on its perception (its brain), and an actuation into the environment (taking its particular action).

The agent, as depicted by Figure 11.3, consists of three distinct parts: sensors, perception (determining which action to take based on the sensors), and actuation (taking the action). Note that the agent model is reactionary; it reacts to the environment. Agents have no capacity to plan, and other than their ability to reproduce, they have no ability to learn. Even with this simple mode, learning occurs through what is known as Larmarckian inheritance. Through reproduction, the characteristics of the parent are passed down to its progeny.

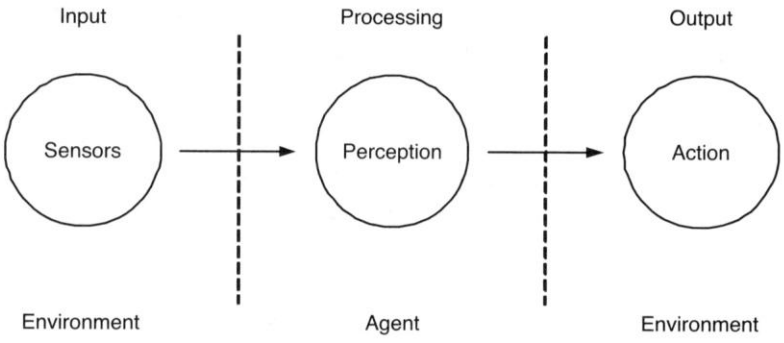


FIGURE 11.3 Agent systems model.

Jean-Baptiste Lamarck (1744–1829) proposed an alternate mechanism for evolution than that offered by Charles Darwin. Instead of the process of natural selection driving the gradual improvement of the species, Lamarck believed that heredity, or the inheritance of acquired traits, drove the process.

Sensors

Agents in the environment can sense what is around them. An agent cannot view the entire environment—only a set of local cells can be sensed (see Figure 11.4).

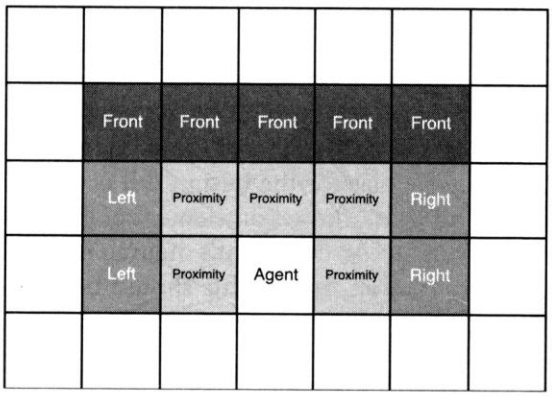


FIGURE 11.4 An agent's area of perception (facing North).

The local environment that the agent can perceive is split into four separate areas. The area local to the agent, *proximity*, is the area in which the agent can act (such as eating an object). The area in front of the agent (5 cells) makes up the *front*. Finally, the two cells at the left and right edges are *left* and *right*, respectively.

The agent is provided with a numerical count of the objects in view in each of the distinct areas. Therefore, for the four separate areas, three quantities are provided for each to identify the types of objects present (plants, herbivores, and carnivores), resulting in 12 inputs to the agent.

Actuators

The agent can perform a limited set of actions in the environment. It can move a single step (in a given direction), turn left or right, or eat an object in its local proximity. The action performed by the agent is provided through its brain, as a function of the inputs provided at the sensory stage.

Agent Brain

The brain of the agent could be one of a variety of computational constructs. Existing Alife simulations have used finite automata (state machines), classifier systems, or neural networks. In keeping with the biological motivation of this simulation, we'll use a simple, fully interconnected, winner-takes-all neural network (as was used in Chapter 5) as the behavioral element of the agent. Figure 11.5 shows a complete network for the agent.

Recall that sensor inputs represent the count of the objects in view in a particular area. After each of the inputs has been captured from the environment, we propagate these inputs through the network to the outputs. This is done using Equation 11.1.

$$o_j = b_j + \sum_{i=0}^{n} u_i w_{i,j} \tag{11.1}$$

In other words, for each output cell (o_j) in the network, we sum the products of the input cells (u_i) by the weights of the connections from the inputs cells to the output cells (w_{ij}). The bias for the output cell is also added. The result is a set of values in the output cells. The action element of the agent then uses these values.

The initial weights of the agent's neural network are selected randomly, but through reproduction the weights should become tuned for survival in the environment.

Agent Action Selection

Recall that the agent can perform one of four actions, as defined by the output cells of the neural network. The process of action selection is finding the output cell with the largest value and performing this action. This is the winner-takes-all aspect of the network. Once the action has taken place, the environment is modified (if warranted by the agent's action), and the process continues.

Energy and Metabolism

Agents survive in the environment given adequate energy. When an agent's internal energy falls to 0, the agent dies. Agents create energy by eating other objects in the environment. An agent can eat only an object that is defined as legal per the food chain. Carnivores can eat only herbivores; herbivores can eat only plants. Agents also have a metabolism—a rate at which they consume energy to stay alive. Carnivores consume a single energy unit each time they take a step, whereas herbivores consume two energy units. This means that herbivores must consume twice as much food as carnivores to stay alive. Although carnivores do not need to consume as much food, they do have to find it. Herbivores have an advantage in that their food does not move in the environment; however, they must still locate it.

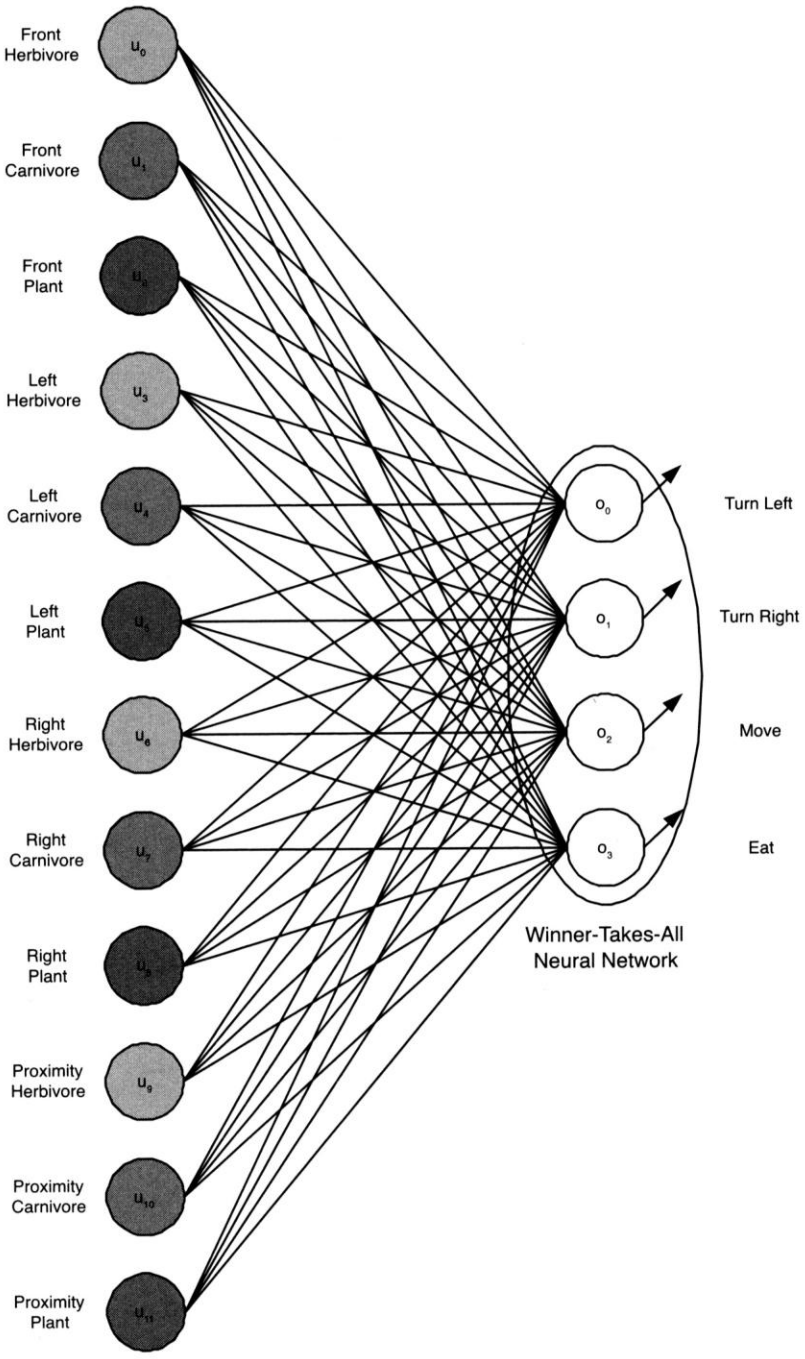


FIGURE 11.5 Winner-takes-all neural network as the agent brain.

Reproduction

When an agent consumes enough food to reach 90 percent of its maximum energy, it is permitted to asexually reproduce. Reproduction permits those agents who are able to successfully navigate their environments and stay alive to create offspring (natural selection). By creating offspring, the agents evolve through random mutation of the weights of their neural networks. No learning takes place within the environment, but the capability of an agent to reproduce means that its neural network is passed down to its child. This process mimics Lamarckian evolution in which the characteristics of the agent are passed down to the child (the child inherits the neural network of the parent).

When a parent reproduces, it is not without consequences. The parent and the child share the available energy of the parent (cutting the parent's energy in half). This limitation keeps an agent from continually reproducing and provides an implicit timer to avoid continual reproduction of the agent.

Death

Agents can die in one of two ways. Either the agent is unable to find sufficient food to sustain itself and starves to death, or an agent further up the food chain eats it. In either case, the agent in question, including its network and learned behaviors, is removed from the simulation.

COMPETITION

During the simulation, a subtle game of competition occurs. Carnivores slowly evolve neural networks that are good at locating and eating herbivores. At the same time, herbivores slowly evolve neural networks that find plants in the environment while simultaneously avoiding carnivores. Though these strategies are visible by watching the simulation, analyzing the resulting neural networks can also provide insight into the strategies. We'll peer into some of these networks in the results section to better understand the agents' motives.

SAMPLE ITERATION

Now let's look at a sample action-selection iteration of an evolved agent. In this example, we'll look at a herbivore that has evolved during the simulation. Although not perfect, this particular agent survived in the environment for more than 300 time steps. It was able to do this by avoiding predators (carnivores) as well as finding and eating plants. Figure 11.6 shows the neural network for this agent.

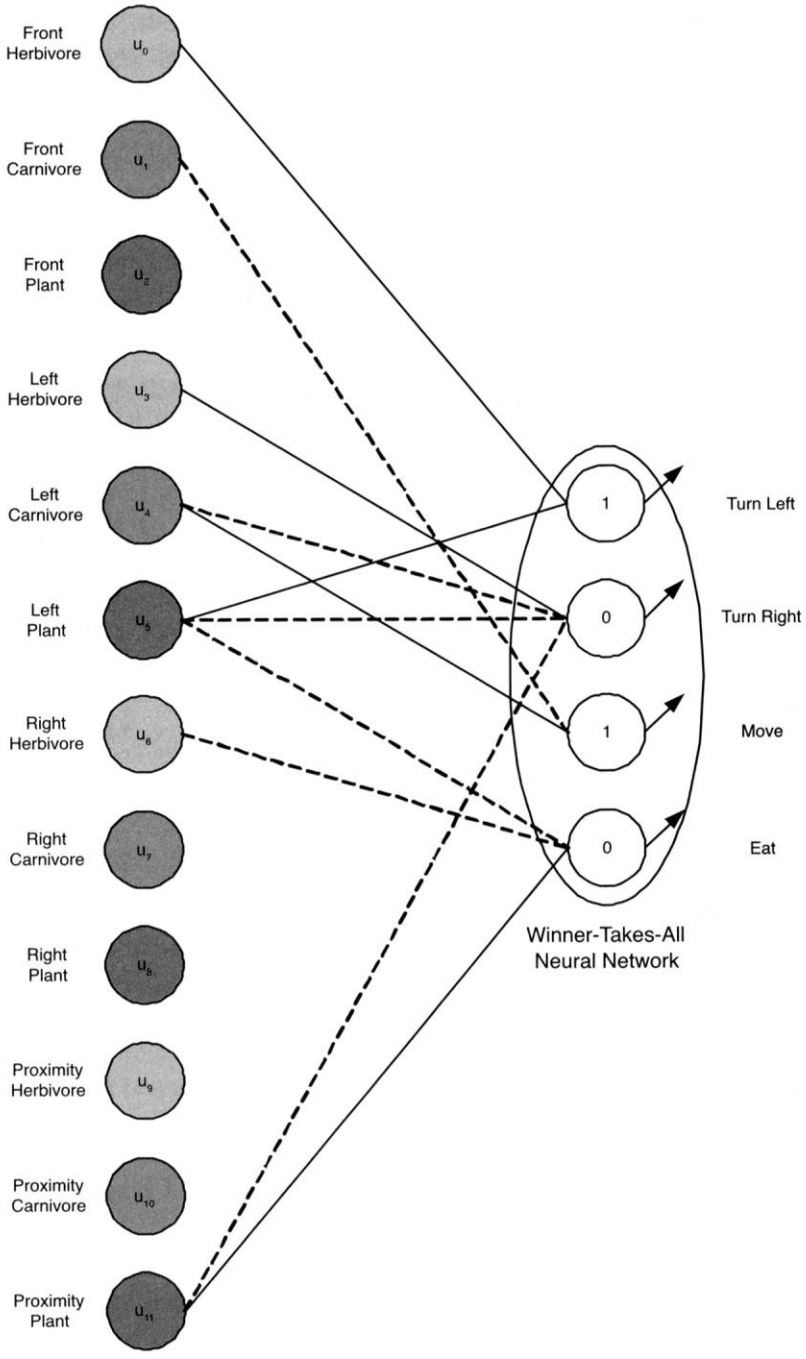


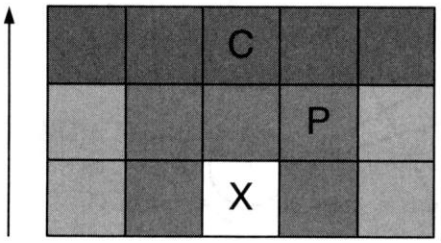
FIGURE 11.6 The neural network for evolved herbivore.

Solid lines in the neural network are excitory connections, whereas dashed lines are inhibitory. In the output cells are the biases that are applied to each output cell as it is activated. From Figure 11.6, we can see that an excitory connection exists for the *eat* action when a plant is in proximity (a plant can be eaten only when it is in proximity of the agent). Also of interest is the inhibitory connection for the *move* action when a carnivore is in front. This is another beneficial action for the survival of the herbivore.

An agent's actions are not formed by a single connection. Instead, the action with the largest value is permitted to fire based on the combination of the sensor inputs. Let's look at a couple of iterations of the herbivore described by the neural network in Figure 11.6.

Recall from Equation 11.1 that we multiply the weights vector (for the particular action) by the inputs vector and then add the bias.

In our first epoch, our herbivore is presented with the scene as shown in Figure 11.7. The different zones are shaded to illustrate them (recall Figure 11.4). In this scene, the *X* denotes the location of the herbivore (its reference point to the scene). A plant is located within *proximity* and a carnivore is located in *front*.




```
                    HF CF PF HL CL PL HR CR PR HP CP PP
weights_left      = { 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0}
weights_right     = { 0, 0, 0, 0,-1,-1, 0, 0, 0, 0, 0,-1}
weights_move      = { 0,-1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0}
weights_eat       = { 1, 0, 0, 0, 0,-1,-1, 0, 0, 0, 0, 1}

bias_left         = 1
bias_right        = 0
bias_move         = 1
bias_eat          = 0

inputs            = { 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1}
```

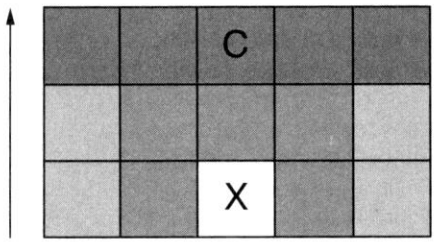
```
Left  = 1 +  0 =  1
Right = 0 + -1 = -1
Move  = 1 + -1 =  0
Eat   = 0 +  1 =  1
```

FIGURE 11.7 Herbivore at time t_0.

The first step is assessing the scene. We count the number of objects of each type in each of the four zones. As shown in Figure 11.7, the weights and inputs are labeled with type/zone (HF is herbivore-*front*, CF is carnivore-*front*, PP is plant-*proximity*, and so on). In this example, our input's vector has non-zero values in only two of the vector elements. As shown in the scene, a carnivore is in the *front* zone and a plant in *proximity*.

To evaluate the function to take, we multiply the input vector by the weights vector for the defined action and then add in the respective bias. This process is shown in Figure 11.7. Per our action-selection algorithm, we take the action with the largest resulting value. In the case of a tie, as is shown in Figure 11.7, we take the largest value that appeared last. In this case, the *eat* action is performed (a desirable action given the current scene).

Once the plant is consumed, it disappears from the scene. The herbivore is then left with the scene as shown in Figure 11.8.




```
                    HF CF PF HL CL PL HR CR PR HP CP PP
weights_left      = { 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0}
weights_right     = { 0, 0, 0, 0,-1,-1, 0, 0, 0, 0, 0,-1}
weights_move      = { 0,-1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0}
weights_eat       = { 1, 0, 0, 0, 0,-1,-1, 0, 0, 0, 0, 1}

bias_left         = 1
bias_right        = 0
bias_move         = 1
bias_eat          = 0

inputs            = { 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

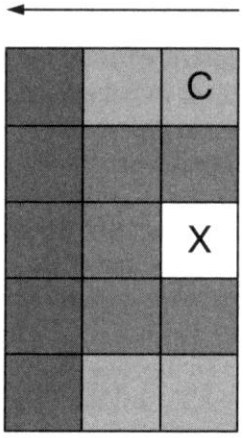
```
Left  = 1 +  0 =  1
Right = 0 +  0 =  0
Move  = 1 + -1 =  0
Eat   = 0 +  0 =  0
```

FIGURE 11.8 Herbivore at time T_1.

The scene is assessed once again, and as shown, the plant no longer exists, but the carnivore remains, which is illustrated by the inputs. (Changes from the prior iteration are shown in boldface.) We again compute the output cells of the neural network by multiplying the input vector times the respective weights vector. In this case, the largest value is associated with the *left* action. Given the current scene, a step to the *left* is a reasonable action to take.

Finally, in Figure 11.9, we see the final iteration of the herbivore. Note that the agent's view has changed because the agent changed direction in the previous iteration. With the change in the scene, the inputs have now changed as well. Instead the carnivore being in the *front* zone, it has moved to the *right* zone due to our new view of the scene.

Computing the output cells once again results in the move action being the largest and, more important, last in this case. The herbivore's behavior allowed it to find food and eat it and then avoid a carnivore within its field of view. From this short demonstration of the herbivore's neural network, it is not surprising that it was able to navigate and survive in the environment for a long period of time.




```
                    HF CF PF HL CL PL HR CR PR HP CP PP
weights_left      = { 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0}
weights_right     = { 0, 0, 0, 0,-1,-1, 0, 0, 0, 0, 0,-1}
weights_move      = { 0,-1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0}
weights_eat       = { 1, 0, 0, 0, 0,-1,-1, 0, 0, 0, 0, 1}

bias_left         = 1
bias_right        = 0
bias_move         = 1
bias_eat          = 0

inputs            = { 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0}
```

```
Left  = 1 +  0 =  1
Right = 0 +  0 =  0
Move  = 1 +  0 =  1
Eat   = 0 +  0 =  0
```

FIGURE 11.9 Herbivore at time T_2.

The source for the Alife simulation can be found on the accompanying CD-ROM at /software/ch11/.

SOURCE DISCUSSION

The source code for the Alife simulation is straightforward. Let's first walk through the structures for the simulation that describe not only the environment but also the agents and other objects in the environment.

Listing 11.1 shows the agent type. Most elements of this structure are self-explanatory; `type` defines the agent as herbivore or carnivore, `energy` is the available energy to the agent, `age` is the age in time steps, and `generation` is a value that describes this agent in the lineage of agents that reproduced.

The location of the agent (as defined by the `locType` type) specifies the x- and y-coordinates of the agent in the landscape. The `inputs` vector defines the values of the inputs to the neural network during the perception stage. The `actions` vector is the output layer of the neural network that defines the action to be taken. Finally, `weight_oi` (weight value from output to input) and `biaso` represent the weights and biases for the output layer of the network. We'll see later how the weights are structured in this single vector.

LISTING 11.1 Agent Types and Symbolics

```
typedef struct {
  short type;
  short energy;
  short parent;
  short age;
 short generation;
  locType location;
  unsigned short direction;
  short inputs[MAX_INPUTS];
  short weight_oi[MAX_INPUTS * MAX_OUTPUTS];
  short biaso[MAX_OUTPUTS];
  short actions[MAX_OUTPUTS];
} agentType;
#define TYPE_HERBIVORE  0
#define TYPE_CARNIVORE  1
#define TYPE_DEAD       -1
typedef struct {
  short x;
  short y;
} locType;
```

The `input` vector defines the inputs as object and zone (such as herbivore and front). To give an agent the ability to differentiate each of these elements separately, each element is given its own input into the neural network. The outputs are also well defined, with each output cell in the `output` vector representing a single action. Listing 11.2 provides the symbolic constant definitions for the input and output cells.

LISTING 11.2 Sensor Input and Action Output Cell Definitions

```
#define HERB_FRONT            0
#define CARN_FRONT            1
#define PLANT_FRONT           2
#define HERB_LEFT             3
#define CARN_LEFT             4
#define PLANT_LEFT            5
#define HERB_RIGHT            6
#define CARN_RIGHT            7
#define PLANT_RIGHT           8
#define HERB_PROXIMITY        9
#define CARN_PROXIMITY       10
#define PLANT_PROXIMITY      11
```

```
#define MAX_INPUTS              12
#define ACTION_TURN_LEFT         0
#define ACTION_TURN_RIGHT        1
#define ACTION_MOVE              2
#define ACTION_EAT               3
#define MAX_OUTPUTS              4
```

The agent environment is provided by a three-dimensional cube. Three planes exist for the agents, with each plane occupied by a single type of object (plant, herbivore, or carnivore). The agent's world is still viewed as a two-dimensional grid, but three dimensions exist to more efficiently account for the objects present. Listing 11.3 provides the types and symbolics used to represent the landscape.

LISTING 11.3 Agent Environment Types and Symbolics

```
#define HERB_PLANE          0
#define CARN_PLANE          1
#define PLANT_PLANE         2
#define MAX_GRID            30
/* Agent environment in 3 dimensions (to have independent
 * planes for plant, herbivore and carnivore objects.
 */
int landscape[3][MAX_GRID][MAX_GRID];
#define MAX_AGENTS 36
#define MAX_PLANTS 30
agentType agents[MAX_AGENTS];
int agentCount = 0;
plantType plants[MAX_PLANTS];
int plantCount = 0;
```

The size of the grid, the number of agents that exist, and the amount of plant life are all parameters that can be manipulated for different experiments. The header file (`common.h`) includes a section for adjustable parameters.

Finally, a set of macros is provided in the header file for commonly used functions associated with random-number generation (see Listing 11.4).

LISTING 11.4 Simulation Macro Functions

```
#define getSRand()      ((float)rand() / (float)RAND_MAX)
#define getRand(x)      (int)((float)x*rand()/(RAND_MAX+1.0))
#define getWeight()     (getRand(9)-1)
```

The `getSRand` function returns a random number between 0 and 1, whereas `getRand` returns a number from 0 to x–1. Finally, the `getWeight` function returns a weight used for the agent neural networks, including biases used with the output cells.

We'll now walk through the source of the simulation. We begin with the `main` function, which has been simplified to remove the command-line processing and statistics collection. The unedited source can be found on the CD-ROM.

The `main` function sets up the simulation and then loops for the number of time steps allotted for the simulation within the header file (via the `MAX_STEPS` symbolic constant). The `simulate` function is the primary entry point of the simulation where life is brought to the agents and the environment (see Listing 11.5).

LISTING 11.5 `main` Function for the Alife Simulation

```
int main( int argc, char *argv[] )
{
  int i;
  /* Seed the random number generator */
  srand( time(NULL) );
  /* Initialize the simulation */
  init();
  /* Main loop for the simulation. */
  for (i = 0 ; i < MAX_STEPS ; i++) {
    /* Simulate each agent for one time step */
    simulate();
  }
  return 0;
}
```

The `init` function initializes both the environment and the objects in the environment (plants, herbivores, and carnivores). Note that when agents are initialized, the type of the agent is filled in. This step is performed so that the `initAgent` can be reused for other purposes in the simulation. With the type filled in, the `initAgent` function knows which kind of agent is being initialized and treats it accordingly (see Listing 11.6)

LISTING 11.6 Function `init` to Initialize the Simulation

```
void init( void )
{
  /* Initialize the landscape */
  bzero( (void *)landscape, sizeof(landscape) );
  bzero( (void *)bestAgent, sizeof(bestAgent) );
  /* Initialize the plant plane */
  for (plantCount = 0 ; plantCount < MAX_PLANTS ;
```

```
    plantCount++) {
      growPlant( plantCount );
    }
    /* Randomly initialize the Agents */
    for (agentCount = 0 ; agentCount < MAX_AGENTS ;
    agentCount++) {
      if (agentCount < (MAX_AGENTS / 2)) {
        agents[agentCount].type = TYPE_HERBIVORE;
      } else {
        agents[agentCount].type = TYPE_CARNIVORE;
      }
      initAgent( &agents[agentCount] );
    }
  }
```

The plant plane is first initialized by creating plants of a count defined by the `MAX_PLANTS` symbolic constant. Function `growPlants` provides this function (see Listing 11.7). Next, the agents are initialized. For the maximum number of agents allowed (defined by `MAX_AGENTS`), half of the space is reserved for each time. We initialize the herbivores first and then the carnivores. The actual initialization of the agents is provided by the `initAgent` function (shown in Listing 11.8).

The `growPlant` function finds an empty spot in the plant plane and places a plant in that position (see Listing 11.7). The function also ensures that no plant exists there already (so that we can control the number of unique plants available in the environment).

LISTING 11.7 Function `growPlant` to Introduce Foliage into the Simulation

```
void growPlant( int i )
{
  int x,y;
  while (1) {
    /* Pick a random location in the environment */
    x = getRand(MAX_GRID); y = getRand(MAX_GRID);
    /* As long as a plant isn't already there */
    if (landscape[PLANT_PLANE][y][x] == 0) {
      /* Update the environment for the new plant */
      plants[i].location.x = x;
      plants[i].location.y = y;
      landscape[PLANT_PLANE][y][x]++;
      break;
    }
  }
```

```
  return;
}
```

The agent planes are next initialized with the herbivore and carnivore species (see Listing 11.8). A reference pointer to an agent that represents the agent element to initialize is passed (as shown in Listing 11.6). Recall that the agent type has been defined already. We first initialize the `energy` of the agent to half of the maximum available to the agent. This step is done because when an agent reaches a large percentage of its maximum allowable energy, it is permitted to reproduce. Setting the `energy` to half of the maximum requires that the agent must quickly find food in the environment to reproduce. The `age` and `generation` are also initialized for a new agent. We keep a count of the number of agents within `agentTypeCounts` to ensure that we maintain a 50/50 split between the two agent species in the simulation. Next, we find a home for the agent using the `findEmptySpot` function. This function, shown in Listing 11.8, finds an empty element in the given plane (defined by the agent `type`) and stores the coordinates of the agent in the agent structure. Finally, we initialize the weights and biases for the agent's neural network.

LISTING 11.8 Function `initAgent` to Initialize the Agent Species

```
void initAgent( agentType *agent )
{
  int i;
  agent->energy = (MAX_ENERGY / 2);
  agent->age = 0;
  agent->generation = 1;
  agentTypeCounts[agent->type]++;
  findEmptySpot( agent );
  for (i = 0 ; i < (MAX_INPUTS * MAX_OUTPUTS) ; i++) {
    agent->weight_oi[i] = getWeight();
  }
  for (i = 0 ; i < MAX_OUTPUTS ; i++) {
    agent->biaso[i] = getWeight();
  }
  return;
}
void findEmptySpot( agentType *agent )
{
  agent->location.x = -1;
  agent->location.y = -1;
  while (1) {
    /* Pick a random location for the agent */
    agent->location.x = getRand(MAX_GRID);
```

```
    agent->location.y = getRand(MAX_GRID);
    /* If an agent isn't there already, break out of the loop
    */
    if (landscape[agent->type]
                [agent->location.y][agent->location.x] == 0)
      break;
  }
  /* Pick a random direction for the agent, and update the map
  */
   agent->direction = getRand(MAX_DIRECTION);
   landscape[agent->type][agent->location.y][agent-
   location.x]++;
   return;
}
```

Note in function `findEmptySpot` that the landscape is represented by counts. This records whether an object is contained at the coordinates of the grid. As objects die or are eaten, the `landscape` is decremented to identify the removal of an object.

We've now completed our discussion of the initialization of the simulation; now let's continue with the actual simulation. Recall that our `main` function (shown in Listing 11.5) calls the `simulate` function to drive the simulation. The `simulate` function (shown in Listing 11.9) permits each of the agents to perform a single action in the environment. The herbivores are simulated first and then the carnivores. This order gives a slight advantage to the herbivores, but because herbivores must contend with starvation and a predator, it seemed like a way to level the playing field, if only slightly.

LISTING 11.9 The `simulate` Function

```
void simulate( void )
{
  int i, type;
  /* Simulate the herbivores first, then the carnivores */
  for (type = TYPE_HERBIVORE ; type <= TYPE_CARNIVORE ;
  type++) {
    for (i = 0 ; i < MAX_AGENTS ; i++) {
      if (agents[i].type == type) {
        simulateAgent( &agents[i] );
      }
    }
  }
}
```

The simulate function (Listing 11.9) calls the simulateAgent function to simulate a single step of the agent. This function can be split into four logical sections: perception, network propagation, action selection, and energy test.

The perception algorithm is likely the most complicated of the simulation. Recall from Figure 11.4 that an agent's field of view is based on its direction and is split into four separate zones (*front*, *proximity*, *left*, and *right*). For the agent to perceive the environment, it must first identify the coordinates of the grid that make up its field of view (based on the direction of the agent) and then split these up into the four separate zones. We can see how this is done in the switch statement of the simulateAgent function (Listing 11.11). The switch statement determines the direction in which the agent is facing. Each call to the percept function sums the objects for the particular zone. Note that each call identifying HERB_<zone> represents the first plane for the zone (herbivore, carnivore, and then plant).

The percept call is made with the agent's current coordinates, the offset into the inputs array, and a list of coordinate offsets and bias. Note that when the agent is facing north, the north<zone> offsets are passed, but when the agent is facing south, we again pass the north<zone> offsets but with a -1 bias. This is done similarly with the west<zone>. The coordinate offsets for the agent to each of the zones are defined for a given direction, but they can be reversed to identify the coordinates in the opposite direction.

So what do we mean by this? Let's look at the coordinate offsets in Listing 11.10.

LISTING 11.10 Coordinate Offsets to Sum Objects in the Field of View

```
const offsetPairType northFront[]=
      {{-2,-2}, {-2,-1}, {-2,0}, {-2,1}, {-2,2}, {9,9}};
const offsetPairType northLeft[]={{0,-2}, {-1,-2}, {9,9}};
const offsetPairType northRight[]={{0,2}, {-1,2}, {9,9}};
const offsetPairType northProx[]=
      {{0,-1}, {-1,-1}, {-1,0}, {-1,1}, {0,1}, {9,9}};
const offsetPairType westFront[]=
      {{2,-2}, {1,-2}, {0,-2}, {-1,-2}, {-2,-2}, {9,9}};
const offsetPairType westLeft[]={{2,0}, {2,-1}, {9,9}};
const offsetPairType westRight[]={{-2,0}, {-2,-1}, {9,9}};
const offsetPairType westProx[]=
      {{1,0}, {1,-1}, {0,-1}, {-1,-1}, {-1,0}, {9,9}};
```

Two sets of coordinate offset vectors are provided, one for north and one for west. Let's take the northRight vector as an example. Let's say our agent is sitting at coordinates <11,9> within the environment (using an <y,x> coordinate system). Using the northRight vector as coordinate biases, we calculate two new coordinate

pairs (because <9,9> represents the list end): <7,11> and <6,11>. These two coordinates represent the two locations for the right zone given an agent facing north. If the agent were facing south, we would negate the `northRight` coordinates before adding them to our current position, resulting in <7,7> and <8,7>. These two coordinates represent the two locations for the right zone given an agent facing south.

Now that we've illustrated the coordinate offset pairs, let's continue with our discussion of the `simulateAgent` function (see Listing 11.11).

LISTING 11.11 Function `simulateAgent`

```
void simulateAgent( agentType *agent )
{
  int x, y;
  int out, in;
  int largest, winner;
  /* Use shorter names... */
  x = agent->location.x;
  y = agent->location.y;
  /* Determine inputs for the agent neural network */
  switch( agent->direction ) {
    case NORTH:
      percept(x, y, &agent->inputs[HERB_FRONT], northFront, 1);
      percept(x, y, &agent->inputs[HERB_LEFT], northLeft, 1);
      percept(x, y, &agent->inputs[HERB_RIGHT], northRight, 1);
      percept(x, y, &agent->inputs[HERB_PROXIMITY], northProx, 1);
      break;
    case SOUTH:
      percept(x, y, &agent->inputs[HERB_FRONT], northFront, -1);
      percept(x, y, &agent->inputs[HERB_LEFT], northLeft, -1);
      percept(x, y, &agent->inputs[HERB_RIGHT], northRight, -1);
      percept(x, y, &agent->inputs[HERB_PROXIMITY], northProx, -1);
      break;
    case WEST:
      percept x, y, &agent->inputs[HERB_FRONT], westFront,1);
      percept(x, y, &agent->inputs[HERB_LEFT], westLeft, 1);
      percept(x, y, &agent->inputs[HERB_RIGHT], westRight, 1);
      percept(x, y, &agent->inputs[HERB_PROXIMITY], westProx, 1);
      break;
    case EAST:
      percept(x, y, &agent->inputs[HERB_FRONT], westFront, -1);
      percept(x, y, &agent->inputs[HERB_LEFT], westLeft, -1);
      percept(x, y, &agent->inputs[HERB_RIGHT], westRight, -1);
```

```
    percept(x, y, &agent->inputs[HERB_PROXIMITY], westProx,  -1);
    break;
  }
  /* Forward propogate the inputs through the neural network
  */
  for ( out = 0 ; out < MAX_OUTPUTS ; out++ ) {
    /* Initialize the output node with the bias */
    agent->actions[out] = agent->biaso[out];
    /* Multiply the inputs by the weights for this output node
    */
    for ( in = 0 ; in < MAX_INPUTS ; in++ ) {
      agent->actions[out] +=
        ( agent->inputs[in] *
          agent->weight_oi[(out * MAX_INPUTS)+in] );
    }
  }
  largest = -9;
  winner = -1;
  /* Select the largest node (winner-takes-all network) */
  for ( out = 0 ; out < MAX_OUTPUTS ; out++ ) {
    if (agent->actions[out] >= largest) {
      largest = agent->actions[out];
      winner = out;
    }
  }
  /* Perform Action */
  switch( winner ) {
    case ACTION_TURN_LEFT:
    case ACTION_TURN_RIGHT:
      turn( winner, agent );
      break;
    case ACTION_MOVE:
      move( agent );
      break;
    case ACTION_EAT:
      eat( agent );
      break;
  }
  /* Consume some amount of energy.
   * Herbivores, in this simulation, require more energy to
   * survive than carnivores.
   */
  if (agent->type == TYPE_HERBIVORE) {
```

```
      agent->energy -= 2;
    } else {
      agent->energy -= 1;
    }
    /* If energy falls to or below zero, the agent dies.
     * Otherwise, we check to see if the agent has lived longer than
     * any other agent of the particular type.
     */
    if (agent->energy <= 0) {
      killAgent( agent );
    } else {
      agent->age++;
      if (agent->age > agentMaxAge[agent->type]) {
        agentMaxAge[agent->type] = agent->age;
        agentMaxPtr[agent->type] = agent;
      }
    }
  return;
}
```

Having discussed perception, we now continue with the remaining three stages of the `simulateAgent` function. The next step is to forward propagate our `inputs` collected in the perception stage to the output cells of the agent's neural network. This process is performed based on Equation 11.1. The result is a set of output cells representing a value calculated using the input cells and the weights between the inputs cell and output cells. We then select the action to take based on the highest output cell (in a winner-takes-all network fashion). A switch statement is used to call the particular `action` function. As shown in Listing 11.11, available actions are `ACTION_TURN_LEFT`, `ACTION_TURN_RIGHT`, `ACTION_MOVE`, and `ACTION_EAT`.

The final stage of agent simulation is an energy test. At each step, an agent loses some amount of energy (the amount differs for carnivores and herbivores). If the agent's energy falls to 0, the agent dies of starvation and ceases to exist in the simulation. If the agent survives, then its age is incremented. Otherwise, the agent is killed using the `killAgent` function.

We'll now walk through the functions referenced within the `simulateAgent` function, in the order in which they were called (`percept`, `turn`, `move`, `eat`, and `killAgent`).

Although the earlier discussion of `percept` may have given the impression that a complicated function was required, as is shown in Listing 11.12, this process is quite simple because much of the functionality is provided by the data structure; the code simply follows the data structure to achieve the intended function.

LISTING 11.12 The `percept` Function

```
void percept( int x, int y, short *inputs,
              const offsetPairType *offsets, int neg )
{
  int plane, i;
  int xoff, yoff;
    /* Work through each of the planes in the environment */
    for (plane = HERB_PLANE ; plane <= PLANT_PLANE ; plane++) {
    /* Initialize the inputs */
    inputs[plane] = 0;
    i = 0;
    /* Continue until we've reached the end of the offsets */
    while (offsets[i].x_offset != 9) {
      /* Compute the actual x and y offsets for the current
       * position.
       */
      xoff = x + (offsets[i].x_offset * neg);
      yoff = y + (offsets[i].y_offset * neg);
      /* Clip the offsets (force the toroid as shown by
       * Figure 7.2.
       */
      xoff = clip( xoff );
      yoff = clip( yoff );
      /* If something is in the plane, count it */
      if (landscape[plane][yoff][xoff] != 0) {
        inputs[plane]++;
      }
      i++;
    }
  }
  return;
}
int clip( int z )
{
  if (z > MAX_GRID-1) z = (z % MAX_GRID);
  else if (z < 0) z = (MAX_GRID + z);
  return z;
}
```

Recall that each call to `percept` provides the calculation of the number of objects in a given zone but for all three planes. Therefore, `percept` uses a `for` loop to walk through each of the three planes and calculates the sums based on the particular plane of interest. For a given plane, we walk through each of the coordinate

offset pairs (as defined by the `offsets` argument). Each coordinate offset pair defines a new set of coordinates based on the current position. With these new coordinates, we increment the `inputs` element if anything exists at the location. This means that the agent is aware only that at least one object exists at the coordinate for the given plane, but it does not know exactly how many objects exist.

Also shown in Listing 11.12 is the `clip` function. This function is used by `percept` to achieve a toroid (wrap) effect on the grid.

The `turn` function, shown in Listing 11.13, is simple—the agent changes direction. As is shown in Listing 11.13, given the agent's current direction and the direction to turn, a new direction results.

LISTING 11.13 The `turn` function

```
void turn ( int action, agentType *agent )
{
  /* Because our agent can turn only left or right, we
   * determine the new direction based on the current
   * direction and the turn action.
   */
  switch( agent->direction ) {
    case NORTH:
      if (action == ACTION_TURN_LEFT) agent->direction = WEST;
      else agent->direction = EAST;
      break;
    case SOUTH:
      if (action == ACTION_TURN_LEFT) agent->direction = EAST;
      else agent->direction = WEST;
      break;
    case EAST:
      if (action == ACTION_TURN_LEFT) agent->direction =
      NORTH;
      else agent->direction = SOUTH;
      break;
    case WEST:
      if (action == ACTION_TURN_LEFT) agent->direction =
      SOUTH;
      else agent->direction = NORTH;
      break;
  }
  return;
}
```

The move function is slightly more complicated. Using a set of offsets to determine the new coordinate position and the direction to determine which pair of offsets to use, a new set of coordinates is calculated. Also shown in Listing 11.14 is maintenance of the landscape. Prior to the agent's move, the landscape for the given plane (as defined by the agent type) is decremented to represent an agent moving from the location. Once the agent has moved, the landscape is updated again to show an agent now located at the coordinates in the given plane.

LISTING 11.14 The move Function

```
void move( agentType *agent )
{
  /* Determine new position offset based on current
  direction */
  const offsetPairType offsets[4]={{-1,0},{1,0},{0,1},{0,-1}};
  /* Remove the agent from the landscape. */
  landscape[agent->type][agent->location.y][agent-
  >location.x]--;
  /* Update the agent's X,Y position (including clipping) */
  agent->location.x =
    clip( agent->location.x + offsets[agent-
    >direction].x_offset );
  agent->location.y =
    clip( agent->location.y + offsets[agent-
    >direction].y_offset );
  /* Add the agent back onto the landscape */
  landscape[agent->type][agent->location.y][agent-
  >location.x]++;
  return;
}
```

The eat function is split into two stages: locating an object to eat in the agent's proximity (if one exists), and then the record keeping required to document and remove the eaten item (see Listing 11.15).

LISTING 11.15 The eat Function

```
void eat( agentType *agent )
{
  int plane, ax, ay, ox, oy, ret=0;
  /* First, determine the plane that we'll eat from based on
   * our agent type (carnivores eat herbivores, herbivores eat
   * plants).
```

```
 */
if (agent->type == TYPE_CARNIVORE) plane = HERB_PLANE;
else if (agent->type == TYPE_HERBIVORE) plane = PLANT_PLANE;
/* Use shorter location names */
ax = agent->location.x;
ay = agent->location.y;
/* Choose the object to consume based on direction (in the
 * proximity of the agent).
 */
switch( agent->direction ) {
  case NORTH:
    ret = chooseObject( plane, ax, ay, northProx, 1, &ox,
    &oy );
    break;
  case SOUTH:
    ret = chooseObject( plane, ax, ay, northProx, -1, &ox,
    &oy );
    break;
  case WEST:
    ret = chooseObject( plane, ax, ay, westProx, 1, &ox, &oy
    );
    break;
  case EAST:
    ret = chooseObject( plane, ax, ay, westProx, -1, &ox,
    &oy );
    break;
}
/* Found an object -- eat it! */
if (ret) {
  int i;
  if (plane == PLANT_PLANE) {
    /* Find the plant in the plant list (based on
    position) */
    for (i = 0 ; i < MAX_PLANTS ; i++) {
      if ((plants[i].location.x == ox) &&
          (plants[i].location.y == oy))
        break;
    }
    /* If found, remove it and grow a new plant elsewhere */
    if (i < MAX_PLANTS) {
      agent->energy += MAX_FOOD_ENERGY;
      if (agent->energy > MAX_ENERGY) {
            agent->energy = MAX_ENERGY;
      }
```

```
            landscape[PLANT_PLANE][oy][ox]--;
            growPlant( i );
          }
        } else if (plane == HERB_PLANE) {
          /* Find the herbivore in the list of agents (based on
           * position).
           */
          for (i = 0 ; i < MAX_AGENTS ; i++) {
            if ( (agents[i].location.x == ox) &&
                 (agents[i].location.y == oy))
              break;
          }
          /* If found, remove the agent from the simulation */
          if (i < MAX_AGENTS) {
            agent->energy += (MAX_FOOD_ENERGY*2);
            if (agent->energy > MAX_ENERGY) {
              agent->energy = MAX_ENERGY;
            }
            killAgent( &agents[i] );
          }
        }
        /* If our agent has reached the energy level to
         * reproduce, allow it to do so (as long as the
         * simulation permits it).
         */
        if (agent->energy > (REPRODUCE_ENERGY * MAX_ENERGY)) {
           if (noRepro == 0) {
            reproduceAgent( agent );
            agentBirths[agent->type]++;
          }
        }
      }
      return;
    }
```

The first step is to identify the plane of interest, which is based on the type of agent doing the consumption. If the agent is an herbivore, we'll look in the plant plane; otherwise, for a carnivore, we'll look in the herbivore plane.

Next, using the agent's direction, we call the `chooseObject` function (shown in Listing 11.16) to return the coordinates of an object of interest in the desired plane. Note that we use the coordinate offset pairs again (as used in Listing 11.11) but concentrate solely on the proximity zone per the agent's direction. If an object is

found, the `chooseObject` function returns a non-zero value and fills the coordinates into the `ox/oy` coordinates as passed in by the `eat` function.

LISTING 11.16 The `chooseObject` Function

```
int chooseObject( int plane, int ax, int ay,
                  const offsetPairType *offsets,
                  int neg, int *ox, int *oy )
{
  int xoff, yoff, i=0;
  /* Work through each of the offset pairs */
  while (offsets[i].x_offset != 9) {
    /* Determine next x,y offset */
    xoff = ax + (offsets[i].x_offset * neg);
    yoff = ay + (offsets[i].y_offset * neg);
    xoff = clip( xoff );
    yoff = clip( yoff );
    /* If an object is found at the check position, return
     * the indices.
     */
    if (landscape[plane][yoff][xoff] != 0) {
      *ox = xoff; *oy = yoff;
      return 1;
    }
    /* Check the next offset */
   i++;
  }
  return 0;
}
```

The `chooseObject` function is similar to the `percept` function shown in Listing 11.12, except that instead of accumulating the objects found in the given plane in the given zone, it simply returns the coordinates of the first object found.

The next step is consuming the object. If an object was returned, we check the plane in which the object was found. For the plant plane, we search through the `plants` array and remove this plant from the `landscape`. We then grow a new plant, which will be placed in a new random location. For the herbivore plane, we identify the particular herbivore with the `agents` array and then kill it using the `killAgent` function (shown in Listing 11.17). The current agent's energy is also increased, per its consumption of the object.

Finally, if the agent has reached the level of energy required for reproduction, the `reproduceAgent` function is called to permit the agent to give birth asexually to a new agent of the given type. The `reproduceAgent` function is shown in Listing 11.18.

Killing an agent is primarily a record-keeping task. We first remove the agent from the landscape and record some statistical data (number of deaths per agent type and number of agents of a given type). Then we store away the agent if it is the oldest found for the given species.

Once record keeping is done, we decide whether we want to initialize a new random agent (of the given type) is its place. The decision point is the number of agents that exists for the given type. Because the evolutionary aspect of the simulation is the most interesting part, we want to maintain a number of open agent slots so that when an agent does desire to reproduce, it can. Therefore, we allow a new random agent to fill the dead agent's place when the population of this agent species fills less than 25 percent of the overall population. This leaves 25 percent of the agent slots (for a given species) open for reproduction.

LISTING 11.17 The `killAgent` Function

```
void killAgent( agentType *agent )
{
  agentDeaths[agent->type]++;
  /* Death came to this agent (or it was eaten)... */
  landscape[agent->type][agent->location.y][agent->location.x]--;
  agentTypeCounts[agent->type]--;

  if (agent->age > bestAgent[agent->type].age) {
    memcpy( (void *)&bestAgent[agent->type],
            (void *)agent, sizeof(agentType) );  }
  /* 50% of the agent spots are reserved for asexual
   * reproduction. If we fall under this, we create a new random agent.
   */
  if (agentTypeCounts[agent->type] < (MAX_AGENTS / 4)) {
    /* Create a new agent */
    initAgent( agent );
  } else {
    agent->location.x = -1;
    agent->location.y = -1;
    agent->type = TYPE_DEAD;
  }
  return;
}
```

The final function within the simulation is the `reproduceAgent` function. This function is by far the most interesting because it provides the Lamarckian learning aspect to the simulation. When an agent gives birth in the simulation, it does so by passing on its traits (its neural network) to its child. The child inherits the parent's

traits with a slight probability of mutating the weights of the neural network. This possible mutation provides the evolutionary aspect of the simulation—a desired increasing level of competence for survival within the environment. The `reproduceAgent` function is provided in Listing 11.18.

LISTING 11.18 The `reproduceAgent` Function

```
void reproduceAgent( agentType *agent )
{
  agentType *child;
  int i;
  /* Don't allow an agent type to occupy more than half of
   * the available agent slots.
   */
  if ( agentTypeCounts[agent->type] < (MAX_AGENTS / 2)) {
    /* Find an empty spot and copy the agent, mutating one of
     * the weights or biases.
     */
    for (i = 0 ; i < MAX_AGENTS ; i++) {
      if (agents[i].type == TYPE_DEAD) break;
    }
    if (i < MAX_AGENTS) {
      child = &agents[i];
      memcpy( (void *)child, (void *)agent, sizeof(agentType)
      );
      findEmptySpot( child );
      if (getSRand() <= 0.2) {
        child->weight_oi[getRand(TOTAL_WEIGHTS)] =
        getWeight();
      }
      child->generation = child->generation + 1;
      child->age = 0;
      if (agentMaxGen[child->type] < child->generation) {
        agentMaxGen[child->type] = child->generation;
      }
      /* Reproducing halves the parent's energy */
      child->energy = agent->energy = (MAX_ENERGY / 2);
      agentTypeCounts[child->type]++;
      agentTypeReproductions[child->type]++;
    }
  }
  return;
}
```

The first step is to identify whether space is available for the new child. For this test, we check to see if less than 50 percent of the total agent slots are filled for the given species. This percentage provides an even distribution of agents in the environment. This percentage may not be biologically correct, but we can simulate one species dominating another in a special playback mode (to be discussed later).

If an open slot is found for the child, we copy the parent's agent structure to the child's and then find an empty spot for the child to occupy. Next, we mutate a single weight within the agent's neural network. Note that we use a `TOTAL_WEIGHTS` symbolic to find the weight to modify. This symbolic encompasses not only the weights but also the biases (because they are contiguous in the agent structure). We then do a little record keeping and halve the energy between the parent and child, which then requires the agents to navigate their environments to find food before they are permitted to reproduce again.

SAMPLE RESULTS

Now we'll look at a few summaries of the simulation in action. The operation of the simulation will also be discussed, including the available command-line options. The simulation can be run by simply executing the application with no options, such as:

```
./sim
```

This command will run the simulation given the parameters defined in the header file `common.h`. The defaults within `common.h` include 36 agents (18 herbivores and 18 carnivores) and 35 plants within a 30×30 grid. The maximum number of simulation steps is set to 1 million. Figure 11.10 shows the maximum age reached for each of the two species.

It's interesting to note the trend of increasing age in the agents. When the carnivore species finds an interesting strategy that increases its longevity, shortly thereafter herbivores find another strategy that gives them the ability to live a little longer. In some ways, the species compete with one another. When a species evolves an interesting strategy, the other species must evolve to counteract this new strategy.

Once a run is complete, the two best agents of the species are saved into a file called `agents.dat`. These agents can then be pitted against one another in a simulation called *playback*. This mode doesn't seed the population with random members but instead starts with the best members from the last run. This mode can be performed using the following command:

```
./sim -prn
```

The p argument defines that we want to run in playback mode. The r specifies that we wish to save the runtime trend information, and n defines that no reproduction is permitted. The trend data stored in *playback* mode includes agent birth and death counts (for both species).

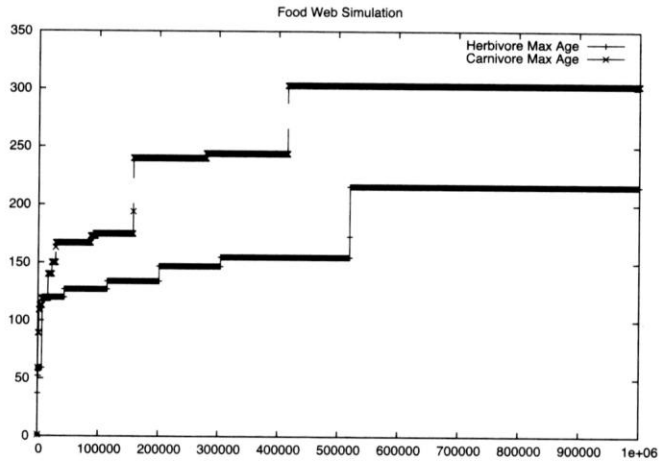


FIGURE 11.10 Age progression in a sample simulation.

Using agents evolved in the prior run, Figure 11.11 shows a plot of the runtime trend data that was created.

Because we've defined that no reproduction can occur, this simulation shows no births, only deaths. When the playback simulation has begun, the landscape is initialized with herbivores and carnivores. It's clear from the plot that the carnivores have a field day with the abundance of herbivores that are available to consume. Once this abundance winds down, carnivore deaths begin a steep increase because the landscape has been stripped of prey and starvation sets in. With the loss of food for the carnivores, their demise is certain.

The simulation provides a number of options, as shown in Table 11.1.

One interesting scenario is provided by the following parameters:

```
./sim       -prncg
```

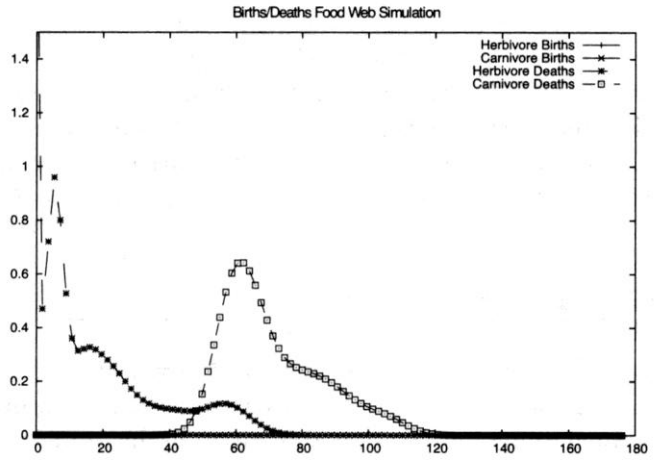


FIGURE 11.11 Runtime trend data from a playback simulation.

TABLE 11.1 Command-Line Options for the Simulation

Option	Description
-h	Command-line help
-p	Playback mode (read agents.dat)
-r	Save runtime trend data (runtime.dat)
-g	Don't regrow plants
-c	Convert carnivores to plants when they die
-n	No reproduction is permitted
-s	Manual step (carriage return required)

This parameter provides a circular food web in which carnivores hunt and eat herbivores, but conversely, once carnivores die, they become food available to herbivores.

INTERESTING STRATEGIES

Though this simulation is simple and agents are provided with a minimal number of input sensors and available actions, very interesting behaviors can result.

One interesting herbivore strategy entailed herding. An herbivore would follow another herbivore in front of it. The analogy could be strength in numbers—as long as you're not the herbivore in front. Carnivores found numerous interesting strategies, one of which was finding plants and then waiting for herbivores to wander by. This strategy was successful but short lived because herbivores finally evolved the ability to avoid carnivores, even if plants were in the vicinity.

ADJUSTING THE PARAMETERS

The size of the environment, the number of agents, and the number of plants in the simulation are all related. For a balanced simulation, the number of plants must be at least equal to the number of herbivores (half the number of agents in the simulation). Any less and herbivores quickly die, followed quickly by the carnivores. The number of agents must not be so large that the species are crowded in the simulation. If the number of agents and grid dimension is similar, the simulation will be balanced.

The simulation parameters are provided in the header file (`common.h`). The simulation can also be adjusted using the command-line parameters, as shown in Table 11.1.

SUMMARY

In this chapter, we investigated artificial life (Alife) by simulating a simple food web. Alife provides a platform for the study of various phenomena in biological and social systems. The greatest strength of Alife in the field of synthetic ethology is the ability to play "what if" games by changing the variables of the simulation and monitoring their effects. We demonstrated the concepts of synthetic ethology using a simple predator/prey simulation, which resulted in the evolution of interesting strategies by both predator and prey.

REFERENCES

[Langton03]. Langton, Chris, "What Is Artificial Life," available online at *www.biota.org/papers/cgalife.html*, accessed January 17, 2003.

[MacLennan03] MacLennan, Bruce, "Artificial Life and Synthetic Ethology," available online at *www.cs.utk.edu/~mclennan/alife.html*, accessed January 17, 2003.

RESOURCES

CALResCo, "The Complexity & Artificial Life Research Concept for Self-Organizing Systems," available online at *www.calresco.org* , accessed January 17, 2003.

Digital Life Lab at Caltech, "Aveda Software," available online at *http://dllab.caltech.edu/avida/*, accessed January 17, 2003.

International Society for Artificial Life, "Home Page," available online at *www.alife.org.*

MacLennan, Bruce, "Bruce MacLennan's Home Page," available online at *www.cs.utk.edu/~mclennan/.*