OOP – Laboratory 1 Report

GitHub Repo: LucaFranceschi01/UPF-POO22-G101-07

Introduction:

Vec2D:

This class is basically made for creating 2D vectors from an X value and a Y value. It had already many methods inside used in the Lab such as normalize, getX, getY, etc. Apart from the setters, getter and constructors given, we decided to add an extra method that is called scalarProductVec2D:

```
// Scalar product
public Vec2D scalarProdVec2D(double scalar) { return new Vec2D(scalar*x, scalar*y); }
```

This method receives a number in the format of a double and what does is multiply each component of the Vec2D vector by the number introduced. We used this method in the updatePosition.

Agent:

This class is basically creating a new object that has the following attributes: position, direction, target, radius and speed. The Agent object is basically a kind of ball that moves following its direction towards a specific target, inside of a pre-delimited world (which we will talk more about it in the World class). We defined the attributes related to direction, position and target as Vec2D, using the given class. We also implemented an overload on the constructor method. As before, we have the basic getters and setters necessary for this lab, with an overload on some setters. For us, the most relevant methods to explain are: updatePosition, targetReached, isColliding and paintAgent.

```
public void updatePosition(){
    pos.add(dir.scalarProdVec2D(speed));
}
```

In this method what we do is basically add to the position we had the multiplication of the direction of the Agent time its speed (using the method created by us in the Vec2D class.

```
public boolean targetReached(){
    Vec2D difference = new Vec2D(target);
    difference.subtract(pos);
    if(difference.length() <= radius){
        return true;
    }
    return false;
}
```

In this method what we did was create a new vector Vec2D that was equal to the difference between the target and the position of the Agent. Then we calculated the length of the difference vector using the function from Vec2D class and compare it to the radius of the Agent. If the length

is smaller than the radius it means it has reached the target and returns true. Else, it means the Agent has not reached the target yet and returns false.

```
public boolean isColliding(Agent A){
    Vec2D difference = new Vec2D(A.pos);
    difference.subtract(pos);
    if(difference.length() <= A.radius+this.radius){
        return true;
    }
    return false;
}
```

The idea behind the isColliding method is really similar to the targetReached one, but now, instead of computing the length of the difference between the Agent and the target we compare it to the sum of the radius of the 2 Agents we want to know if they are colliding. Again, if the length is smaller, it means they are colliding, so it returns true, and else returns false.

```
public void paintAgent(Graphics g) {
    int x = (int) (pos.getX() - radius);
    int y = (int) (pos.getY() - radius);
    int d = (int) (2 * radius);

    g.setColor(Color.RED);
    g.fillOval(x, y, d, d);
}
```

This method called paintAgent basically calculates the X and Y positions that it needs to start painting and the diameter of the Agent in order to use the fillOval method to fill the area specified of the color specified above with the setColor method.

World:

In the World class we've implemented and overloaded constructor, so when no margin is defined, the constructor assigns a default value to it. In this class we've also defined the following public methods:

- simulationStep(): For each Agent, if an agent has reached the target, it is assigned a new one to it. If the target is not reached, then the position is updated. In addition, in all simulation steps the collisions are managed.
- paintWorld(Graphics g): It paints all the agents inside the World delegating the rask to the PaintAgent(Graphics g) method.
- manageCollisions(): It checks if any agent is colliding with another one. If affirmative, by means of changing the target of the colliding agents, it solves the problem. That task is

mainly to randomPosInLine(Vec2D pos, Vec2D dir), which returns a new target that is both in the margin of the screen and in the line traced by the centers of the colliding agents.

```
public void manageCollisions(){
    for (int i=0; i<numAgents; ++i){
        for (int j = i+1; j<numAgents; ++j){
            if(agents[i].isColliding(agents[j])){
                // Get vector between centers of colliding agents
                Vec2D diffVec = new Vec2D(agents[i].getPosition());
                diffVec.subtract(agents[j].getPosition()); // This vector goes "from j to i"
                diffVec.normalize();

                // Set new target in the new direction (diffVec) "randomly" so that it does not fall off the window
                agents[i].setTarget(randomPosInLine(agents[i].getPosition(), diffVec)); // No need to flip the vector since the direction is OK
                agents[j].setTarget(randomPosInLine(agents[j].getPosition(), diffVec.scalarProdVec2D(-1))); // We need to flip the vector "from i to j"
            }
        }
    }
}
```

In particular, the manageCollisions() method uses the following private methods:

- randomPosInLine(Vec2D pos, Vec2D dir): We want that, given a direction and a position, returns a new Vec2D of the position of both the line of the margin (one of the four ones, depending on the position and direction in respect with the world) and in the direction given. Theoretically, that is achieved by means of 4 independent systems of equations (2 lineal equations that coincide in one point (positions)), choosing the minimum positive distance to travel k (the correct margin) and finally solving system of equations of the form of:

$$newPos_x = pos_x + k_{min} \cdot dir_x, \qquad newPos_y = pos_y + k_{min} \cdot dir_y$$

to get the minimum, the task is delegated to the min(double[] kn) method.

```
private Vec2D randomPosInLine(Vec2D pos, Vec2D dir){
    // We store all the four possible equations in an array
    double[] k = new double[4];
    k[0] = (margin-pos.getY())/dir.getY();
    k[1] = (margin-pos.getX())/dir.getX();
    k[2] = (height-margin-pos.getY())/dir.getY();
    k[3] = (width-margin-pos.getX())/dir.getX();
    // Take the index of the minimum positive value for k (the correct one depending on each case)
    int kMinIndex = min(k);

    // Return the position vector of the new target using the point-slope equation
    return new Vec2D(pos.getX()+k[kMinIndex]*dir.getX(), pos.getY()+k[kMinIndex]*dir.getY());
}
```

- min(double[] kn): From an array of doubles 'kn' it returns the index of the minimum positive value in the array.

```
private int min(double[] kn){
    int min = 0;
    for(int i=0; i<kn.length; i++){
        if((kn[i] > 0 && kn[min] < 0) || (kn[i] > 0 && kn[i] < kn[min])) { min = i; }
    }
    return min;
}
```

World GUI:

This class is basically in charge of the Graphical representation of the system created. We can already tell by the name as it includes GUI which means Graphical User Interface. In this class basically we define a private attribute of type World and then initialize it with some values. After that, the most relevant parts of the code to us are the simulate method, where we call the simulationsStep function from World class and then the paint method which we will explain below.

```
public void paint(Graphics g) {
    super.paint(g);
    W.paintWorld(g);
}
```

What this method does is to call the function paintWorld from the World class that calls the paintAgent method from the Agent class as many times as the number of Agents we want to paint. This is a clear example of delegating the tasks to other methods, as in the GUI we mainly call the methods of other classes and use them.

Lastly, we have the TestWorld class, that is the main of all these files, it's from where we call all the other classes, by creating a new WorldGUI in the main.

Alternative solutions:

We could have approached some of the attributes of World such as position, direction and target with the values of X and Y separately instead of creating Vec2D instances, but we thought it would be more practical to use the class as it was already available. This means we are promoting reuse and abstraction.

The methods used by manageCollisions (or others) have private visibility in order to promote encapsulation as the users don't need to know how manageCollisions or other functions work. To achieve our goals we are delegating the task to other smaller methods (which usually are private), which also helps with abstraction.

Conclusion:

In terms of practice the solution works perfectly and there should not be any bugs while running the code in any of the sections.

The task of creating new classes such as Agent and World was not too difficult as we had very clear instructions, and the same goes for the creation of the main methods used by these classes. However, when talking about our own implemented methods we had to think harder and longer to come to a solution.

The most difficult part of this first practice for us was the implementation of the manageCollisions method, as at first, we had a lot of bugs and it worked only sometimes, but when we thought of a mathematical solution, we cleared our doubts. The bad thing about this mathematical solution was that it took us a while to put it to practice as it was hard to implement it in code. In our approach to find the solution we used 4 systems of equations made of 4 equations each to find which one of the four margin each Agent colling should be directed to.