

Com S 490: Simulation of Food-gathering in Ant Colonies

An Independent Study in Artificial Intelligence

Steve Tangeman
Spring 2004

Supervised by Dimitris Margaritis (dmarg@cs.iastate.edu)
Department of Computer Science
Iowa State University
Ames, IA 50011

Copyright © 2004, Steve Tangeman.

Table of Contents

Introduction	2
Simulation Overview.....	3
The Finite State Machine	4
Class Descriptions	6
The Ant Class	6
The Anthill Class.....	6
The Food Class.....	6
The State Class	6
Modifying the Simulation	8
Future Work	9
Conclusion and Lessons Learned.....	11
References	12
Appendix A: Simulation Controls.....	13
Appendix B: Java Code.....	13
Ant.java:	13
Anthill.java:.....	14
Food.java:	14
State.java:	14
AntSim.java:.....	15

Introduction

The goal of this independent study is to use an AI approach to simulate the collective intelligence of ant colonies in a virtual 3D environment. Collective intelligence is defined as the ability of a group to solve more problems than its individual members [1]. This is also called “Swarm Intelligence” or “emergent behavior” [4]. The term “collective behavior” refers to groups that exhibit this kind of problem-solving intelligence.

Some examples that occur in nature are schools of fish, flocks of birds, and colonies of termites and ants [2]. In each case, each individual member possesses some form of communication that enables them to gather local data from their peers, and then act according to a simple set of rules. This results in a global organization and adaptation which increases the group’s collective intelligence as it grows in number.

The teamwork of ants is a canonical example of collective behavior in simple organisms. Communication is achieved through the release and detection of pheromones. Ants use pheromones for finding food, attracting mates, finding their way back to the anthill, and attacking threats to the colony. They are also able to distinguish between their own pheromones and those from ants of other nests [3].

Finding and transporting food is their most complex problem, but it is one for which they are best suited because it can be broken down into smaller subproblems. If an ant comes across a piece of food that it cannot move by itself, it begins releasing a pheromone to call for assistance. Other ants detect and follow the pheromone trail, each also releasing pheromone, until enough ants have come to move the food back to the anthill. As soon as the food is home, they disperse and search again.

This method of problem solving is becoming more relevant to the world of computing, particularly when the problem can be broken down into smaller subproblems. The food-gathering problem is analogous to other Artificial Intelligence searching problems, such as finding the shortest path in a network [4]. Algorithms like this have already been successfully applied in telecommunications routing.

Simulation Overview

Our 3D simulation environment currently consists of an open field surrounded by mountains, one anthill, and scattered food. The “food” is symbolized by varying sizes of light-gray cubes. The volume of each cube is directly proportional to its weight, which is measured in units of ants (see **Figure 1**).

Each ant can carry 20 times its own bodyweight. The smallest pieces of food are currently 20 units, so it only takes one ant to carry them. The largest piece of food is 100 units, so it takes five ants to carry it. If an ant collides with a piece of food that is already on its way home, it will grab on and help lighten the load.

Ants can smell some kinds of food from a short distance and change course to intercept it, but they can sense pheromones from a much greater distance. To more clearly illustrate which ants are following the pheromones in this simulation, the ants cannot smell the simulated food. They must come upon the food either randomly or by following pheromones.

Ants also have a sense of duration. If they have been releasing pheromone by a piece of food for over a certain amount of time with no response, they give up and move on to search elsewhere [2]. In this simulation, their stagnation period is 30 seconds.

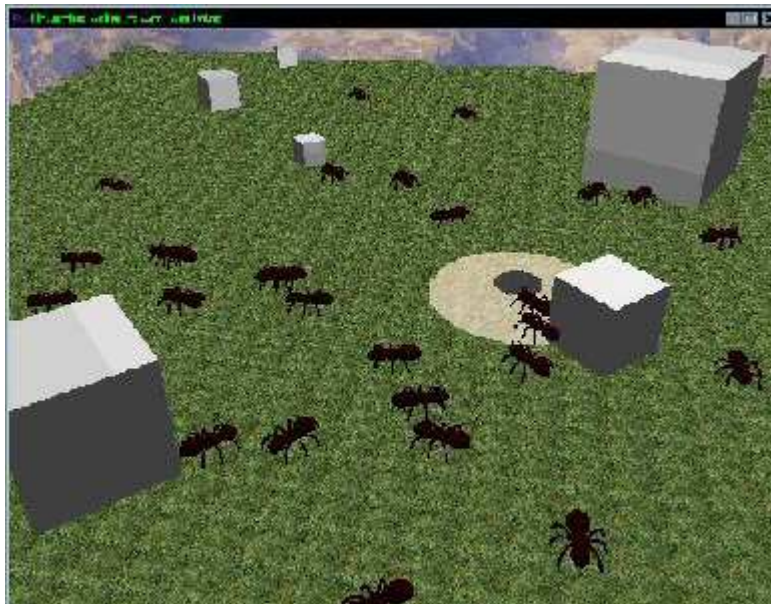


Figure 1

The Finite State Machine

Since the agents' actions are governed by a simple set of rules, it is possible to implement their behavior using a Finite State Machine (FSM). The state transition diagram is illustrated below in **Figure 2**. This FSM has four states:

1. Searching for Food
2. Following Pheromone
3. Budging Food
4. Returning Food Home

The ants are spawned into the searching state, at which point they spread out randomly and look for food. If they collide with a wall or another ant, they turn and continue searching. If they find a piece of food, they attach to it and if it is now movable, they transition to the returning food home state. If the food is still not moving, then they transition to the budging food state and begin releasing their pheromone.

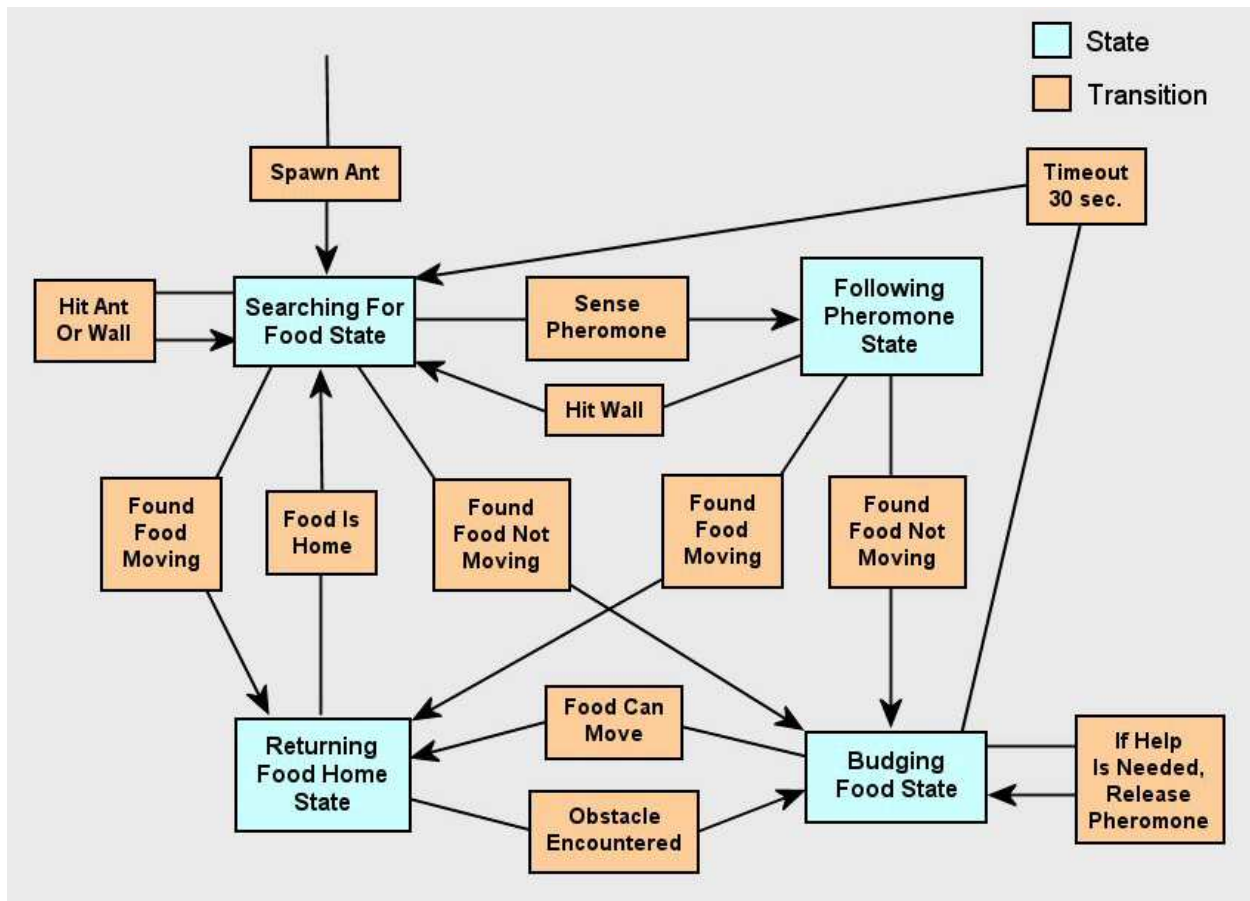


Figure 2

If a pheromone is detected, the ant transitions to the following pheromone state. In the following pheromone state, the ants face the direction of the source of the pheromone, and continue until they hit the food. If the food is moving, they transition to the returning food home state. If the food is not moving, they transition to the budging food state.

Occasionally the food has already moved by the time they get to the source of the pheromone. It is unclear what real ants do in such a situation, so currently they just continue on the same path. If another piece of food is found, they will continue as usual. If a wall is hit, they will transition back to the searching for food state.

While in the budging food state, the ants attached to the same piece of food will wait for 30 seconds for more ants to come and help them. If the time limit is exceeded, they turn away from the food and transition to the searching for food state. The ants constantly check the movability of the food and continue releasing pheromone until there are enough ants attached. When there are, the food starts moving and they transition to the returning food home state. It could be that the food is stuck behind another piece of food, in which case they transition back to the budging food state, but do not release pheromones.

In the returning food home state, the ants will continue pushing/pulling the food until an obstacle is encountered or until it reaches their anthill. As soon as the food is within range of the anthill, it disappears down the hole and the ants transition back to the searching for food state. In some cases, food will jam up at the anthill, but after a while the ants usually clear it and take turns putting the food in. When the field is cleared of food, the ants continue in the searching for food state indefinitely.

For more information on the Finite State Machine, see Appendix B: AntSim.java.

Class Descriptions

All classes but the State class use [WildTangent](#) objects. [WildTangent's](#) "Web Driver" works on top of Microsoft DirectX, an advanced suite of multimedia application programming interfaces (APIs), to provide a set of programming functions, currently implemented in Java, that allow a user to create, display and maintain objects in a 3D environment.

The Ant Class

Each instance of the Ant class consists of a boolean indicating whether or not it is releasing pheromone, an integer that represents its current state in the FSM, a WTActor that points to its [WildTangent](#) body, a Food object that indicates which piece of food the ant is attached to, an Anthill object that indicates which anthill the ant belongs to, and a constructor that takes that Anthill. The FSM function takes an Ant object and determines its behavior based on these properties.

The Anthill Class

Each Anthill object consists of integers for its x and y coordinates, a WTActor that points to its [WildTangent](#) body, and a constructor that takes the x and y coordinates.

The Food Class

Each Food object consists of a boolean indicating whether or not the food is budging, a boolean indicating whether or not the food is down the hole, an integer that contains the number of ants attached to it, a WTGroup that points to its [WildTangent](#) body, an Anthill object that indicates which anthill the food is heading towards, an integer indicating the weight and a constructor that takes that weight.

The State Class

The State Class is used to make it easier to read the code in the FSM. It consists of four integers, each representing a state. The values are: SearchingForFood = 1, FollowingPheromone = 2, BudgingFood = 3, ReturningFoodHome = 4.

The AntSim Class

The AntSim class implements the simulation environment and the Finite State Machine of the ants. It extends the Java Applet, and implements the WTEventCallback interface ([WildTangent's](#) event handler). Arrays are created that will hold Ant, Anthill and Food objects. When the applet is loaded, the first step is setting up the virtual world, including the ground, grass, mountains, light, the color of the sky, and the camera that will be controlled by the user. Next, the Anthills are created and the field is populated with food. The `wt.start()` function sets the world in motion.

The function `createAnthill(int x, int y)` takes the x and y coordinates of where to place the anthill. Similarly, the `createFood(int weight, int x, int y, int hillIndex)` function places food on the field. Its weight will determine its size, and the `hillIndex` variable refers to which anthill the food is/will be heading towards.

The function `spawnAnt(Anthill hill)` is called through user input. Each Ant's `anthill` property refers to the Anthill that is passed as a parameter. This is so the ant knows which anthill to bring the food back to. The new Ant leaves its hole in some random direction. The name of each ant begins with "Ant" and ends with the number of its position in the Ant array. This is used to determine whether an ant is releasing pheromone in the searching for food state. In this simulation, the 'A' key creates up to `MAXANTS` ants from the only anthill. However multiple anthills could be added, each with a key assigned to it that would spawn ants from it.

The `onRenderEvent(WTEvent e)` function is called every time the screen refreshes. It is responsible for all updating all objects in the virtual world. First it moves the camera depending on user input, then moves each piece of food if it is movable and there are no obstacles, then it passes every Ant in the `antColony` array to the FSM function where its behavior is determined. See the code in Appendix B for a further breakdown of the FSM function. See Appendix A for a further description of user input in the simulation.

Modifying the Simulation

To modify the code you must have a Java integrated development environment (IDE), [Eclipse](#) for example, and you must have installed the [Java 2 software developer' s kit \(SDK\)](#) and the [Wild Tangent software developer' s kit \(SDK\)](#) You must also add two JAR files to the build path of your project. In [Eclipse](#), you go to Project Properties – Java Build Path – Libraries – Add JARs... Then add these paths (Note that paths may differ depending on your version):

- C:\j2sdk1.4.2_01\jre\lib\plugin.jar (for netscape.javascript.*)
- C:\Windows\wt\webdriver\wildtangent.jar (for wildtangent.webdriver.*)

The code in AntSim.java has comments indicating which variables for the ants, food and anthills can be easily changed. These include the maximum number of each, the positions of the anthill(s) and food, and the weights of the food. There are also comments for every decision and action of the ants in every state of the FSM function.

When the simulation begins, the camera is oriented facing downward on the field such that the coordinate system is standard, i.e., positive y is up and positive x is right. The default size of the field is 1000 X 1000 and the origin is at the center, so the range of x and y is -500 to 500. When placing either anthills or food onto the field, their coordinates refer to their center point, so an x and y range of -450 to 450 or so should be used to ensure that they will not be embedded into a wall.

Future Work

Many definitions of intelligence have been proposed. One of them is that, given a domain, a system's intelligence can be measured by the number of problems that it can solve in that domain within a finite amount of time and using a limited set of resources. One of my goals in this Independent Study was to create a framework for further development by others who want to experiment with Artificial Intelligence in a 3D environment. The section contains some examples of how my ants could be made more intelligent according to the above definition.

Figure 3 shows a sample of how many minutes it may currently take for 5 to 15 ants to clear the present food scenario. There must be at least 5 in this case because the largest piece of food requires 5 ants to move. The minimal amount of ants take 75 minutes to clear the food, however efficiency increases rapidly with only a few more ants. A more intelligent ant colony should be able to clear the present food scenario in less time and/or with fewer ants.

The experiment that is almost ready to implement is multiple team competitions for food. AntSim.java contains comments indicating where the changes need to be made. In summary, first the anthills must be created, and the key that will be used for spawning the ants from that anthill must be specified in the `onKeyboardEvent(WTEvent e)` function. The most important aspect is how the competition itself will take place. They could try pulling it in different directions and the speed and direction could depend on the number of ants from each colony

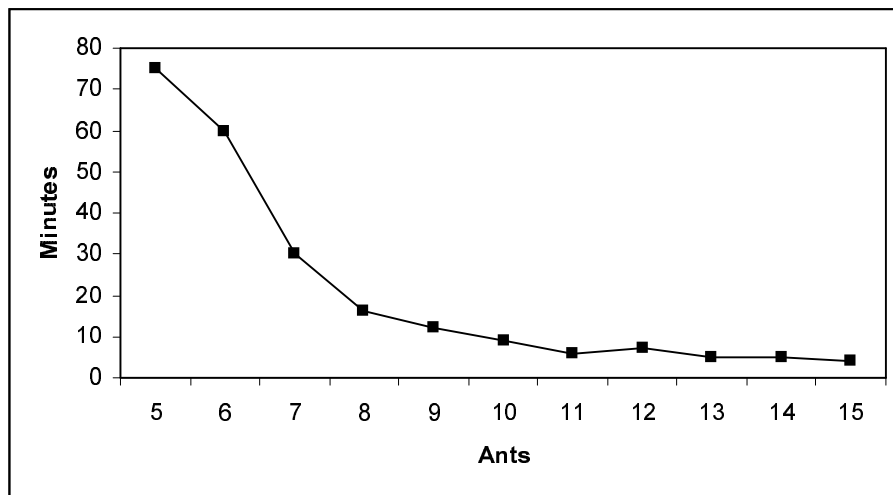


Figure 3

and/or the strength of each ant. Another state could be added, the competing for food state, where the ants could either have a tug-of-war, or actually fight one another. A dead state could also be added if they are given the ability to kill each other.

The competing for food state could also have sub states itself. This implementation is called a hierarchical Finite State Machine. The sub states belong to their own FSM, which could implement any variety of different behaviors. For instance, they could have a pheromone that is recognized as a possible threat to the ant colony. As each ant senses the pheromones it leaves whatever its doing, starts releasing pheromone, and moves in to help fight. With every fighting ant constantly releasing this pheromone, they would call others en route to the battle. This would start a chain reaction that would not stop until it was somehow determined that the threat had been defeated.

Another possible addition is a path-finding algorithm that the ants could use to move food around obstacles. Path finding is one of the most frequently implemented forms of AI in 3D games. Invisible mapnodes are placed on the ground, and the ants follow the paths between them. A straight-line distance heuristic could be used in an A* search algorithm. Mapnodes could be used to emulate behavior more like real ants. In reality, ants leave a pheromones trail back to the anthill. Currently the ants just know where their anthill is, but with mapnodes, they could mark a path that they could follow back later.

This approach could also be used to enable ants return to their anthill and gather a team to help them move the food. Some species of ants return to their anthill while leaving a trail of pheromone back to the food. They then release a different pheromone until they are completely covered by other ants. The lead ant then brings the team with it on the path back to the food.

If a genetic algorithm was used in the spawning of ants, the mating pheromone could also be implemented. Certain traits that are beneficial to the efficiency of the colony could be favored, for example strength or pheromone detection range, until the less favorable traits are bred out.

Other implementation changes could include changing the food cubes to leaves or grasshoppers that hop into the field and die, or having a timeout period in the searching state indicating how long it has been since more food has been found, after which they could return to the anthill and disappear down the hole.

Conclusion and Lessons Learned

During this Independent Study, I have demonstrated how collective behavior allows simple intelligent agents to perform complex tasks by simulating the food-gathering behavior of ant colonies. I have also provided a framework for the implementation of other AI algorithms in a virtual 3D environment.

When I first began my intent was to create something that would be practically useful to AI developers in the gaming industry. As I researched, I found that game AI is applied as a final addition to most commercial games because it is completely dependent on their implementation of the agents and the game environment itself.

I also discovered that deductive reasoning capabilities are not nearly as useful in game AI as behavioral algorithms. The appearance of teamwork is important because it helps create enemies that are more believable and difficult to beat. Finite State Machines are the most common implementation of game agents, so I decided to create one that would emulate collective behavior.

Of course, I still needed a 3D environment in which to create my agents. [WildTangent's](#) 3D engine and web driver was one answer to that problem. I was able to build an environment in Java and import my 3D Studio Max models of the ants and anthills. However, by the time that I had the environment set up and the agents ready to program, the semester was almost over.

I already had the FSM planned out so I was able to implement it rather quickly. Most of the time that I spent coding was related to tweaking the sensors and actuators (movement, collision detection, etc.) of the agent in the virtual environment. I have found that most game AI programmers spend most of their time dealing with similar technical issues.

Creating this simulation has given me the opportunity to see how Finite State Machines can be used to produce realistic collective behavior in intelligent agents. I hope that I have also created a useful framework for others interested in AI to quickly start programming their own agents in a 3D environment.

References

1. Heylighen, Francis. “Collective Intelligence and its Implementation on the Web: algorithms to develop a collective mental map.”
<http://pespmc1.vub.ac.be/Papers/CollectiveWebIntelligence.pdf>
2. Cornett, Kyle. “Modeling Collective Behavior.”
<http://www.stetson.edu/departments/mathcs/students/research/cs/cs498/2000/kylecornett.pdf>
3. “Argentine Ants.” Insecta Inspecta World. <http://www.insecta-inspecta.com/ants/argentine/>
4. Gordon, David. “Collective Intelligence in Social Insects.”
<http://ai-depot.com/Essay/SocialInsects.html>

Appendix A: Simulation Controls

A Button	Spawn an ant
Up Button	Move Forward
Down Button	Move Backward
Left Button	Strafe Left
Right Button	Strafe Right
Left Click and Drag Up	Rotate camera up
Left Click and Drag Down	Rotate camera down
Left Click and Drag Left	Rotate camera left
Left Click and Drag Right	Rotate camera right

Appendix B: Java Code

Ant.java:

```
import wildtangent.webdriver.*;

public class Ant
{
    public boolean pheromone;
    public int state;
    public WTActor body;
    public Food attached;
    public Anthill anthill;

    public Ant(Anthill hill)
    {
        pheromone = false;
        state = 1;
        body = null;
        attached = null;
        anthill = hill;
    }
}
```

Anthill.java:

```
import wildtangent.webdriver.*;

public class Anthill
{
    public int xPos;
    public int yPos;
    public WTActor body;

    public Anthill(int x, int y)
    {
        xPos = x;
        yPos = y;
        body = null;
    }
}
```

Food.java:

```
import wildtangent.webdriver.*;

import wildtangent.webdriver.*;

public class Food
{
    public boolean budgeted;
    public boolean downhole;
    public int attachedAnts;
    public WTGroup body;
    public Anthill anthill;
    public int weight;
    public long timeStamp;

    public Food(int setWeight)
    {
        budgeted = false;
        downhole = false;
        attachedAnts = 0;
        body = null;
        anthill = null;
        weight = setWeight;
        timeStamp = System.currentTimeMillis();
    }
}
```

State.java:

```
public class State
{
    public static final int SearchingForFood = 1;
    public static final int FollowingPheromone = 2;
    public static final int BudgetingFood = 3;
    public static final int ReturningFoodHome = 4;
}
```

AntSim.java:

```
import java.applet.*;
import wildtangent.webdriver.*;
import netscape.javascript.*;

/**
 * Com S 490: Simulation of<br>
 * Food-gathering in Ant Colonies<br>
 * Steve Tangeman <br>
 * Spring 2004
 */
public class AntSim extends Applet implements WTEventCallback
{
    /** The WT object that is located on the HTML page. */
    WT wt;

    /** The camera object that serves to show the scene. */
    WTCamera camera;

    /** The stage object that's used to add/create objects in the scene. */
    WTStage stage;

    /** The light that's used to light the scene. */
    WTLight light;

    /** The hosting HTML page that hosts this applet. */
    JSObject webPage;

    // Ant Colony variables
    /** Change this value to increase the <br>
    ** number of possible ants */
    public static int MAXANTS = 40;
    int antCount = 0;
    Ant[] antColony = new Ant[MAXANTS];

    // Anthill variables
    /** Change this value to increase the <br>
    ** number of possible anthills */
    public static int MAXANTHILLS = 10;
    int anthillCount = 0;
    Anthill[] allHills = new Anthill[MAXANTHILLS];

    // Food Supply variables
    /** Change this value to increase the <br>
    ** number of possible pieces of food */
    public static int MAXFOOD = 40;
    int foodCount = 0;
    Food[] foodSupply = new Food[MAXFOOD];

    // WT variables
    int MOUSEDOWN = 0;
    int move = 0;
    int strafe = 0;
    int dt = 0;
    WTVector3D v;
```

```

/**
 * This function creates the simulation <br>
 * environment and sets it in motion
 * @param object
 */
public void load( Object object )
{
    // Try getting the webpage
    try
    {
        // Get the main window
        webPage = JSObject.getWindow( this );
    }
    catch ( Exception e )
    {
        System.out.println( "Error occurred while trying to acquire reference
                             to HTML web page: " + e );
    }

    // Create the virtual world
    try
    {
        // Get the WT object
        wt = wt3dLib.getWT( object );
        wt.designedForVersion( "2.0" );
        stage = wt.createStage();

        // Make the sky blue
        stage.setBGColor( 98, 227, 255 );

        // Create the ground plane
        WTModel ground = wt.createPlane( 1000, 1000 );
        ground.setTextureRect( "front", 0, 0, 50, 50 );
        WTGroup groundContainer = wt.createGroup();
        groundContainer.attach( ground );
        stage.addObject( groundContainer );

        // Add grass to the ground plane
        WTBitmap groundBitmap = wt.createBitmap( "images/grass.jpg" );
        ground.setTexture( groundBitmap );

        // Create bitmap for mountains
        WTBitmap mtnBitmap = wt.createBitmap( "images/mtn.png" );
        mtnBitmap.setColorKey( 255, 0, 0 ); // make red transparent

        // Create mountain model
        WTModel mtn = wt.createPlane( 1000, 250 );
        mtn.setTexture( mtnBitmap );

        // Create mountain containers and add them to the stage
        WTGroup northCont = wt.createGroup();
        northCont.setName( "Wall" );
        WTGroup southCont = wt.createGroup();
        southCont.setName( "Wall" );
        WTGroup eastCont = wt.createGroup();
        eastCont.setName( "Wall" );
        WTGroup westCont = wt.createGroup();

```

```

westCont.setName( "Wall" );

// Attach mountain models
northCont.attach( mtn );
southCont.attach( mtn );
eastCont.attach( mtn );
westCont.attach( mtn );

// Position and orient containers
northCont.setAbsolutePosition( 0, 500, 30 );
northCont.setAbsoluteOrientationVector( 0, -1, 0, 0, 0, 1 );
southCont.setAbsolutePosition( 0, -500, 30 );
southCont.setAbsoluteOrientationVector( 0, 1, 0, 0, 0, 1 );
eastCont.setAbsolutePosition( -500, 0, 30 );
eastCont.setAbsoluteOrientationVector( 1, 0, 0, 0, 0, 1 );
westCont.setAbsolutePosition( 500, 0, 30 );
westCont.setAbsoluteOrientationVector( -1, 0, 0, 0, 0, 1 );

// Add containers to stage
stage.addObject( northCont );
stage.addObject( southCont );
stage.addObject( eastCont );
stage.addObject( westCont );

// Create point light
WTLight light = wt.createLight( 1 );           // point light
stage.addObject( light );
light.setAbsolutePosition( 0, 0, 500 );

// Create ambient light
WTLight light2 = wt.createLight( 0 );         // ambient light
stage.addObject( light2 );
light2.setColor( 80, 80, 80 );

// Create the camera
camera = stage.createCamera();
camera.setAbsolutePosition( 0, 0, 1500 );
camera.setAbsoluteOrientationVector( 0, 0, -1, 1, 0, 0 );
stage.addObject( camera );

// Initialize the event handlers
wt.setNotifyMouseEvent( 1 );
wt.setNotifyRenderEvent( true );
wt.setNotifyKeyboardEvent( true );
wt.setOnMouseEvent( this );
wt.setOnRenderEvent( this );
wt.setOnKeyboardEvent( this );

// Create the Ant Hills
// In this case there is only one anthill
createAnthill( 80, 130 );

// Populate the field with food
// The last number must be the index of the
// Anthill that the food will face towards.
// If you want competing ants to take food
// from eachother, the orientation of the food

```

```

// must be changed to face towards the new ant's
// hill when it attaches to the food in the FSM.
// Each food's anthill property must also be changed.
createFood( 40, 140, 160, 0 );
createFood( 40, -210, 390, 0 );
createFood( 40, -350, -10, 0 );
createFood( 40, 2, -100, 0 );
createFood( 30, 250, 250, 0 );
createFood( 20, -200, -200, 0 );
createFood( 20, 130, -390, 0 );
createFood( 20, -280, -450, 0 );
createFood( 20, 450, -150, 0 );
createFood( 50, 3, 200, 0 );
createFood( 50, 300, 10, 0 );
createFood( 50, -300, 100, 0 );
createFood( 30, -300, -300, 0 );
createFood( 30, -250, 250, 0 );
createFood( 60, -175, 175, 0 );
createFood( 20, 275, -375, 0 );
createFood( 90, 250, -200, 0 );
createFood( 20, 300, 120, 0 );
createFood( 20, 80, -200, 0 );
createFood( 20, -400, -220, 0 );
createFood( 20, 350, -400, 0 );
createFood( 100, 400, 400, 0 );
createFood( 20, 50, 350, 0 );
createFood( 20, 200, 350, 0 );
createFood( 20, -400, 400, 0 );
createFood( 20, -100, -400, 0 );
createFood( 30, -200, -100, 0 );
createFood( 50, -10, -250, 0 );
createFood( 20, -150, 50, 0 );
createFood( 20, 150, 40, 0 );
createFood( 20, 120, -100, 0 );
createFood( 20, -50, 80, 0 );

// Start the wt rendering
wt.start();

// Give the scene focus
wt.focus();
}
catch ( NullPointerException error )
{
    error.printStackTrace();

    try
    {
        // Failed to recognize the Web Driver, open the Download URL:
        webPage.eval( "open( DOWNLOAD_URL );" );
    }
    catch ( Exception e )
    {
    }
}
}

```

```

/**
 * This function creates an anthill at the given<br>
 * x and y coordinates from the camera's perspective.
 * @param x
 * @param y
 */
public void createAnthill( int x, int y )
{
    if (anthillCount < MAXANTHILLS)
    {
        Anthill thisHill = new Anthill( y, x );
        WTActor hill = wt.createActor("actors/AntHill.wsad");
        hill.setAbsolutePosition( y, x, 0 );
        hill.setAbsoluteOrientationVector( 0, 1, 0, 0, 0, 1 );
        hill.setCollisionMask( 0 );
        thisHill.body = hill;
        stage.addObject( hill );
        allHills[anthillCount] = thisHill;
        anthillCount++;
    }
}

/**
 * This function creates a piece of food with the given weight,<br>
 * and places it at the given x and y coordinates. It is oriented<br>
 * to face the anthill at the given hillIndex in the allHills array.
 * @param int weight
 * @param int x
 * @param int y
 * @param int hillIndex
 */
public void createFood( int weight, int x, int y, int hillIndex )
{
    if (foodCount < MAXFOOD)
    {
        WTGroup group = wt.createGroup();
        group.setAbsolutePosition( y, x, weight / 2 );
        stage.addObject(group);
        WTModel model = wt.createBox(weight, weight, weight);
        float half = (float)(weight / 2 + 5);
        group.setCollisionBox( -half, half, -half, half, -half, half);
        group.setName(String.valueOf(foodCount));
        group.attach(model);
        group.setAbsoluteOrientationVector( allHills[hillIndex].xPos - y,
            allHills[hillIndex].yPos - x, 0, 0, 0, 1);
        model.setColor(180, 180, 180);
        Food thisFood = new Food(weight);
        thisFood.body = group;
        thisFood.anthill = allHills[hillIndex];
        foodSupply[foodCount] = thisFood;
        foodCount++;
    }
}

```

```

/**
 * This function spawns an ant out of the given anthill
 */
public void spawnAnt( Anthill hill )
{
    WTActor actor = wt.createActor("actors/Ant.wsad");
    float x = (float)(-230 + Math.random() * 460); // between -230 and 230
    float y = (float)(-230 + Math.random() * 460); // between -230 and 230
    actor.setAbsolutePosition( hill.xPos, hill.yPos, 5 );
    actor.setAbsoluteOrientationVector( x, y, 0, 0, 0, 1 );
    float flt = (float)0.5;
    actor.setScale( flt, flt, flt );
    actor.setName("Ant" + String.valueOf(antCount));
    actor.setCollisionBox( -10, 10, -10, 10, 3, 10);
    stage.addObject( actor );
    Ant thisAnt = new Ant( hill );
    thisAnt.body = actor;
    thisAnt.anthill = hill;
    antColony[antCount] = thisAnt;
    antCount++;
}

/**
 * Implements the WTEventCallback interface
 */
public void onRenderEvent( WTEvent e )
{
    // put mouse back in center if mouse is down
    if( MOUSEDOWN == 1 )
        wt.setMousePosition( wt.getWidth()/2, wt.getHeight()/2 );

    // get time interval, make sure it's a reasonable size
    dt = e.getInterval();
    if( dt > 100 ) dt = 100;

    // reposition the camera for strafing
    camera.moveBy((float)(strafe * dt * 0.2), 0, 0);

    // get camera's absolute fwd vector
    v = camera.getAbsoluteOrientationVector();

    // get camera's absolute position
    WTVector3D pos = camera.getAbsolutePosition();

    // get magnitude of camera's fwd vector
    // projected on the ground plane
    float xyzMag = (float)(Math.sqrt( Math.pow( v.getX(), 2 ) +
        Math.pow( v.getY(), 2 ) + Math.pow( v.getZ(), 2 )));

    // normalize the x and y components to this magnitude
    float vX = v.getX() / xyzMag;
    float vY = v.getY() / xyzMag;
    float vZ = v.getZ() / xyzMag;

    // calculate camera's potential new position
    float newX = (float)(pos.getX() + move * vX * dt * .2);
    float newY = (float)(pos.getY() + move * vY * dt * .2);

```

```

float newZ = (float)(pos.getZ() + move * vZ * dt * .2);

// if it's outside walls (with some padding), then set it at the walls
if( newX > 490 ) newX = 490;
if( newX < -490 ) newX = -490;
if( newY > 490 ) newY = 490;
if( newY < -490 ) newY = -490;
if( newZ < 10 ) newZ = 10;

// set the camera's new position
camera.setAbsolutePosition( newX, newY, newZ );

// Update position of food
for( int j = 0 ; j < foodCount ; j ++ )
{
    if(foodSupply[j].weight <= (foodSupply[j].attachedAnts * 20))
        foodSupply[j].budged = true;

    // Move the food if it is budging
    if(foodSupply[j].budged)
    {
        WTVector3D vec = foodSupply[j].body.getAbsolutePosition();
        float thisX = Math.abs(vec.getX() - foodSupply[j].anthill.xPos);
        float thisY = Math.abs(vec.getY() - foodSupply[j].anthill.yPos);
        if (thisX < 40 && thisY < 40)
        {
            foodSupply[j].budged = false;
            foodSupply[j].downhole = true;
            stage.removeObject(foodSupply[j].body);
        }
        else
        {
            WTCollisionInfo far = foodSupply[j].body.checkCollision(
                0, 0, (float)(.4 * dt), true, 1 );
            if (far != null)
            {
                WXObject farObject = far.getHitObject();
                String farName = farObject.getName();
                if (isNumber(farName))
                    foodSupply[j].budged = false;
                else
                    foodSupply[j].body.moveBy(
                        0, 0, (float).07 * dt);
            }
            else
                foodSupply[j].body.moveBy( 0, 0, (float).07 * dt);
        }
    }
}

// Move the ants according to the Finite State Machine
for( int i = 0 ; i < antCount ; i ++ ) {
    FSM(antColony[i]);
}
}

```

```

/**
 * FINITE STATE MACHINE: <BR>
 * This function takes an ant and determines its behavior <br>
 * based on its current state, and the conditions of the <br>
 * transitions to other states.
 * @param Ant thisAnt
 */
public void FSM(Ant thisAnt)
{
    // Get collision information for the ant
    WTCollisionInfo coll = thisAnt.body.checkCollision( 0, 0, (float)(.5 *
        dt), true, 1 );

    switch(thisAnt.state)
    {
        // SEARCHING FOR FOOD STATE
        case State.SearchingForFood:

            // If the ant will hit something
            if( coll != null)
            {
                // Find out what was hit
                WLObject hitObject = coll.getHitObject();
                String thisName = hitObject.getName();

                // If this ant will hit food
                if (isNumber(thisName))
                {
                    // THIS IS ONE OF THE TWO PLACES THAT YOU NEED TO CHANGE THE
                    // ORIENTATION OF THE FOOD'S "body" AND THE "anthill" PROPERTY
                    // OF THE FOOD IF YOU WANT TO HAVE COMPETING ANT COLONIES.
                    // THE OTHER PLACE IS IN THE FOLLOWING PHEROMONE STATE.

                    // Find out which piece of food and attach ant to it
                    int thisNum = Integer.valueOf(thisName, 10).intValue();
                    thisAnt.attached = foodSupply[thisNum];
                    thisAnt.attached.attachedAnts++;
                    thisAnt.attached.timeStamp = System.currentTimeMillis();

                    // Position the ant next to the food
                    WTVector3D v3D =
                        thisAnt.attached.body.getAbsoluteOrientationVector();
                    thisAnt.body.moveBy( 0, 0, (float).3 * dt);
                    thisAnt.body.setAbsoluteOrientationVector(v3D.getX(), v3D.getY(),
                        0, 0, 0, 1);

                    // Check the weight to ant ratio for movability
                    if(thisAnt.attached.weight <= (thisAnt.attached.attachedAnts *20))
                        thisAnt.attached.budged = true;

                    // State Transitions depending on movability
                    if(thisAnt.attached.budged)
                        thisAnt.state = State.ReturningFoodHome;
                    else
                        thisAnt.state = State.BudgingFood;
                }
            }
    }
}

```

```

// If this ant will hit another ant
else if (thisName.substring(0, 3).equalsIgnoreCase("Ant"))
{
    // Find out which ant was hit
    int thisNum = Integer.valueOf(thisName.substring(3)).intValue();
    Ant hitAnt = antColony[thisNum];

    // IF YOU HAVE COMPETING ANT COLONIES YOU WILL NEED TO
    // DISTINGUISH WHICH COLONY THE PHEROMONE IS COMING FROM

    // If the ant is releasing pheromone
    if (hitAnt.pheromone)
    {
        // Turn towards the position of the food and continue
        WtVector3D foodVec = hitAnt.attached.body.getAbsolutePosition();
        WtVector3D antVec = thisAnt.body.getAbsolutePosition();
        thisAnt.body.setAbsoluteOrientationVector(foodVec.getX() -
            antVec.getX(),
            foodVec.getY() - antVec.getY(), 0, 0, 0, 1);
        thisAnt.body.moveBy( 0, 0, (float).07 * dt);

        // State Transition
        thisAnt.state = State.FollowingPheromone;
    }
    else
    {
        // Turn between 30 and 90 degrees left or right randomly
        float flt = (float)(Math.random());
        float ang = (float)(30 + flt * 60);
        if (flt > 0.5)
            thisAnt.body.setRotation( 0, 0, 1, ang, 1 );
        else
            thisAnt.body.setRotation( 0, 0, 1, ang * -1, 1 );
    }
}

// If the ant will hit a wall
else
{
    // Turn between 30 and 90 degrees left or right randomly
    float flt = (float)(Math.random());
    float ang = (float)(30 + flt * 60);
    if (flt > 0.5)
        thisAnt.body.setRotation( 0, 0, 1, ang, 1 );
    else
        thisAnt.body.setRotation( 0, 0, 1, ang * -1, 1 );
}
}
else
{
    // Move forward and keep searching
    thisAnt.body.moveBy( 0, 0, (float).07 * dt);
};
break;

```

```

// FOLLOWING PHEROMONE STATE
case State.FollowingPheromone:

    // If this ant will hit something
    if( coll != null)
    {
        // Find out what was hit
        WLObject hitObject = coll.getHitObject();
        String thisName = hitObject.getName();

        // If this ant will hit food
        if (isNumber(thisName))
        {
            // THIS IS ONE OF THE TWO PLACES THAT YOU NEED TO CHANGE THE
            // ORIENTATION OF THE FOOD'S "body" AND THE "anthill" PROPERTY
            // OF THE FOOD IF YOU WANT TO HAVE COMPETING ANT COLONIES.
            // THE OTHER PLACE IS IN THE SEARCHING STATE.

            // Find out which piece of food and attach ant to it
            int thisNum = Integer.valueOf(thisName, 10).intValue();
            thisAnt.attached = foodSupply[thisNum];
            thisAnt.attached.attachedAnts++;
            thisAnt.attached.timeStamp = System.currentTimeMillis();

            // Position the ant next to the food
            WTVector3D v3D =
                thisAnt.attached.body.getAbsoluteOrientationVector();
            thisAnt.body.moveBy( 0, 0, (float).3 * dt);
            thisAnt.body.setAbsoluteOrientationVector(v3D.getX(), v3D.getY(),
                0, 0, 0, 1);

            // Check the weight to ant ratio for movability
            if(thisAnt.attached.weight <= (thisAnt.attached.attachedAnts*20))
                thisAnt.attached.budged = true;

            // State Transitions depending on movability of food
            if(thisAnt.attached.budged)
                thisAnt.state = State.ReturningFoodHome;
            else
                thisAnt.state = State.BudgingFood;
        }

        // If the food that the ant is following is not hit for
        // some reason, stop at the wall and start searching again
        else if (thisName.equalsIgnoreCase("Wall"))
        {
            // State Transition
            thisAnt.state = State.SearchingForFood;
        }
        else
        {
            // Continue forward
            thisAnt.body.moveBy( 0, 0, (float).07 * dt);
        }
    }
else
{

```

```

    // Continue forward
    thisAnt.body.moveBy( 0, 0, (float).07 * dt);
}
break;

// BUDGING FOOD STATE
case State.BudgingFood:

    // If this ant's food is movable
    if(thisAnt.attached.budged)
    {
        // Stop releasing Pheromone
        thisAnt.pheromone = false;

        // Retract detection range of ant
        thisAnt.body.setCollisionBox( -10, 10, -10, 10, 3, 10 );
        thisAnt.body.moveBy( 0, 0, (float).07 * dt);

        // State Transition
        thisAnt.state = State.ReturningFoodHome;
    }
    else
    {
        // Check to see how long this ant has been
        // waiting for its food to start moving
        long wait = System.currentTimeMillis() -
            thisAnt.attached.timeStamp;

        // If this ant has been waiting for over 30 seconds
        if (wait > 30000)
        {
            // Stop releasing Pheromone
            thisAnt.pheromone = false;

            // Retract detection range of ant
            thisAnt.body.setCollisionBox( -10, 10, -10, 10, 3, 10 );

            // Release and turn away from the food and start searching
            WtVector3D foodVec =
                thisAnt.attached.body.getAbsolutePosition();
            WtVector3D antVec = thisAnt.body.getAbsolutePosition();
            thisAnt.body.setAbsoluteOrientationVector(antVec.getX() -
                foodVec.getX(), antVec.getY() - foodVec.getY(),
                0, 0, 0, 1);
            thisAnt.attached.attachedAnts--;

            // State Transition
            thisAnt.state = State.SearchingForFood;
        }

        // If the ant has enough help but can't move because of obstacle
        else if(thisAnt.attached.weight <= thisAnt.attached.attachedAnts
            * 20)
        {
            // Stop releasing Pheromone

```

```

        thisAnt.pheromone = false;

        // Retract detection range of ant
        thisAnt.body.setCollisionBox( -10, 10, -10, 10, 3, 10 );
    }

    // Otherwise the ant still needs help moving the food
    else
    {
        // Release Pheromone to call for help
        thisAnt.pheromone = true;

        // Extend detection range of ant
        thisAnt.body.setCollisionBox( -60, 60, -60, 60, -70, 70 );
    }
};
break;

// RETURNING FOOD HOME STATE
case State.ReturningFoodHome:

// If the food has been returned home
if(thisAnt.attached.downhole)
{
    // Turn between 30 and 90 degrees left or right randomly
    float flt = (float)(Math.random());
    float ang = (float)(30 + flt * 60);
    if (flt > 0.5)
        thisAnt.body.setRotation( 0, 0, 1, ang, 1 );
    else
        thisAnt.body.setRotation( 0, 0, 1, ang * -1, 1 );

    // State Transition
    thisAnt.state = State.SearchingForFood;
}

// If there is an obstacle in the way of the food
else if (!thisAnt.attached.budged)
{
    // State Transition
    thisAnt.state = State.BudgingFood;
}
else
{
    // Continue forward
    thisAnt.body.moveBy( 0, 0, (float).07 * dt);
};
break;
default:
break;
}
}

/**
 * This function returns true if the <br>

```

```

* String contains only numbers.
* @param String s
* @return boolean
*/
public static boolean isNumber(String s) {
    try {
        if (s == null) return false;
        Double.valueOf(s).doubleValue();
        return true;
    }
    catch (NumberFormatException e) {
        return false;
    }
}

/**
 * Implements the WTEventCallback interface
 */
public void onMouseEvent( WTEvent e )
{
    // If left button down
    if( e.getType() == 6 )
    {
        wt.setCursorState( 0 );           // cursor invisible
        MOUSEDOWN = 1;                   // set state
    }

    // If left button down AND mouse moved
    if( MOUSEDOWN == 1 && e.getType() == 2 )
    {
        // get the mouse movement in x
        float turn = (float)(0.05 * (e.getX() - wt.getWidth()/2));
        // get the mouse movement in z
        float pitch = (float)(0.05 * (e.getY() - wt.getHeight()/2));
        // rotate the camera
        if( Math.abs( turn ) >= 1 ) camera.setRotation( 0, 0, 1, turn, 1 );
        if( pitch >= 1 )
            camera.setRotation( 1, 0, 0, pitch, 0 );
        if( pitch <= -1 && v.getZ() < .7 )
            camera.setRotation( 1, 0, 0, pitch, 0 );
    }

    // If left button up
    if( e.getType() == 7 )
    {
        wt.setCursorState( 1 );           // cursor visible
        MOUSEDOWN = 0;                   // set state
    }
}

/**
 * Implements the WTEventCallback interface
 */
public void onKeyboardEvent( WTEvent e )
{
    int key = e.getKey();

```

```

int keyState = e.getKeyState();
if (keyState == 1)
{
    switch (key)
    {
        case 38:          // up key
            move = 1;      // move forward
            break;
        case 40:          // down key
            move = -1;     // move backward
            break;
        case 37:          // left key
            strafe = -1;   // strafe left
            break;
        case 39:          // right key
            strafe = 1;    // strafe right
            break;

        // This is where you must distinguish which
        // keys produce ants from which ant colonies
        // keynumbers can be found at
http://www.wildtangent.com/developer/help/function.asp?fid=72&versionmask=-1
        case 65:          // 'A' key
            if (antCount < MAXANTS)
                spawnAnt( allHills[0] );
            break;

        default:
            break;
    }
}

// If no keys pressed
if (keyState == 0)
{
    move = 0; // don't move
    strafe = 0; // don't strafe
}
}

/**
 * Implements the WTEventCallback interface
 */
public void onExceptionEvent( WTEvent e )
{
}

/**
 * Unloads all WT assets and performs any necessary destroying and unloading
 * executions. This <B>must</B> be called from the HTML page's
<i>onunload</i>
 * event or else the WT object does not get released properly from memory.
 */
public void unload()
{
    // Release references to WTObjects from memory.
    // This is necessary to successfully release the WT object from memory:

```

```
camera = null;
stage = null;
light = null;

// Tell the WT object that this object is done with it:
wt.shutdown();
wt = null;
System.out.println( "AntSim.unload() called successfully." );
}
}
```