Zena's CS231 Project 4

This week's project was all about interaction between instances of rule-following dots, aka "agents". We observed the behavior of different types of groups as they moved around on the window they were drawn in.

Task 1 was to create an agent class. This would be what every dot represents and is based off of. Each agent has a position associated with it, and a way to get and set those x and y values. We also define an updateState method, which we will use in the groupers, but which needs to be included in agent as well because later we have an arraylist of agents and call updateState on each.

Task 2 was to make a grouper class. This extends agent, which we do because we want to make agents of this type behave differently, but still be considered agents. For example, this first kind would have the tendency to clump together when close to others. But for now, we just made a draw method, which draws a black circle, and a constructor.

Task 3 was to make a landscape class. This was based off of the one from project 2, and has methods such as get height, get width, addAgent, and getNeighbors. Then there is also a method to draw them, which just calls the draw method of every agent. The getNeighbors method was the most difficult part, as I had to break down the distance formula into code using the math package. I used the formula with the x and y values of the current agent as well as the parameters passed in for the check, then compared that to the radius. In other words, I checked how far an agent was from a given point and if it was close enough, I added it to an arraylist of agents I made and returned.

In the fourth task, we just implemented a graphics interface for the agents. This meant that we could see all of the agents represented as dots on the screen. To do this, we used the LandscapeDisplay class that we have been using already, and modified a few things. We created new groupers, and added them to the display, then repainted.

For task 5, we made a method in grouper to update the state. This was where we indicated how the movement patterns should look like. For example, for the default groupers, if there was a neighbor nearby, we made them have a 1% (so basically no) chance of moving. Otherwise, they would move in the x and y direction according to a random double between -5 and 5.

In task 6, we added a function to update all of the agents to the landscape class. To do this in the requested random order, I made an arraylist, added all of the agents from the agentlist to it, then called Collections.shuffle on it. Then, I looped through all of the agents in the new list and updated them according to the rules in task 5.

Task 7 had us creating a simulation class for the grouper. Like other simulation classes, it creates a landscape, puts it onto the display, and then loops over for however many iterations are given in the command line. Each time, it updates the agents then repaints. We then save the images that are generated.

After that, in task 8, we created a new grouper, which has three different categories, which an object will have one of, specified by a parameter. The dots move around and check if they have a neighbor with the same category. If they do, they will stop (or nearly stop) moving and clump together. I made each category have a different color so it was easy to observe the results. My code to check the neighbors and see if it is an instance of the categorized grouper is below. Note that neighbors returns the agent we care about, as well, meaning that we should check if the neighbor is actually the same as this, which I do here. This fixed my bug where barely anything would move and everything thought it had a neighbor. I use continue to go to the next agent that we want to loop over, skipping the undesirable one.

CategorizedGrouper:

```
for (Agent neighbor : scape.getNeighbors(this.getX(),this.getY(),3)) //for every neighbor
{
    if (neighbor == this) //don't include itself as neighbor
    {
        //continue means go to next iteration of loop
        continue; //go on to next neighbor
    }
    if (neighbor instanceof CategorizedGrouper) //if this is a categorized grouper agent
    {
```

Task 9 was just to test this in a simulation class, which is similar to the one from task 7. I just had to change it so the images have different names, the call to create categorized groupers has another argument, etc.

The last task was to make our own groupers with unique rules. For mine, I made "shy groupers" which move around, and if they have many neighbors, they will move faster (eg farther distance) and turn red. These agents were supposed to avoid each other, as they are very shy and get embarrassed in front of each other like the cute little dots they are. Some of my code for this check is shown below:

ShyGrouper:

```
if (neighborList.size() > 0 && neighborList.size() <= 2) //if there are 1-2 neighbors
{
    this.setColor(Color.pink);
    this.setX((this.getX())+(10*rand.nextDouble()-5)); //move agent
    this.setY((this.getY())+(10*rand.nextDouble()-5));
}
else if (neighborList.size() > 2 )//lots of neighbors
{
    this.setColor(Color.red);
    this.setX((this.getX())+(20*rand.nextDouble()-10)); //move from -10 to 10
    this.setY((this.getY())+(20*rand.nextDouble()-10));
}
```

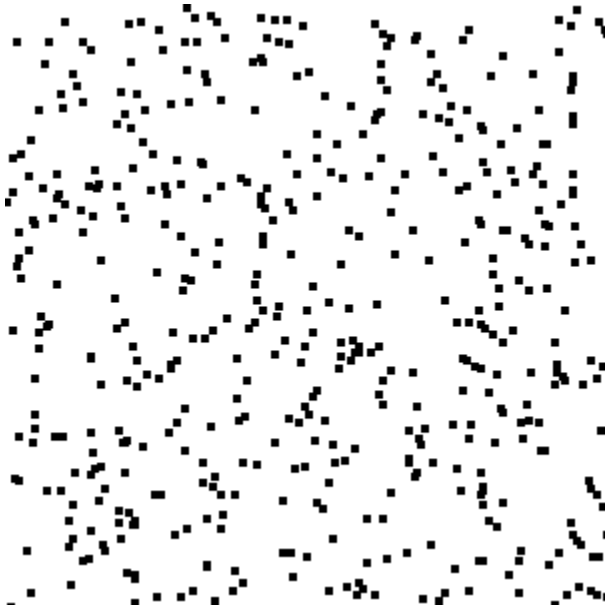
In addition did the second extension, which was to make a simulation that mixes the agents. I made a new simulation class, called MasterGrouperSimulation, and made one of the user inputs be a number indicating which kind of grouper they want to use. I checked which integer they provided, and decided which kind of agent to add to the landscape based on this. I also made changes to how the pictures would be saved based on this in the main method. The main issue with this extension was deciding how many of each kind of grouper to make. To do this, I used code like that below, where I get the remainder of the number of requested agents, and if there is none, I divide the groupers evenly between the number of agents divided by three. A third of them will be in each respective group. For cases where the number is not divisible by three, I get the number of agents requested, and subtract the remainder that prevents it from being divisible. Then, the leftover ones are created as regular groupers.

MixedGrouper:

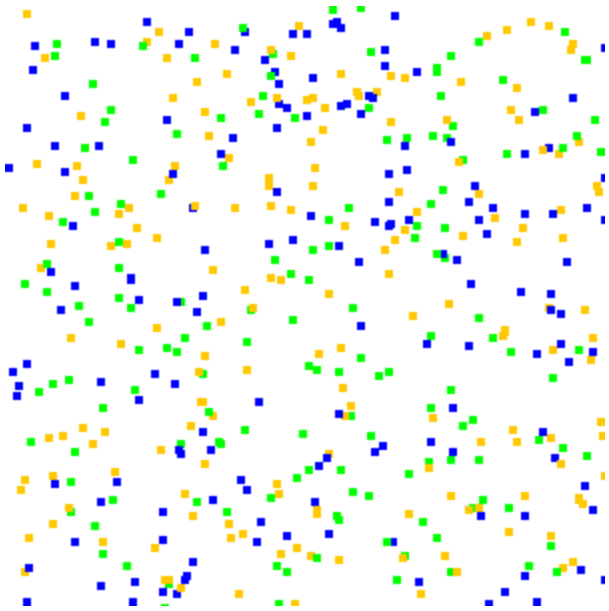
```
else //not divisible by 3
{
    int remainder = (numAgents%3); //the remainder
    int numToDivide = numAgents - remainder; //the number we CAN divide by 3

    for (int i = 0; i < numToDivide/3; i++)
    {
        grid.addAgent(new Grouper(gen.nextInt(grid.getWidth()),gen.nextInt(grid.getHeight())));
        //random coordinates within the width and height of the grid
    }
```

In case it wasn't obvious, I also did the first extension, which is to let the user control which groupers they use without having to use different kinds of simulation files. If they provide 0, it is a regular group, if they choose 1, it is a categorized grouper, if the number is 2, it is a shy grouper, and if it is 4, it is a mixed grouper, from the other extension.

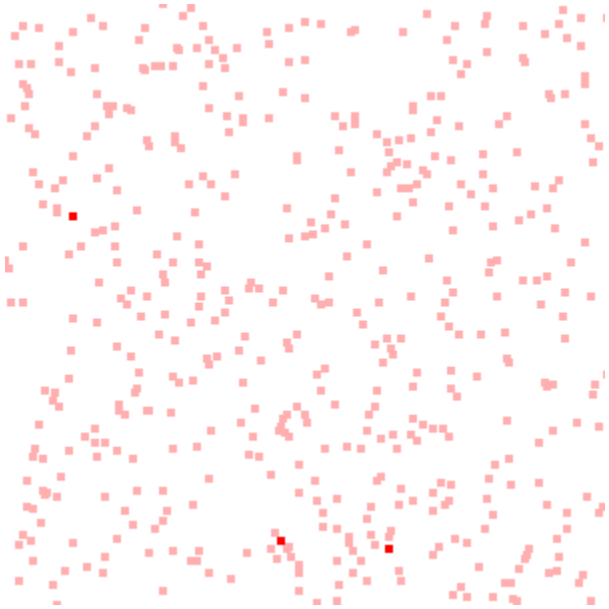
Top to bottom: the results of a simulation with normal Grouper, the CategorizedGrouper, and the ShyGrouper.



The dots here clump together if they are close enough.

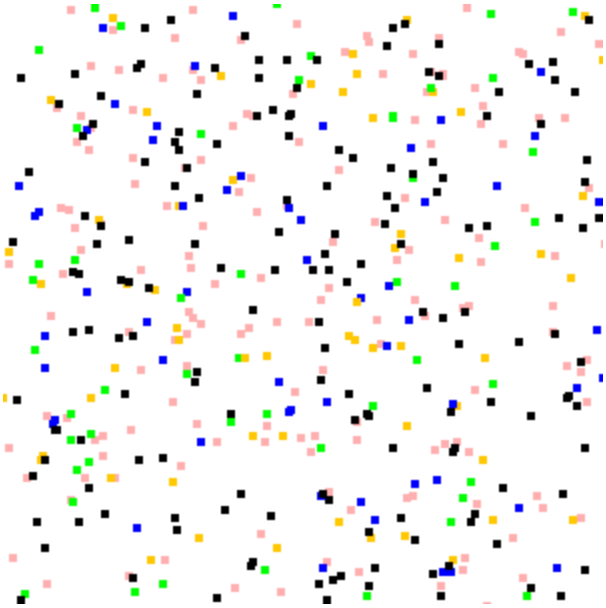


If the dots here are close enough to another of their category, they will stop moving, also resulting in clumping.



The dots here will turn red and move faster if they are around many others.

Below is is the MixedGrouper extension's simulation:



As you can see, the pink ones are shy and do not want to be near each other. The black ones are the normal groupers and will pair up with each other or categorized groupers. They would LIKE to pair up with the shy ones, but because the shy ones move away, they exit the neighbor status and the black ones have to search again. The categorized groupers will still pair up with each other if they are the same color.

This project familiarized me with extending classes and categorizing data with if statements. I also learned what continue does, because I apparently had it wrong before. I also learned about some Math methods. It was interesting to observe the behavior of the various kinds of agents.

Thank you to:

Melody Mao
Stephanie