WOUTER BULTEN, ROBERT-JAN DRENTH & FRANK DORSSERS

WORKSHOP ARTIFICIAL LIFE

SYMPOSIUM COMMITTEE COGNAC, ACAIS 2013

PUBLISHED BY SYMPOSIUM COMMITTEE COGNAC, ACAIS 2013

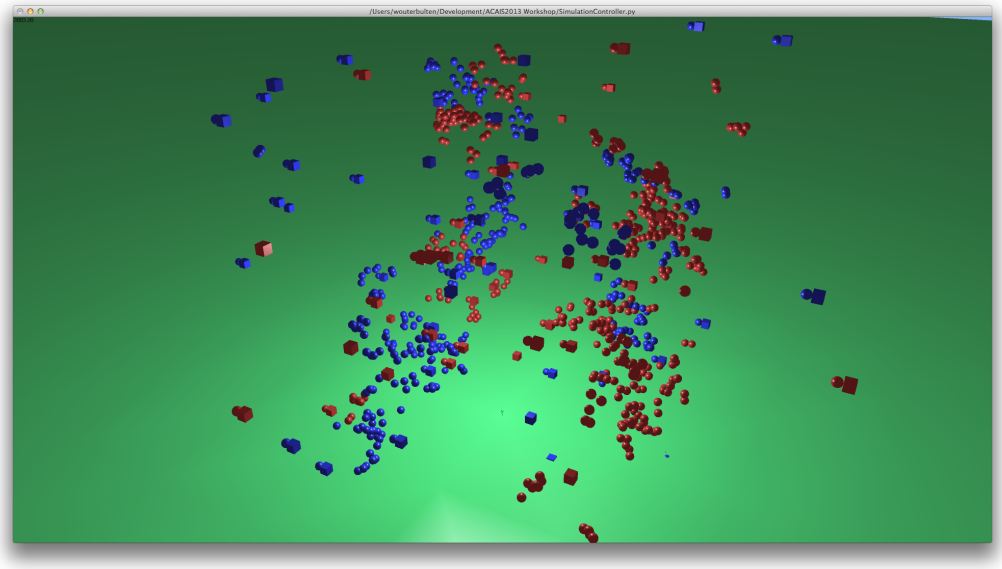
First printing, June 2013

Contents

Introduction

During ACAIS 2013 a workshop will be hosted which can be joined by all attendees who can bring a laptop (working together is of course allowed). The workshop will focus on becoming familiar with the 3D multi-agent simulation environment Breve. You will learn the basics of creating agents and letting them move in an artificial environment. After this workshop you will be able to create your own simulations using Breve.

This manual will guide you in creating your first simulation. You will create a environment were agents have to collect food. Your simulation is finished when all food sources are stacked together. There is one limitation, agents may not communicate with each other. If you have time left you can extend the simulation by introducing two competing groups. Finally it will look something like this:



We hope you will like this workshop and get a good grasp of creating multi-agent and artificial life simulations.

Wouter Bulten, Frank Dorssers, Robert-Jan Drenth

Setting up Breve

The program essential to this workshop is Breve. In this guide we will be guiding you through all necessary steps to get it up and running.

Windows

In this section we will explain how to set up Breve under Windows. Extract the files to a location of your choosing. For the remainder of the instruction we will assume you have placed Breve in the C: directory.

Previous Python installation

If you have previously installed Python and added this to the path, you will have to remove the path. You can leave Python installed. If you do not remove the path, Breve will try to use the wrong Python version. Where you can find these paths can be seen in the following section.

Setting paths

You will have to set two paths. One for Breve and one for the Python version that comes with Breve.

There are two possibilities here. The first one is temporary and has to be run in command prompt every time you start a new command prompt session. It consists of the following two lines:

```
set BREVE_CLASS_PATH=<breve_path>\lib\classes
set PYTHONPATH=\%PYTHONPATH\%;<breve_path>\lib\python2.3
```

If, for example, you have copied Breve to 'C:
breve_2.7.2' it would look like this:

```
set BREVE_CLASS_PATH=C:\breve_2.7.2\lib\classes
set PYTHONPATH=\%PYTHONPATH\%;C:\breve_2.7.2\lib\python2.3
```

There is an alternative option which is more permanent, but can easily be undone once it is not required anymore. This is done by changing the environment veriables in the Advanced System Settings:

```
Windows + r > Enter "control sysdm.cpl" > Run the command
> Advanced > Environment Variables
```

Here you will see two areas, one are the user variables and the other are the system variables. We will be adding two variables to the last one. Simply press "New..." and enter the following information, substituting <breve_path> with the actual path on your pc:

```
Variable name: BREVE_CLASS_PATH
Variable value: <breve_path>\lib\classes
```

```
Variable name: PYTHONPATH
Variable value: <breve_path>\lib\python2.3
```

Testing

Breve comes with several demos. You can use these to check whether the installation went okay.

You can run breve by running the breve executable followed by a project, don't forget to set the paths if you are using the temporary version.

The following command prompt command will run the DLA demo:

```
<breve_path>\bin\breve <breve_path>\demos\DLA.py
```

Mac OS X (/ Linux)

In this section we will explain how to set up Breve under Mac OS X. Extract the files to a location of your choosing. For the remainder of the instruction we will assume you have placed Breve in your Applications directory.

Setting path

On Mac OS X you only have to set one path. Just like with Windows you have two options, one is only temporary while the second is sort of permanent.

First of all the temporary version. Run the following line each time you start a new terminal session in which you will be using Breve:

```
export BREVE_CLASS_PATH=<breve_path>/lib/classes
```

For the second, more permament version, you will have to change/create a .bash_profile file. Anything in this file will be run each time you start a terminal session. Run the following lines in a terminal session to create or change the file:

```
nano ~/.bash_profile
```

Now you have either created a .bash_profile file or you are editing the existing version. Enter the following line:

```
export BREVE_CLASS_PATH=<breve_path>/lib/classes
```

Now you can save this file by pressing 'ctrl+o' and you can leave this file by pressing 'ctrl+x'. After restarting the terminal you should be able to use Breve.

Testing

Breve comes with several demos. You can use these to check whether the installation went okay.

You can run breve by running the breve executable followed by a project, don't forget to export the path if you are using the temporary version.

The following terminal command will run the DLA demo:

```
<breve_path>/bin/breve <breve_path>/demos/DLA.py
```

Step 1: Creating a simulation

Every simulation is based around a *Controller* which controls the simulation. To make such a controller we extend the *Breve.Control* class and implement two methods: the constructor and an iterate method.

```
import breve

class SimulationController ( breve.Control ):

        def __init__( self ):
                breve.Control.__init__(self)

                print "Simulation Started"

        def iterate( self ):

                breve.Control.iterate( self )

# Start the simulation
SimulationController()
```

> **Save the code above in a file called SimulationController.py.**

The *__init__* function is called when the simulation starts and can be used to initialise other components (such as agents).

The *iterate* function is called on every iteration in the simulation. It is important to call the parent iterate method in *Breve.Control*, without this the simulation will not run.

> **Run the simulation to test your controller.**

```
<breve_path>/bin/breve <path to source files>/SimulationController.py
```

Step 2: Basic environment

Our environment is now totally empty. To resolve this we will add a simple floor.

> **In the SimulationController, add the following code in the *__init__* function[1]:**

[1] Make sure that you place it after the call to the parent constructor.

```
# Create a floor for the world
self.floor = breve.Floor()
self.floor.setTextureImage(None)
```

These two lines add a new floor object (which has a ground plane around $Y = 0$). We set the texture to None to prevent the use of the (ugly) default texture.

To improve the visual aspects of the simulation we enable lighting and shadows. This should only be used for demonstration purposes as it makes the simulation run slower.

> **Add the following snippet after the code for the floor:**

```
# Set display settings
self.enableLighting()
self.enableShadows()
self.moveLight(breve.vector(80,100,0))
self.enableReflections()
self.enableSmoothDrawing()
```

Step 3: Creating an agent

After a floor has been added we can add more objects to our world. We will start with agents.

> **Create a new file agents.py**

As we are going to use functions (and classes) of the breve engine, we must import it at the top of our file.

```
import breve
```

> **Add the import statement to the top of our agents file.**

Creating a simple agent

We will start with a very simple agent which will do nothing but stand still. Every agent must extend the *breve.Mobile* class in order for it to move and interact.

Just as with our controller, we have to add a constructor and an *iterate* function.

```
class SimpleAgent (breve.Mobile):

    def __init__(self):
        breve.Mobile.__init__(self)

        print "Created agent"

    def iterate(self):
        None
```

> **Add the class definition above to the agents file.**

> **Try to run the simulation and see what happens.**

As you can see nothing happened yet. This is because we still have to add our agent to the environment. This is done through the simulation controller.

```
self.agent = agents.SimpleAgent()
```

> **Add the snippet to the *__init__* function of the controller, just below the lighting settings.**

We also have to add an import to the controller, so it has access to the agent.

```
import agents
```

> **Add the import to the top of SimulationController.py**

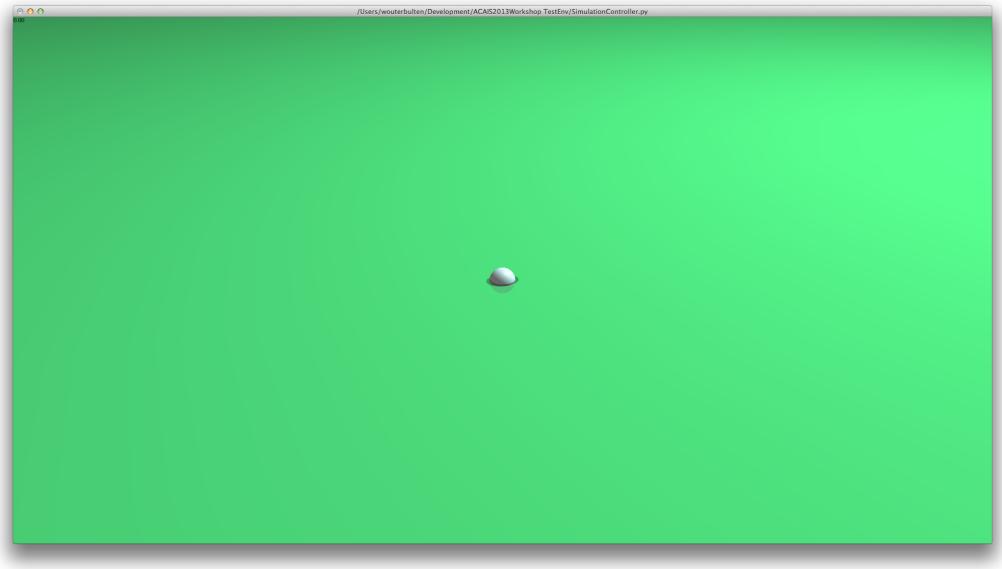


Figure 1: A single very simple agent has been added to the environment.

Movement

This agent is very boring, it just sits there and does nothing. We will create a new agent which wanders around in the world.

```
import wanderer

class RandomAgent (wanderer.WanderingAgent):
    def __init__(self):
        wanderer.WanderingAgent.__init__(self)

        self.setWanderRange(breve.vector(20, 0, 20))

        self.randomizeLocation()

        print "Created agent"

    def iterate(self):
        wanderer.WanderingAgent.iterate(self)
```

> **Add the new agent class below the *SimpleAgent*.**

This *RandomAgent* extends the WanderingAgent class that we created for you [2], so make sure to add the extra import-line at the top of the file. The only setting we have to set is the range of the wanderer, in this case we set it to 20 in a 2D plane. It is important to call the super *iterate* function, otherwise the wander behaviour will not be executed.

> **Add the agent to the simulation by creating one in the constructor of the controller.**

```
self.randomAgent = agents.RandomAgent()
```

[2] Feel free to inspect this class later to see how we build it. What it does is it picks a random spot to go to within its allowed wandering range. It will pick a new place to go to once it has reached its previous target or when it has chased its target for a certain amount of time. Once it has picked a new target spot, it will adjust its course accordingly.

Appearance

Within Breve it is very easy to change the appearance of objects and agents. As an example we are going to change the shape of our agent to a cube. As we will add more agents in the future, we will define one shape object for all agents.

> **Add the following to the constructor of the SimulationController before the line that creates the agent:**

```
# Create a shape for the agents
self.agentShape = breve.createInstances(breve.Cube, 1).initWith(breve.vector↩
    (1,1,1))
```

This single line will create a cube with a dimension of 1x1x1. To be able to use this shape in our agent we add a getter method to the controller.

> **Add the following getter to the simulation controller:**

```
def getAgentShape(self):
    return self.agentShape
```

It is now very easy to change the shape of the agent. Every *Real*[3] object within breve has access to the simulation controller using the variable *controller*. We change the shape like so:

[3] The Real object is a base class for more high-level classes such as Mobile and Stationary.

```
# Set the shape of the agent
self.setShape(self.controller.getAgentShape())
```

> **Try to change the shape of the agent by adding the code to the constructor of the agent. Test it by running the simulation, has the shape changed?**

Creating a group

A single agent cannot do much. If we want to see more emergent behaviour we have to create a group. Luckily, the only thing we have to change to accomplish this is changing the way we instantiate agents. Remember that we used the following code in our controller to create a single agent:

```
self.agent = agents.SimpleAgent()
```

We can now replace this with the following to create any number of agents:

```
breve.createInstances(agents.RandomAgent, 10)
```

> **Remove the lines that created the two agents and replace it with the snippet above. Run the simulation, you should see 10 wandering agents.**

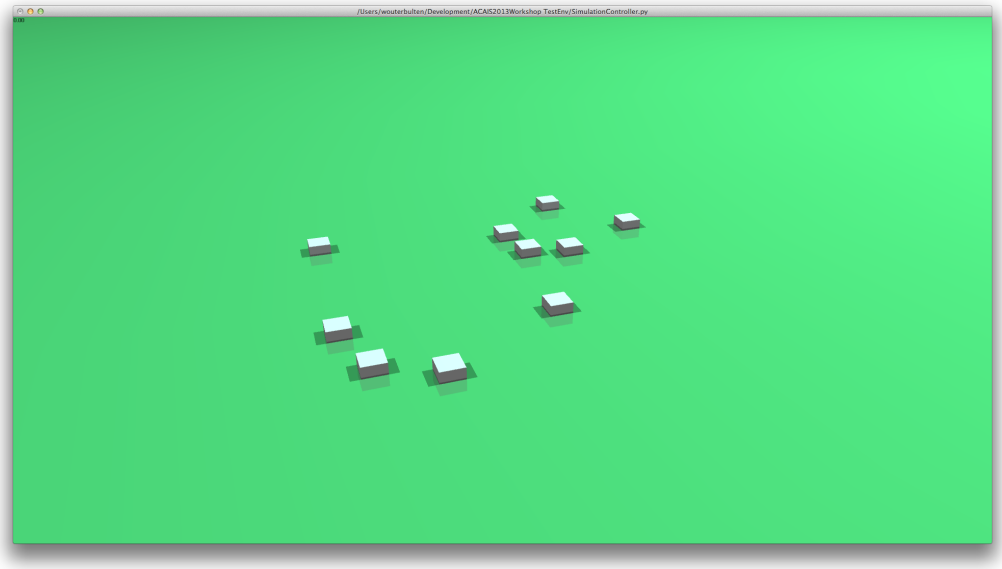


Figure 2: By using the a different method of creating agents we can add multiple to the environment. In this instance 10 random agents.

Step 4: Adding food

The second object we will be adding is food. This gives our agents something that they can interact with.

> **Create a new file food.py**

As we are going to use functions (and classes) of the breve engine, we must import it at the top of our file.

```
import breve
```

> **Add the import statement to the top of our food file.**

Creating the food object

Our initial version of the food is very simple, it will simply stay on the field. Every food source must extend the *breve.Stationary* class. It is not a mobile class like the agent, but rather stationary. We still want to be able to interact with it.

Just as with our controller and agent, we have to add a constructor and an *iterate* function.

```
class SimpleFood (breve.Stationary):

    def __init__(self):
        breve.Stationary.__init__(self)

    def iterate(self):
        None
```

> **Add the class definition above to the food file.**

From the previous section we know that this is not enough. We will also have to add the food to the controller

```
self.SimpleFood = food.SimpleFood()
```

> **Add the snippet to the *__init__* function of the controller, above the code that creates the agents**

We also have to add an import to the controller, so it has access to the food.

```
import food
```

> **Add the import to the top of SimulationController.py**

The food should now have been added to the simulation.

> **Try to run the simulation and see what happens.**
As you can see you have now added a single food object to the field, though it does nothing yet.

Location

Our food is always at the same location when running the program. We want it to be more random. This can be done with the following function.

```
def randomizeLocation(self):
    randomLoc = breve.randomExpression(2 * breve.vector(20,0,20)) - breve.vector↩
        (20,20,20)
    self.move(randomLoc)
```

> **Add the function above to the food class**
Now we just have to call this function when we create the food.

```
self.randomizeLocation()
```

> **Add the line above to the *__init__* function of the food**

Appearance

We will again define one shape for all food sources.
> **Add the following to the constructor of the SimulationController**

```
# Create a shape for the food sources
self.foodShape = breve.createInstances(breve.Sphere, 1).initWith(0.5)
```

This will create a small sphere. To be able to use this shape in our agent we add a getter method to the controller.
> **Add the following getter to the simulation controller:**

```
def getFoodShape(self):
    return self.foodShape
```

Now it is very easy to change the shape of the food. We change it with the following line of code:

```
# Set the shape of the food source
self.setShape(self.controller.getFoodShape())
```

> **Try to change the shape of the food by adding the code to the constructor of the food. Test it by running the simulation, has the shape changed?**

Creating more food

A single food object is not much for all our agents, so we want to add some more. We first added a single food object to the SimulationController with the following code:

```
self.SimpleFood = food.SimpleFood()
```

We now want more food objects, this can be done with the following code:

```
breve.createInstances(food.SimpleFood, 3)
```

> **Replace the first piece of code with the second piece of code to create more food objects**

Step 5: An extra dimension

As an intermediate step, we are going to switch from a 2D plane to a 3D space to illustrate how easy it is to do so in Breve. Additionally, it will gives us a much nicer view of the final result when we get there.

The only changes we need to make are to the *agent* and *food* classes; the *wanderer.WanderingAgent* class which our RandomAgent extends is able to handle 2D and 3D spaces just as easily without requiring any changes in the code.

> **Change the *y-value* of wander range of the RandomAgent from 0 to 20**

```
self.setWanderRange(breve.vector(20, 20, 20))
```

> **Change the food.SimpleFood.randomizedLocation() function so that also a random value is taken for the y-axis**

```
randomLoc = breve.randomExpression(2 * breve.vector(20,20,20)) - breve.vector↩
    (20,20,20)
```

If you run the simulation now, you will notice how the food sources are scattered in 3D space and how the agents will move through them. As agents can now move in an area below y=0 we have to remove the floor.

> **Remove the floor from the simulation controller.**

Application: Gatherers

Now that we have created the basics of our *Artificial Life* simulation, *agents* and *food sources*, it is time to make these interact since currently nothing is happening and agents just wander around aimlessly. To make things more interesting we will implement a working *Gatherers* simulation.

In the Gatherers-simulation agents will move around freely in a world that is rich in food. Every time an agents encounters some food, it will pick it up and carry it with him until it comes across another food source. When it does that, it will drop the first food source next to the second and move on. Eventually, an agent will gather all food and put it all together in one big pile.

Naturally, it can take a while before an agent has done this, especially if it walks around randomly like our agents. A group of agents will be able to achieve this task faster if they work together, which is why you created 10 of them.

Adding interaction between food and agents

Having food that does nothing is not interesting, we want to be able to interact with it. When an agent touches a food source he should pick it up and carry it for a while, before dropping it again.

In our case a food source can only be carried by one agent. So let us start with that. We will have to add an owner to the food to see if it is being carried.

```
self.owner = None
```

> **Add the line initializing the owner to the *__init__* function of the food**

Now we want to be able to get and set this value from other classes. This can be done by adding a getter and a setter.

```
def setOwner(self, o):
    self.owner = o

def getOwner(self):
    return self.owner
```

> **Add the two functions to the food file**

That was all we had to change to the food file. From now on we will be working in the agent file. First we will add whether an agent is carrying an object or not.

```
# Store a possible food source
self.carrying = None
```

> **Add the line initializing the carrying status to the *__init__* function of the agent**

First we will define what happens if an agent collides with a food source. There are several possibilities. The food you collide with could already be carried by someone or you have recently collided with another food source, in these two cases you do nothing. If you are carrying food and you collide with a food source that no one is carrying, you place your current food on the ground. If you are not carrying food and none of the above possibilities apply, you will pick the food up.

```
def collisionWithFood(self, f):
    # Checks if the food is already being carried
    if(f.getOwner()):
        return

    # We want two iterations without an collision
    if(self.collidedTimer > 0):
        self.collidedTimer = 2
        return

    # Set the timer for the collision
    self.collidedTimer = 2

    if(self.carrying != None):
        self.placeFoodObject(self.carrying, f)
        self.carrying = None
        return

    self.carrying = f
    self.carrying.setOwner(self)
```

> **Add the collion function to the RandomAgent class**

You might have noticed that we talked about placing food, something we will implement now. The food you are carrying is placed near the food you collided with.

```
def placeFoodObject(self, ownFood, placedFood):

    location = placedFood.getLocation()
    location = location + ( breve.randomExpression( breve.vector( 2, 2, 2 ) ) - ↩
       breve.vector( 1, 1, 1 ) )

    ownFood.move(location)
    ownFood.setOwner(None)
```

> **Add the placeFoodObject function above to the RandomAgent class**

In the collision function we used a timer which represents how long you have not touched any food. This is so you do not pick up the same object right away. However we have not initialized this variable yet.

```
# Timer to prevent / to inhibit the collision behavior
self.collidedTimer = 0
```

> **Add the line initializing the timer to the *__init__* function of the agent**

Changing the timer will happen in the iterator, so that every refresh this value is decreased.

```
self.collidedTimer -= 1
```

> **Add the line that decreases the timer to the *iterate* function, before the iterate call**

At the moment we have all necessary functions to describe what happens when you collide with food. We are still missing what actually happens to the food while you are carrying it. When you are carrying food the location of the food should be adjusted along with your location.

```
if(self.carrying):
    self.carrying.move(self.getLocation() - breve.vector(1,0,0))
```

> **Add the if and its body to the *__init__* function of the agent**

We now have all necessary functions to interact with and carry food. But our agents do not know yet when they actually collide with food. For this we will use a standard breve function. First you give the object that it is colliding with and next you give the function that should be executed in case of collision.

```
# Setup the collision
self.handleCollisions('SimpleFood', 'collisionWithFood')
```

> **Add the breve function and its parameters to the *__init__* function of the agent**

Creating two groups: Red versus Blue

Now that we have finished creating a basic Gatherers simulation, it is time to go one step further. When creating an Artificial Life simulation, most of the time you will want to create different conditions of the environment and compare them to each other. We won't really make different conditions, but we will create two groups of agents who will compete with each other in the same world: the *Red Agents* and the *Blue Agents*.

First, we will create two new classes, *RedAgent* and *BlueAgent* underneath our RandomAgent. These new types of agent will extend our RandomAgent. Then we need a property *group* to indicate to which group they belong. We will also give them a color so we can distinguish them from each other ourself when we watch the simulation. The color can be specified with the setColor(r, g, b) function where r, g, b are values between 0 and 1 for *red, green* and *blue* respectively.

These extra properties should be set in the constructor. Naturally, these classes will also need an iterate function. However, these will do nothing special so simply calling the iterate function of the superclass is sufficient.

> **Add the RedAgent and BlueAgent classes below the RandomAgent class**

```
class BlueAgent (RandomAgent):
    def __init__(self):
        RandomAgent.__init__(self)
        self.setColor(breve.vector(0.2,0.2,0.8))
        self.group = 1
```

```
    def iterate(self):
        RandomAgent.iterate(self)


class RedAgent (RandomAgent):
    def __init__(self):
        RandomAgent.__init__(self)
        self.setColor(breve.vector(0.8,0.2,0.2))
        self.group = 2

    def iterate(self):
        RandomAgent.iterate(self)
```

We will also want to give the RandomAgent a default group it belongs, just to make sure that we will not brake the interaction between food and agents in a bit.

> **In the constructor of RandomAgent, add a line stating that it belongs to group 0**

```
self.group = 0
```

Now that we created two new classes for the two groups of agents, we will have to make sure they are actually going to compete with each other (by stealing each other's food). We can do this by taking into account the group an agent belongs to when it interacts with a food source.

The first step is to mark the food that an agent picks up as belonging to the group of that agent. In order to do this we must first update the *SimpleFood* class so that it supports belonging to a group. We do this by adding a property *group* in the constructor, just like we did for the agents. As all food starts off as neutral, it will belong to group 0.

> **In the constructor of SimpleFood, add a line stating that it belongs to group 0 (it is still neutral)**

```
self.group = 0
```

Of course, we will also want to be able to check to what group food belongs to and change it. Thus, we must create a getter and setter function for the group-property.

> **Add a getter and setter for the group-property at the bottom of the class**

```
def setGroup(self, o):
    self.group = o

def getGroup(self):
    return self.group
```

Now we should make use of this property by making agents state that the food they pickup belongs to their group. We can do this by updating the *collissionWithFood()* function you created and changing the group-property of the food source with the setter we just created. While we are doing that, we will change the color of the food source as well to be the same as the color of the agent so we can see what is happening in the simulation.

> **Update the RandomAgent's collisionWithFood() function to update the group and**

color of a food source. Add the following lines to the bottom of the function

```
self.carrying.setColor(self.getColor())
self.carrying.setGroup(self.group)
```

Next up, we want to make sure that agents only deposit the food they are carrying near either neutral food or food of their group has already claimed, since they definitely do not want to put it down near the stack of the other group!

To do this, we must once again update the collisionWithFood() function and check to what group the newly encountered food belongs to. If it is neutral food or belongs to the agent's group, the food should be dropped down and the agent can continue on its way. If it belongs to the other group, the agent should hold on to his food and ignore the newly encountered food source. We can do this by simply making the function return.

> **Update the RandomAgent's collisionWithFood() function by adding the follow snippet directly below the if(self.carrying != None):-statement:**

```
if(self.carrying.getGroup() != f.getGroup() and f.getGroup() != 0):
    return
```

That's it! That's all there is needed in order to create two competing groups of agents that steal each other's food and only put down the food they are holding when they encounter neutral food or food already belonging to their group!

The last thing to do is updating the SimulationController by removing the *breve.createInstances(agents.RandomAgent, 10)* line and replacing it with two lines that create Red and Blue agents.

> **Update the SimulationController by creating Red and Blue agents:**

```
breve.createInstances(agents.RedAgent, 10)
breve.createInstances(agents.BlueAgent, 10)
```

Now run the simulation and watch the agents claim and gather food for their groups.