REPAST JAVA BOIDS MODEL

ROWAN COPLEY - GOOGLE SUMMER OF CODE

0. Before We Get Started

Before we can do anything with Repast Simphony, we need to make sure that we have a proper installation of Repast Simphony 2.0. Instructions on downloading and installing Repast Simphony on various platforms can be found on the Repast website. This tutorial will also assume a familiarity with Java.

1. What is Boids?

Boids is an agent-based model which simulates the steering behavior of flocking birds using simple to understand rules. Applications of the model range from studying Artificial Life to implementing the algorithm in a flock of robots. The advantage of Boids is that it is a bottom-up approach to flocking: each individual Boid need only follow a few simple rules and flocks will emerge naturally. The overall design of the Boids algorithm is based on Craig Reynold's original Boids algorithm consists of three rules:

(1) Alignment: Each Boid steers itself to align its heading with that of its neighbors
(2) Cohesion: Each Boid steers itself to the average position of its flockmates
(3) Separation: Each Boid steers itself in order not to crowd its neighbors

These three rules can conflict with each other, as we will see, the way that we deal with those conflicts will influence the final behavior of the flocks of Boids. When finished, the model should look like Fig. 1.

2. Setting up the framework

Open Repast and switch to the Java perspective by clicking the Java button in the upper-hand corner (Fig 2). Next, we need to create a new Java project for Repast. To do this properly, click File, go to New, and click Other. Find Repast Simphony Project in the Wizard selection box and click it (Fig. 3). Click Next. Name your project Boids and click Finish. Your newly created Boids folder should appear in your Package Explorer window pane on the left-hand side of the Repast window. Double click on the folder or click the arrow to the left of the folder to explore the contents.

The only files we're going to create right now are in the /src subfolder, so the rest of the project files can safely be ignored. Click on the arrow to the left of the /src subfolder. A subfolder with your project's name will drop down underneath /src. The two files inside

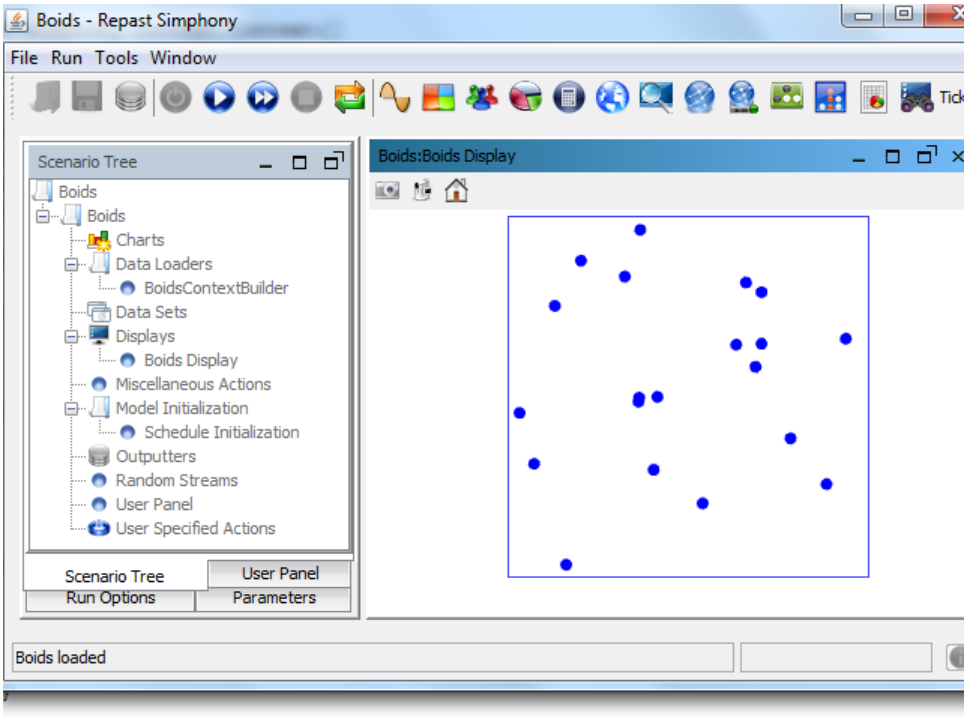

FIGURE 1. Completed Boid Model

of this subfolder can safely be deleted. In their place we can make a new Java file called Boid. To do this, right-click on the folder called Boids inside /src and click New, then click Class. Name it Boid and click Finish.

Your project's hierarchy should look like Fig. 4.

Next, add in code to your Boid class so it looks like this:

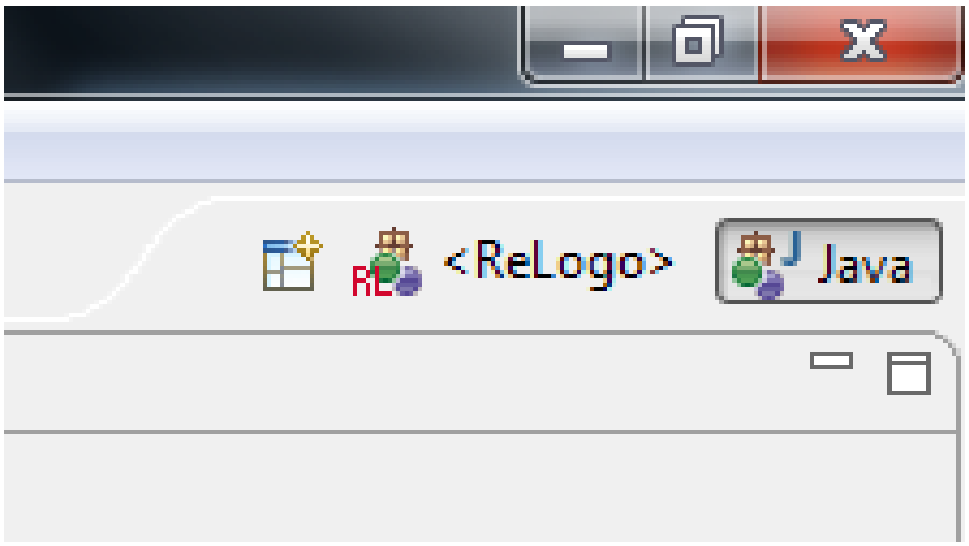

FIGURE 2. Changing to Java perspective.

```
1  package Boids;
2
3  public class Boid {
4     double distance = 0.2;
5     double heading;
6     ContinuousSpace<Object> space;
7     Grid<Object> grid;
8
9     public Boid(){
10       distance = 0.3;
11       heading = 0;
12    }
13
14    @ScheduledMethod(start = 1, interval = 1)
15    public void step(){
16
17    }
18 }
```

LISTING 1. Boid Class constructor and placeholder step() method.

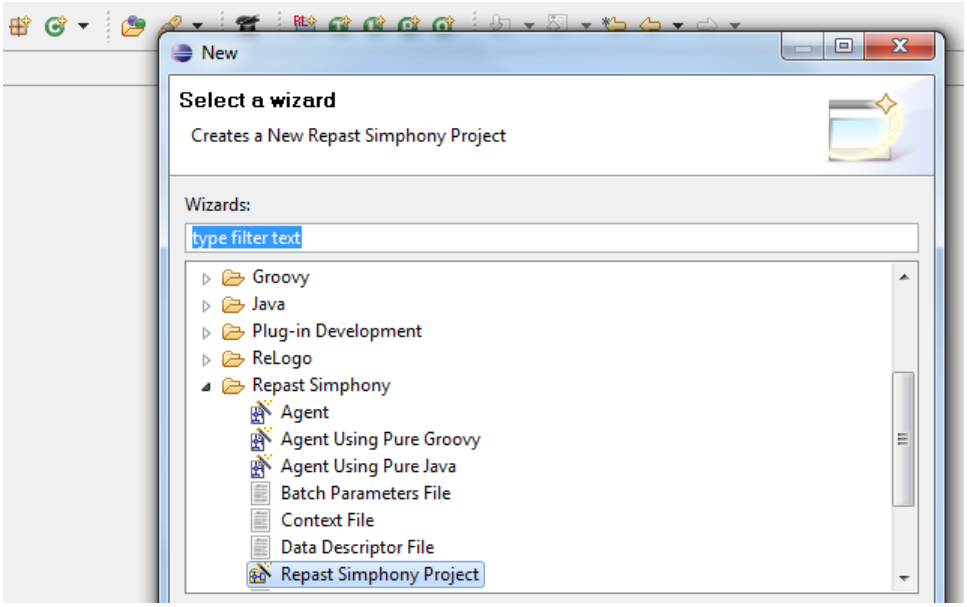

FIGURE 3. Creating a Repast Simphony project.

The method named `Boid` will be a constructor that we'll return to when we create the `ContextBuilder` for the Boids. The `step()` method is a method that will be called at every tick, or time step, of the simulation.

You'll notice that the `@ScheduledMethod` command is underlined in red. This is because we haven't made the necessary import for the command. There are two ways of doing this. You can hover over the command until a popup window appears and click on Import ScheduledMethod. The second way is to click on Source in the top menu and click on Organize Imports. As we add more code, you should automatically check for an import any time a command is underlined in red.

2.1. **step() and forward() methods.** Here is the code we'll use for now for `step()`:

```
1 public void step(){
2     forward();
3 }
```

LISTING 2. step() method.

Now create the forward method below `step()`:

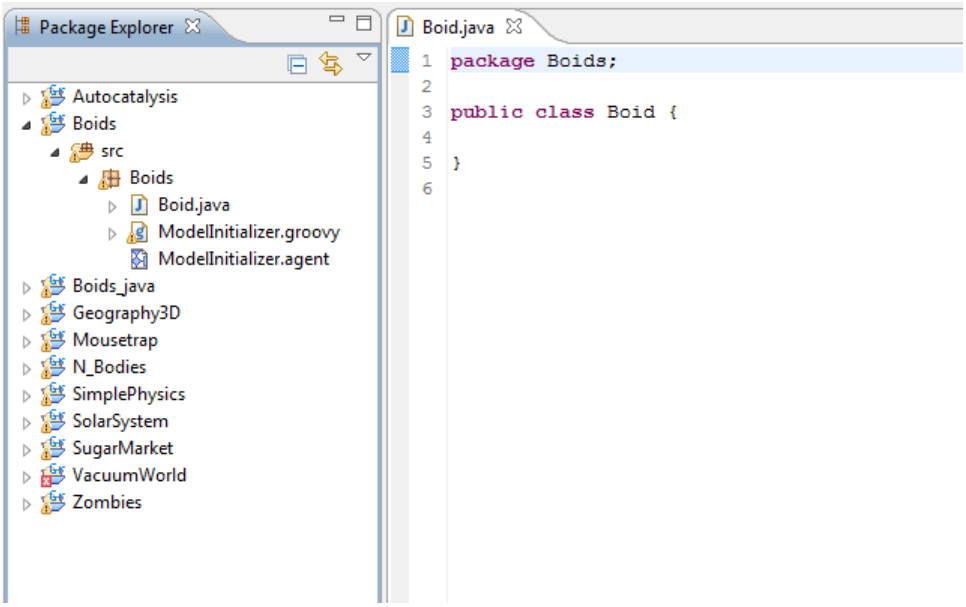

FIGURE 4. A newly created project's hierarchy.

```
1 public void forward(){
2     NdPoint pt = space.getLocation(this);
3     double moveX = pt.getX() + Math.cos(heading)*distance;
4     double moveY = pt.getY() + Math.sin(heading)*distance;
5     space.moveTo(this, moveX, moveY);
6     grid.moveTo(this, (int)moveX, (int)moveY);
7 }
```

LISTING 3. forward() method.

In order to understand this block of code, you must first understand the underlying representations of space that Repast uses. In this particular model, we use both kinds: we use a `Grid`, where each object has a location on a grid of discrete cells; and we use a `Continuous Space`, where each object has a location defined by two coordinates. Both the grid representation of space and the continuous space representation of space are called projections. We'll put the actual projections into our code later, for now leave them underlined in red.

Whenever an object moves itself and we are using these two projections, we need to update both of them. This means that we will need to convert between one projection's

coordinates and another's. Line 2 creates an object of type `NdPoint` (the 'Nd' stands for N-dimensions) which contains a representation of the Boid's current coordinates in the continuous space. Lines 3-4 take the Boid's current coordinates and change them so that the Boid has moved by a total of 0.3, the value in `distance`, in the direction of `heading`.

2.2. **Context Builder.** Create a new class in the Boids package and name it `BoidsContextBuilder`. Let's do the simple stuff first:

```
1  public class BoidsContextBuilder implements ContextBuilder<Object>{
2    public Context build(Context<Object> context) {
3         context.setId("Boids");
4         int width = 16;
5         int height = 16;
6         int numBoids = 20;
7
8         //placeholder
9
10        return context;
11     }
12 }
```

Listing 4. Context class build() method.

Note that this class is implementing an interface. Another thing to notice is that when you try to find the right import option for Context, there are lots of options that show up. Make sure to scroll down the list until you find the one from Repast. In general, you should always import code from Repast rather than other libraries.

Next, we need to create the two space projections, our `grid` and our `space`. This will go where our `//placeholder` comment is right now. Here is the code:

```
1  ContinuousSpaceFactory spaceFactory =
2     ContinuousSpaceFactoryFinder.createContinuousSpaceFactory(null);
3  ContinuousSpace<Object> space = spaceFactory.createContinuousSpace
4     ("space",context,new RandomCartesianAdder<Object>(), new
5     repast.simphony.space.continuous.WrapAroundBorders(), width,height);
6
7  Grid<Object> grid = GridFactoryFinder.createGridFactory(null).createGrid
8     ("Grid", context, new GridBuilderParameters<Object>
9     (new repast.simphony.space.grid.WrapAroundBorders(), new
10    SimpleGridAdder<Object>(),true, width,height));
```

Listing 5. Continuous Space and Grid projection creation in build() method.

This complicated-looking snippet of code creates two objects, called factories, which in turn create two more objects, `grid` and `space`, our projections of space. The parameters that these methods (Lines 3-5 and 7-9) take amount to the following:

(1) name: an arbitrary ID that we assign to this projection to refer to it elsewhere
(2) context: this is the object that contains all the objects that we are dealing with in this simulation
(3) method of adding objects to the space projections: defines whether the Boid object is added automatically to the projection and how it is added.
(4) size of dimensional axes: defines how many dimensions the projections represent and how far those dimensions extend.
(5) for the grid object, a boolean parameter is used to define whether each grid cell can contain multiple objects

Let's go over the commands we used for the method of adding objects to our two projections. For the continuous space, we have defined a `RandomCartesianAdder`, which means that whenever we add a Boid to the context with the command `context.add(boid)` as we do in the next step, the Boid automatically gets added to the continuous space at a random location. For the grid, the Boid will not be physically located on the grid until we manually put it somewhere.

The other point of confusion here is that there is another object being created in the parameters for the method `createGrid()`, `GridBuilderParameters`. We can think of this as merely a carrier or wrapper for the parameters inside of it. For a better understanding of the commands being invoked, refer to the Repast Java API, specifically the pages GridBuilderParameters, ContinuousSpace and Grid.

The last thing we need to add to the method is creation of the Boid objects and the addition of them to the context. We do that with this code snippet, which goes right underneath the last snippet of code:

```
1 for(int i = 0; i < numBoids; i++){
2     Boid boid = new Boid(space, grid);
3     context.add(boid);
4     NdPoint pt = space.getLocation(boid);
5     grid.moveTo(boid, (int)pt.getX(), (int)pt.getY());
6 }
```

Listing 6. Boid creation in build() method.

Lastly, we modify the `Boid()` constructor.

2.3. **Boid().** This is how the constructor in the Boid class should look:

```
1 public Boid(ContinuousSpace<Object> space, Grid<Object> grid){
2     this.space = space;
3     this.grid = grid;
4     heading = 0;
5 }
```

LISTING 7. New Boid constructor.

2.4. **Modify context.xml.** Before we start up our model, we need to make it aware of our two projections. We do this by modifying a file named `context.xml`. In order to see the folder that it's contained in, however, we need to disable a filter to show all the files that are part of our Boids project. Find the white arrow that's part of the menu immediately to the right of your Package Explorer tab (Fig. 5). Click it, and find the menu item ReLogo Resource Filter. Uncheck it if it's checked. Now there should be more subfolders in your project folder.

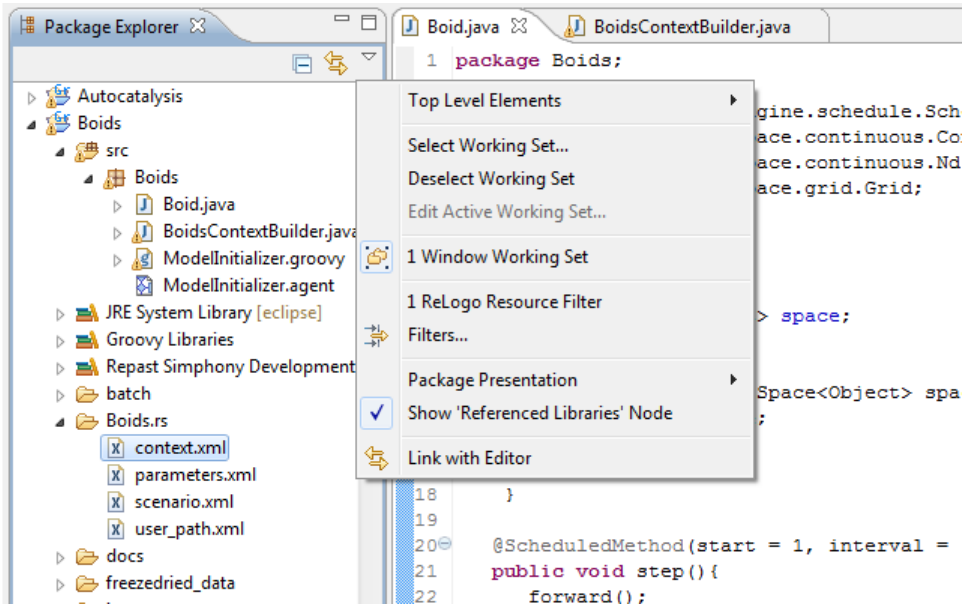

FIGURE 5. Turning off the ReLogo resource filter.

Find Boids.rs, expand it, and open context.xml. Right-click on the menu item named context and go to Add Child, then select projection. Do this once again, so we have two projections that are children of context. Find the drop-down menu for Type, and make

one projection a continuous space and the other a grid. Now use the ID tag to name the grid projection "grid" and the continuous space projection "space" (without the quotes). These IDs should correspond exactly with the IDs we assigned ourprojections in our build() constructors.

2.5. **Set up a display.** Click on the arrow to the right of the green play button to see the popup menu of all the run options for your models. Find Boids Model and click it. In a minute, a Java window will appear.

Make sure the menu to the left is on the Scenario Tree tab. First, find Data Loaders and delete the menu item beneath it. Right-click Data Loaders and click New Data Loader. Click on Custom ContextBuilder Implentation, then click next. The class we just made should be automatically selected. Click next, then finish. Now find Displays and right-click on it, and click Add Display. Name it something like "Boids Display" and move "space" from the left-hand box to the left-hand box. You can do this by clicking on "space" and then clicking on the green arrow that is pointing to the right.

Make sure that the Type drop-down menu is set to 2D. If the default is GIS, then there is something wrong with your `BoidsContextBuilder`.

Click next. This menu lets you tell the program which class(es) will be the agents in the simulation. Move Boid to the right-hand box and click next.

This menu lets you define how the Boids look. If you click the far-right button (the one to the right of the Style Class text box) you can customize the look of your agent. Go ahead and click on it. The default is a blue circle, and that's good for now. Click okay, and then click next twice, and finally click finish.

Your model should now run! Click on the blue power button, and then click on either the blue next button or the blue play button to step the simulation forward by one step or many steps, respectively. Your Boids will all move in the same direction, initially, since their `heading` variables are all set to 0. Make sure to save your simulation's display settings using the save button at the upper left-hand corner.

3. Alignment

This method needs to average the heading of all Boids within a certain distance (referred to as its 'neighborhood'). This average is then returned by the `alignmentDirection()` method as a measurement of radians.

But first, we need a method to actually create the neighborhood.

3.1. **neighborhood() method.** This method will return an `ArrayList` of type Boid with all the Boids that we want to be included in the neighborhood of the Boid calling the method.

```
1  public ArrayList<Boid> neighborhood(){
2     MooreQuery<Boid> query = new MooreQuery(grid,this);
3     Iterator<Boid> iter = query.query().iterator();
4     ArrayList<Boid> boidSet = new ArrayList<Boid>();
5     while(iter.hasNext()){
6        boidSet.add(iter.next());
7     }
8     Iterable<Object> list = grid.getObjectsAt(grid.getLocation(this).getX(),
9        grid.getLocation(this).getY());
10    for(Object boid : list){
11       boidSet.add((Boid) boid);
12    }
13    boidSet.remove(this);
14    SimUtilities.shuffle(boidSet,RandomHelper.getUniform());
15    return boidSet;
16 }
```

LISTING 8. neighborhood() method in Boid class.

Lines 2-3 declare an object that can iterate through all Boids in the Moore neighborhood of the calling Boid. In 5-7 we add these all to an `ArrayList`. However, since the Boids on the same cell as the calling Boid aren't included in the Moore neighborhood, we need to add those too, which is done in 10-12. Finally, we need to remove the calling Boid so it doesn't count itself.

Note that ArrayList¡¿ is a Java object, you'll need to import it through java.utils. Also, the correct import for Iterator is `import java.util.Iterator;`

3.2. **alignmentDirection() method.** This method averages the direction in which the neighboring Boids are traveling.

```
1  public double alignmentDirection(){
2     ArrayList<Boid> boidSet = neighborhood();
3     double averageHeading = heading;
4     for(Boid boid : boidSet){
5        averageHeading = averageTwoDirections(averageHeading,towards(boid));
6     }
7     return averageHeading;
8  }
```

LISTING 9. alignmentDirection() method in Boid class.

3.3. **averageTwoDirections().** This method is fundamental to the workings of the model, because all three steering methods depend on it to work.

```
1  public double averageTwoDirections(double angle1, double angle2){
2     assert(angle1 <= Math.PI*2);
3     assert(angle2 <= Math.PI*2);
4     angle1 = Math.toDegrees(angle1);
5     angle2 = Math.toDegrees(angle2);
6     if(Math.abs(angle1-angle2) < 180){
7        return Math.toRadians((angle1+angle2)/2);
8     }
9     else return oppositeDirection( (oppositeDirection(angle1) + oppositeDirection(ar
10 }
```

LISTING 10. averageTwoDirections() method in Boids class.

The astute reader may have noticed that, as it currently stands, the `alignmentDirection()` method won't give a very accurate average; the average will be weighted towards the boids it averages last. However, in order to keep things as simple as possible, we'll omit this from our model for now and leave the solution up to the reader to figure out. Since we've added in the line to shuffle the neighborhood `ArrayList`, this won't bias the Boids to travel in a certain direction.

3.4. **oppositeDirection().** Here is the method that averageTwoDirections() relies on.

```
1  public double oppositeDirection(double angle){
2      assert(angle <= Math.PI*2);
3      angle = Math.toDegrees(angle);
4      return Math.toRadians((angle + 180) % 360);
5  }
```

LISTING 11. oppositeDirection() method in Boid class.

If you want to see your model run using just the `alignmentDirection()` steering method, add this line of code to the end of `step()`: `heading = alignmentDirection();`

3.5. **towards().** This method returns the distance from the calling Boid to the Boid being passed in as an argument.

```
1 public double towards(Boid boid){
2     NdPoint myPoint = space.getLocation(this);
3     NdPoint otherPoint = space.getLocation(boid);
4     return SpatialMath.calcAngleFor2DMovement(space, myPoint, otherPoint);
5 }
```

Listing 12. towards() method in Boid class.

4. Cohesion

In order for a group of birds to be labeled a flock, or a group of fish to be labeled a school, they need to have a certain proximity to each other.

4.1. **cohesionDirection() method.** This method returns the direction that the Boid should turn in order to get closer to its neighboring Boids.

```
1  public double cohesionDirection(){
2     ArrayList<Boid> boidSet = neighborhood();
3     if(boidSet.size() > 0){
4        double avgBoidDirection = heading;
5        double avgBoidDistance = desiredDistance;
6        for(Boid boid : boidSet){
7           avgBoidDirection = averageTwoDirections(boid.getHeading(),avgBoidDir
8           avgBoidDistance = avgBoidDistance + distance(boid)/ 2;
9        }
10       if(avgBoidDistance > desiredDistance){
11          moveTowards(avgBoidDirection,0.2);
12          return avgBoidDirection;
13       }else{
14          return heading;
15       }
16    }else{
17       return heading;
18    }
19 }
```

Listing 13. cohesionDirection() method in Boid class.

In order to fly close to their flockmates, each Boid takes all its neighboring Boids using the `neighborhood()` method and determines the average direction they are travelling in lines 6-9. Also in lines 6-9 it determines the average distance of all the Boids currently in its neighborhood. If the neighboring Boids are too far away from this particular Boid, then it shifts its course to face the average direction of its neighboring Boids in 10-12.

We rely on the methods `distance()` and `moveTowards()`, shown below. We also need to define the constant `desiredDistance` and a getter for the `heading` variable. Put this line of code at the beginning of the class: `final double desiredDistance = 0.5;`. Create method that returns heading named `getHeading()`.

4.2. **moveTowards() method.** This method can move the Boid in any direction in space, without regard to its current heading.

```
1 public void moveTowards(double direction, double dist){
2    NdPoint pt = space.getLocation(this);
3    double moveX = pt.getX() + Math.cos(heading)*dist;
4    double moveY = pt.getY() + Math.sin(heading)*dist;
5    space.moveTo(this, moveX, moveY);
6 }
```

Listing 14. moveTowards() method in Boid class.

4.3. **distance().** This method returns the distance between the Boid calling the method and the Boid passed in as an argument.

```
1 public double distance(Boid boid) {
2    NdPoint myPoint = space.getLocation(this);
3    NdPoint otherPoint = space.getLocation(boid);
4    double differenceX = Math.abs(myPoint.getX() - otherPoint.getX());
5    double differenceY = Math.abs(myPoint.getY() - otherPoint.getY());
6    return Math.hypot(differenceX, differenceY);
7 }
```

Listing 15. distance() method in Boid class.

Your code should now be able to run. If you want to see your model run using just the cohesionDirection() steering method, add this line of code to the end of step(): `heading = cohesionDirection();` If you want to see both methods in action together, try adding this instead:

```
heading = averageTwoDirections(cohesionDirection(),alignmentDirection());
```

5. Separation

Real flocks have another key trait: the units comprising them are fairly evenly spaced, and no one unit is crowded out.

5.1. **separationDirection() method.** This method steers the Boid away from neighboring clusters of Boids if they are too close.

```
1  public double separationDirection(){
2     ArrayList<Boid> boidSet = neighborhood();
3     ArrayList<Boid> boidsTooClose = new ArrayList<Boid>();
4     for(Boid boid : boidSet){
5        if(distance(boid) < desiredDistance){
6           boidsTooClose.add(boid);
7        }
8     }
9     if(boidsTooClose.size() > 0){
10       double avgBoidDirection = heading;
11       double distanceToClosestBoid = Double.MAX_VALUE;
12       for(Boid boid : boidsTooClose){
13          avgBoidDirection = averageTwoDirections
14             (boid.getHeading(),avgBoidDirection);
15          if(distance(boid) < distanceToClosestBoid){
16             distanceToClosestBoid = distance(boid);
17          }
18       }
19       if(distanceToClosestBoid < desiredDistance*0.5){
20          moveAwayFrom(avgBoidDirection, 0.2);
21       }
22       if(toMyLeft(avgBoidDirection)){
23          return (heading+turnSpeed)%Math.PI*2;
24       }else{
25          return (heading-turnSpeed)%Math.PI*2;
26       }
27    }
28    return heading;
29 }
```

LISTING 16. separationDirection() method in Boid class.

In this method, all neighboring Boids which are not too close can be ignored. We define too close as being closer than the desired distance. In order for the Boids to not get tangled up, we allow a little lateral movement–if there is a Boid that is far too close (defined as being within half the desired distance) then we move directly away from it using `moveAwayFrom()`.

No matter what, if there are any too-close Boids, the current Boid needs to turn away from them by some amount, which we'll call `turnSpeed`. We'll need to add in the declaration of this variable at the top of the class with this line: `final double turnSpeed = 0.4;`.

5.2. **toMyLeft() method.** This method returns true if the Boid passed as argument is on the left-hand side of the calling Boid.

```
1  public boolean toMyLeft(double angle){
2     double angleDegrees = Math.toDegrees(angle);
3     double headingDegrees = Math.toDegrees(heading);
4     if(Math.abs(angleDegrees - headingDegrees) < 180){
5        if(angleDegrees < headingDegrees){
6           return true;
7        }else return false;
8     }else{
9        if(angleDegrees < headingDegrees){
10          return false;
11       }else return true;
12    }
13 }
```

Listing 17. toMyLeft() method in Boid class.

6. Putting it all together

6.1. **step().** Now let's combine the three steering methods we established above into the step method.

```
1  public void step(){
2     assert(heading <= Math.PI*2);
3     forward();
4     heading = averageTwoDirections(averageTwoDirections
5           (averageTwoDirections(cohesionDirection(),heading),
6           averageTwoDirections(separationDirection(),heading)),
7        alignmentDirection());
8  }
```

Listing 18. New step() method for Boid class.

The rather bulky Lines 4-7 are in effect averaging together five different directions. This implementation lets the result of `alignmentDirection()` have much greater bearing on the outcome than the other methods because it causes the Boids to move more smoothly. Your model will run as is, so try other implementations and see if they are more to your liking.

6.2. **Adding random weights.** To make this model look more lifelike, we can add random weighting to the three steering methods. First, let's randomize each Boid's starting `heading` so that each simulation is different. Put this method somewhere in your class:

```
1 public double randomRadian(){
2     return Math.random()*(2*Math.PI);
3 }
```

Listing 19. randomRadian() method for Boid class.

In the Boid() constructor, set `heading = randomRadian();`.

Now declare two doubles named `cohesionWeight` and `separationWeight` inside the `Boid` class.

```
1 final double separationWeight =
2    RandomHelper.nextDoubleFromTo(0.0, 1.0);
3 final double cohesionWeight = 1.0 - separationWeight;
```

Listing 20. Code to be added to beginning of Boid class.

Now find the following line in `cohesionDirection()`: `moveTowards(avgBoidDirection,0.2);` and change that to this: `moveTowards(avgBoidDirection,0.2*cohesionWeight);`.

Do the same for `separationDirection()` using `separationWeight`. Now the Boids won't all behave exactly alike.

7. Extending the model

You're now done with the basics of this model. If you want to continue to work on it, here are some suggestions:

(1) The Boids currently behave differently from flocks of birds in how they move around. Tweak variable values to see how differently they behave with slight changes to the code.
(2) Slow down the simulation. Look at the code of the included Boids model for a way to slow down the simulation using a `Thread.sleep()` command.
(3) The separation and cohesion steering methods are weighted, but the alignment is not. Find a way to vary how influential that steering method is on a Boid.
(4) Create a new class, BoidOfPrey. Objects of this class will prey on Boids, and Boids will do their best to avoid them. You'll have to figure out how to weight the new steering method to avoid the predator well when the Boid can see it, but also ignore the predator when it's far enough away.

8. Sources

Craig Reynold's webpage on Boids: http://www.red3d.com/cwr/boids/