Personal information

Title - **Congestion SImulation**
Student's name - **Aayush Kucheria**
Student number - **781798**
Degree program - Bachelor of Science in Science and Technology, Majoring in **Data Science**
Year of studies - Year 1
Date - 27 April, 2020.

General description

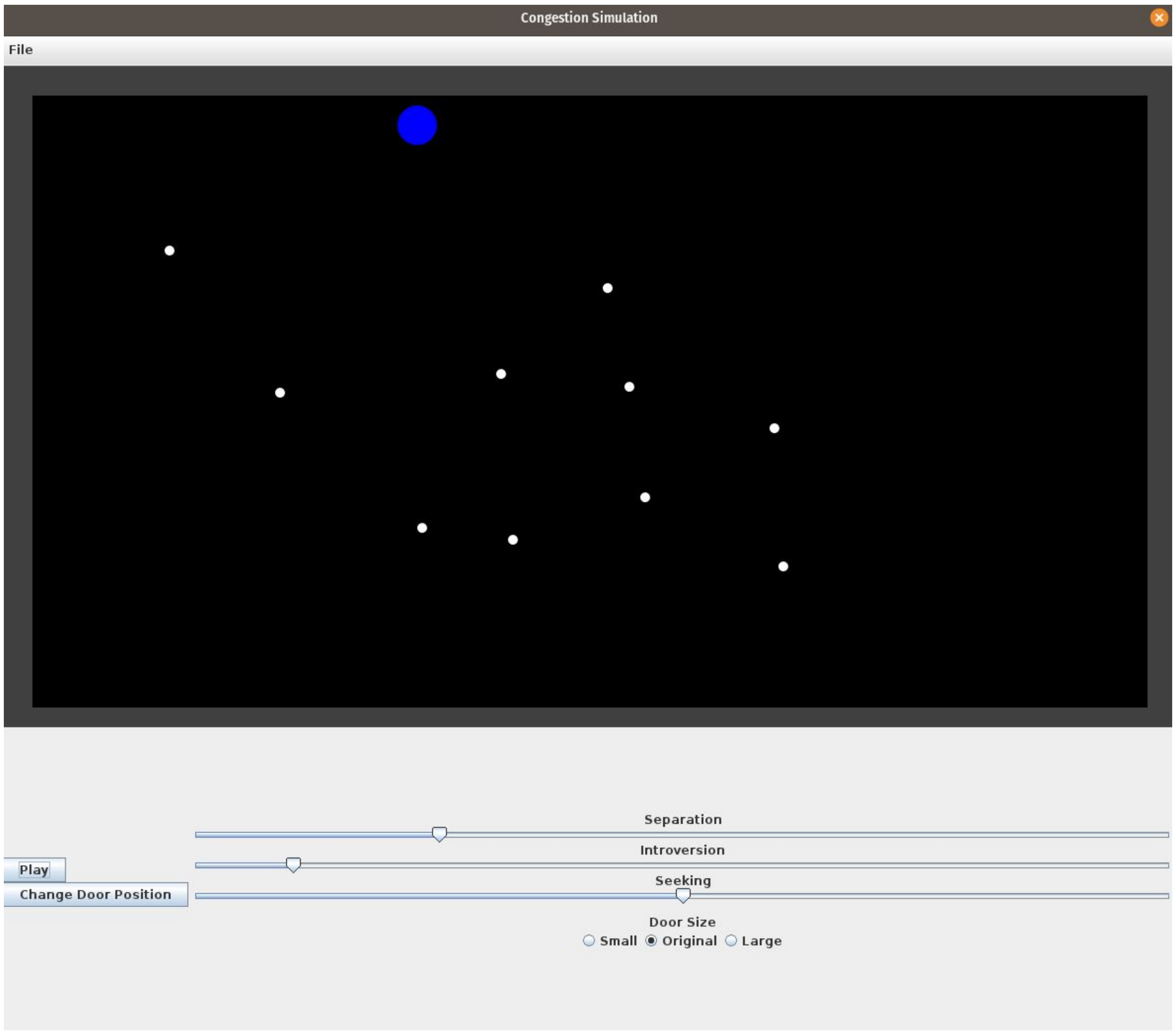
Aiming for difficulty level → Intermediate-Difficult
A congestion simulation that simulates the movement of a group to reach a given point. More specifically, a rush simulation where all individuals in a big room want to leave the room through Its small door as quickly as possible.

The people move based on certain conditions

- They slow down on seeing others close in front of them moving slower than themselves (braking).
- The door attracts the people (seeking).
- They avoid walls (avoidance).
- They avoid collisions where possible by adjusting their speed vector (separation).

Using these conditions as a base, I've to find a solution to the congestion problem where everyone gets out of the room as quickly as possible without collisions.
The simulation will have a GUI and a custom file format through which initial positions of the people can be read. The GUI will be customizable, i.e. the user can change the size or location of the doorway, and adjust the parameters of the motion conditions in real-time.

User interface



Separation

Introversion

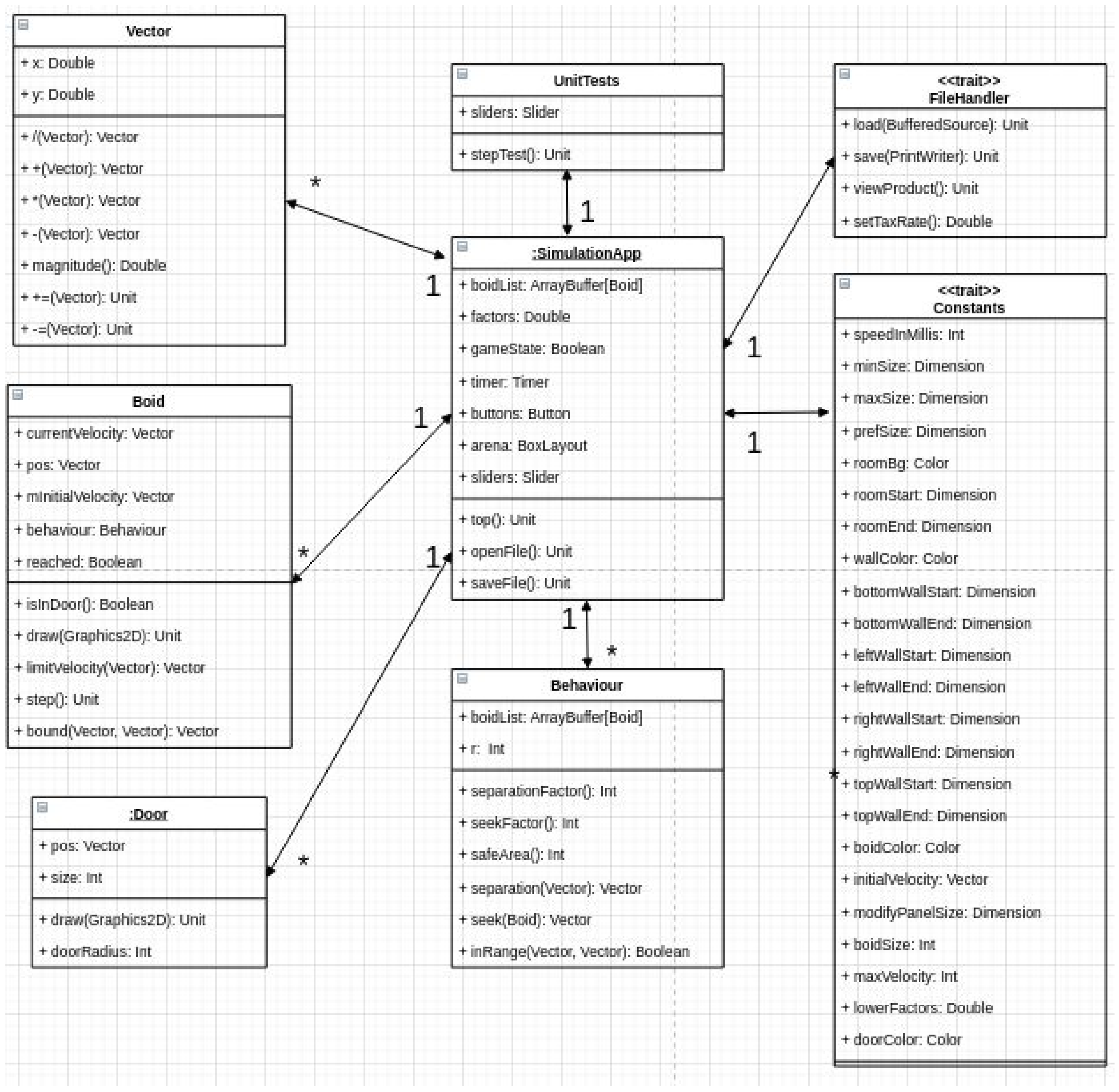
Seeking

Door Size

Small [✓ Original] Large

The Simulation starts with this screen. The user can either click on the arena to add new boids, or load an existing txt file with the specified format.
The user can also interact with the buttons and sliders below to pause the game, change door position, modify motion parameters, and change the door size.
The user can save the current state of the game to a txt file by clicking on the menu bar.

Program structure



The SimulationApp object is the main object of the program. It coordinates every aspect of the program. Vector, Boid, Door, and Behaviour handle the visibility and motion of the Boids, while FileHandler handles loading and saving of the simulation. Constants is a simple trait which stores all constant variables in one place for easy access. SimulationApp handles the GUI of the application.

- SimulationApp
  - Main object coordinating different aspects of the program.

    - Handles the GUI including drawing of arena, sliders and buttons, handling click events, and calling FileHandler for saving and loading files.
- Vector
    - A user defined datatype for 2d Vectors
    - Implements common arithmetic functions for the Vector Datatype
- Boid
    - A class for the boids in the arena.
    - Stores and updates their positions, velocities, and behaviour.
    - Checks for the state of the boid (whether reached door or not).
- Door
    - A singleton object for the Door,
    - Stores and updates the position, size, and radius of the door.
    - Draws the door onto the arena.
- Behaviour
    - A class to account for the separation, seeking, and avoidance behaviour of the Boids.
    - Calculates velocities for each behaviour accounting for their weight.
- FileHandler
    - A trait to save the state of the simulation to a txt file or load an existing simulation from a txt file.
- Constants
    - A trait which stores all the constants in the program for easy access.
- UnitTests
    - Testing of several aspects of the program.

The current class structure was decided upon because each class only has close to 1 function. This makes it simpler to write and read.
The GUI could've been a separate class, so SimulationApp would only coordinate all the classes, while the GUI could be a separate function handled by another class.

Algorithms

The movement of the boids follows the Simple Vehicle Model Approach by Craig Reynolds.

Summary - The aim is to seek the door while avoiding colliding with other Boids or going through walls. Find the shortest path to the door and try to get there as fast as possible. Each Boid avoids collision by imagining a circle around itself and trying to keep that circle empty.

The physics of this simple vehicle model is based on Forward Euler Integration. At each simulation step, behaviourally determined steering forces are applied to the vehicle's point mass. This produces an acceleration equal to the steering force divided by the vehicle's mass. That acceleration is added to the old velocity to produce a new velocity, which is then truncated by max_speed. Finally, the velocity is added to the old position to get the new position.

$$steering\ force = truncate(steering\ direction,\ max\ force)$$
$$acceleration = steering\ force\ /\ mass$$
$$velocity = truncate(velocity + acceleration,\ max\ speed)$$
$$position = position + velocity$$

The simple vehicle model maintains its velocity-aligned local space by incremental adjustment from the previous time step.
The three behaviour factors for the motion are Target Seeking and Obstacle Avoidance

Seek (or pursuit of a static target) acts to steer the character towards a specified position in global space. This behaviour adjusts the character so that it's velocity is radially aligned towards the target. The "desired velocity" is a vector in the direction from the character to the target.

$$desired\ velocity = normalize(position - target) * max\ speed$$
Return $desired\ velocity * seekFactor$

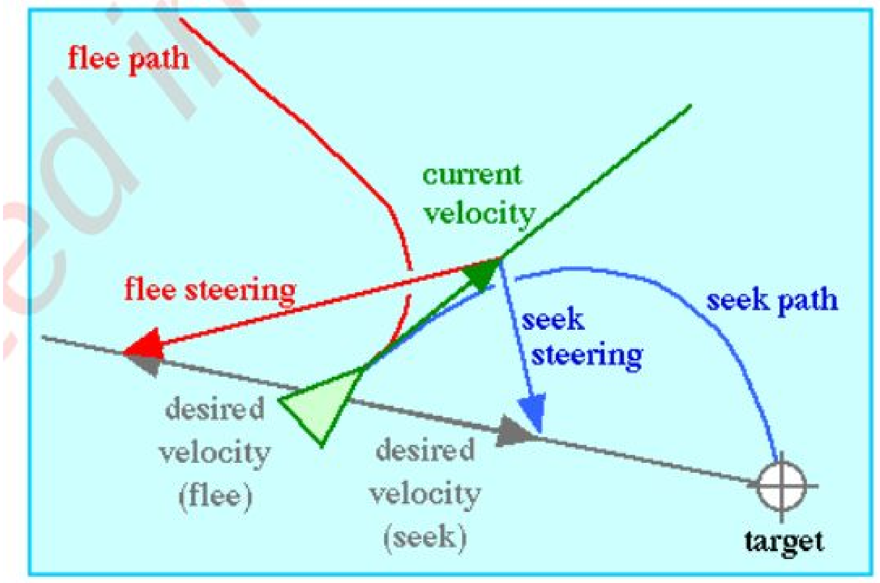


Figure 3: **seek** and **flee**

Obstacle Avoidance behavior gives a character the ability to maneuver in a cluttered environment by dodging around obstacles. The implementation of obstacle avoidance behavior described here will make a simplifying assumption that both the character and obstacle can be reasonably approximated as spheres, although the basic concept can be easily extend to more precise shape models. The goal of the behavior is to keep an imaginary sphere of free space around the character. The obstacle avoidance behavior considers each obstacle in turn using a spatial portioning scheme to cull out distance obstacles and determines if they intersect with the sphere.

For all obstacles inside the sphere of the boid →
$change\ in\ Pos = (current\ Position - obstacle\ Position)$
$totalForce \mathrel{+}= normalize(change\ in\ Pos)\ /\ mass$

After summing up totalForce for all closeby obstacles
$return\ totalForce * avoidanceFactor$

Data structures

The program uses a user-defined data structure for the Boids - to store their positions, velocities, and other properties. The program also uses a user defined type Vector for storing mutable positions and velocities of the Boids.

Boid
Boid stores the positions, velocities, and other properties for each boid in the program. This structure helps collect all the relevant information for a boid and stores it together. This helps divide the program in a meaningful way, and provides structure to the program as a whole. Along with the mutable relevant information, it also defines a few methods to calculate future positions, bounds, status, and velocity limits of the boids.

Vector
It takes in two parameters of type double x and y.
It overrides the normal arithmetic operations such as +, -, /, and * as follows →

Addition → $def\ +(other : Vector) = new\ Vector(this.x + other.x,\ this.y + other.y)$
Subtraction → $def\ -(other : Vector) = new\ Vector(this.x - other.x,\ this.y - other.y)$
Division → $def\ /(other : Vector) = new\ Vector(this.x\ /\ other.x,\ this.y\ /\ other.y)$
Multiplication → $def\ *(other : Vector) = new\ Vector(this.x * other.x,\ this.y * other.y)$

It also implements a few additional methods such as “Magnitude” and “Normalize” which work as follows →
Magnitude → $def\ magnitude\ =\ math.sqrt(math.pow(this.x,\ 2)\ +\ math.pow(this.y,\ 2))$
Normalie → $def\ normalize\ =\ this/this.magnitude$

The reason to choose this format for the datatype was to make calculations in a 2D world easier. This datatype is also easily scalable to n-dimensions.
Another option would have been to have different 1-dimensional variables for x and y positions of the boids, which would be cumbersome. This format implemented collects n-dimensions and makes it easier + intuitive to work with them.

Others
All the other data types used are existing scala collections. The program uses an **ArrayBuffer[Boid]** to store the list of boids currently in the simulation. The reason to choose this collection was because it’s mutable and efficient.

Files and Internet access

The program has a unique file structure to save the state of the simulation. It’s stored as a **.txt** file.
Here is a template - (the underlined words store the data)

congestion simulation
#factors
separation=<u>50</u>
seek=<u>100</u>
safearea=<u>20</u>
#door
doorpos=<u>936.0,173.0</u>
size=<u>40</u>
#boid
poslist
<u>663.8,262.9</u>
<u>716.5,882.1</u>
.
.
.
<u>313.5,161.5</u>
#/#

This file structure categories different kinds of data, is human-readable, and is scalable.
I didn't use XML or Json because the program was comparatively simple, and the job could be done with a simple defined format.

Testing

The most important aspects to test are

- If the simulation crashes for boundary conditions.
- If the boids move in an optimal way.
- If the file reading and writing work correctly, while handling exceptions.
- If the calculations work correctly.
- If the data gets shared and updated through all the classes.

Optimality of the motion of the boids, file reading/writing, and the graphical aspect of the program were tested **manually** through observation, print statements, and dry running the code. The boundary conditions, real-time sharing, and calculations were tested through **unit testing**.

The program passes boundary conditions, reads and writes files correctly, and calculates in an expected way. It falls short on the optimality of the motion of the boids.

The planned testing process was to use unit testing for all the features but I later figured out that it was impractical. Most testing can be done through simple observation.

Known bugs and missing features

- Can't resize arena
    - Since I've implemented the room and walls through hardcoded pixel values, it's hard to add a feature to resize the arena. One possible solution to this was using BorderPanel with the room in the center surrounded by walls. I did try this approach but it didn't communicate well with other parts of my program, and due to time constraint I couldn't experiment more.
- Motion of Boids not optimal
    - Even though I implemented the formulas for each type of behaviour, the motion of the boids is not optimal. When separationFactor and safeAre of the boids are maxed out, they don't seek the door but instead stick to one of the walls. This bug can be overcome by tuning with the parameters like mass, and lowering of

factors. I did try a few values, but generally don’t know how to approach this problem.

- Multithreading Not Implemented
    - I experimented implementing multithreading in my program to separate the drawing of the gui and calculation of vectors into different threads. But since the scale of the program is small, and it’s relatively fast, I decided on not implementing them after all.
- Sometimes program fails to recognize mouse clicks
    - This might not be a bug and just a problem with eclipse. But sometimes when adding new boids by clicking on the room, the program fails to notice a few random clicks. Frankly, I don’t know why this is happening. I have two simple lines for this functionality - listen to mouse clicks and do something when mouse clicked. Both of them are implemented by Scala Swing, and it’s difficult to identify where the problem is.

3 best sides and 3 weaknesses

Best Sides

- File Handling
  - The File Handling of my program runs perfectly. It outputs files in a defined matter, but more than that - it handles incorrect files in a very efficient way. Leaves out the incorrect part and only inputs the correct parameters and values.
- Program Design
  - The program structure is clean and scalable. I can implement additional functionalities like different colors, more factors, additional dimensions, with just a few modifications to my code.
- Motion of the Boids
  - Even though it's not perfect and can be improved, I'm overall pretty happy with how the boids behave in different conditions. They seem to get the job done (get to the door) most of the time!

Weaknesses

- Motion of the Boids
  - This is a strength as well as a weakness. Though I implemented the motion to quite an extent, it could have been a lot better with more tweaking. For example, the values of the constant factors could be tweaked and the separation functionality in my code could be divided into separation, obstacle avoidance, and braking. Right now, I've mixed them all up into one behaviour.
- GUI Structure
  - The position of my arena is based on brute force. That led me to quite some trouble with changing the size of the arena (which I couldn't implement), and changing the size of the window (which increases the size of the wall). I did try quite a lot to change the brute force method to percentages, BorderLayout, or GridLayout, but couldn't form a suitable one because of the time constraint as I noticed this problem near the end of the timeline.

Deviations from the plan, realized process and schedule

Dates	Planned	Reality
14 Feb - 27 Feb	Read research paper and plan basic platform of	Read research paper and coded basic GUI

	program	
28 Feb - 12 March	Implement and test algorithm. Work on simple GUI	Refined GUI, trying different approaches.
13 March - 26 March	Finish implementing algorithm and GUI. Test	Didn't work on the project due to exams
27 March - 9 April	Try implementing other algorithms and look for Machine Learning usecases	Implemented GUI and started working on the algorithm
10 April - 27 April	Implement Machine Learning Methods and finalize project	Implemented Algorithm, tested, refined, and finalized the project.

The plan was an optimal view of the whole process, but heavy course load, exams, and other activities prevented me from following the optimal plan. Most of my program was written, tested, and rewritten in the last 14 days.

Learnings

- Plan for deviations in the initial plan. Take into account factors such as other courses and miscellaneous things I might end up doing.
- The initial plan was quite good, and I would've loved implementing the ideas I came up with but couldn't implement. Maybe in the summer!
- Learned that having specific goals during coding helps make it more productive.
- Learned that ending a coding session on a high note makes me more attracted to the task in the future, while ending on a low note makes me procrastinate further.

Final evaluation

Summary and Evaluation

The overall program achieves the "difficult" goals, while being a clean and scalable code structure. The aspects surrounding the boids, their behaviour and their movements were well designed. Used a bit of brute force in the GUI to construct the arena.
"Summary" and self-assessment, where you can repeat some of the above things.

As mentioned above, the shortcomings of my program are that the motion of the boids could be improved by tuning with the constant parameters, while the GUI could be constructed in a more structured manner probably using BorderPanel.

- The program can be improved further by allowing the arena to have variable size, which should be achievable if the GUI structure is improved.
- The movement of the boids can be improved by subdividing the roles such as separation, obstacle avoidance, and braking into separate behaviours and tuning them further. Right now, these three behaviours are combined into one behaviour in my program.
- The class structure can be improved by separating the GUI aspect from the main SimulationApp object.
- With a few modifications, the program might be able to have different values for each behavioural aspect. Right now, all of them are confined to the same value at a given point of time.

What I would do differently →

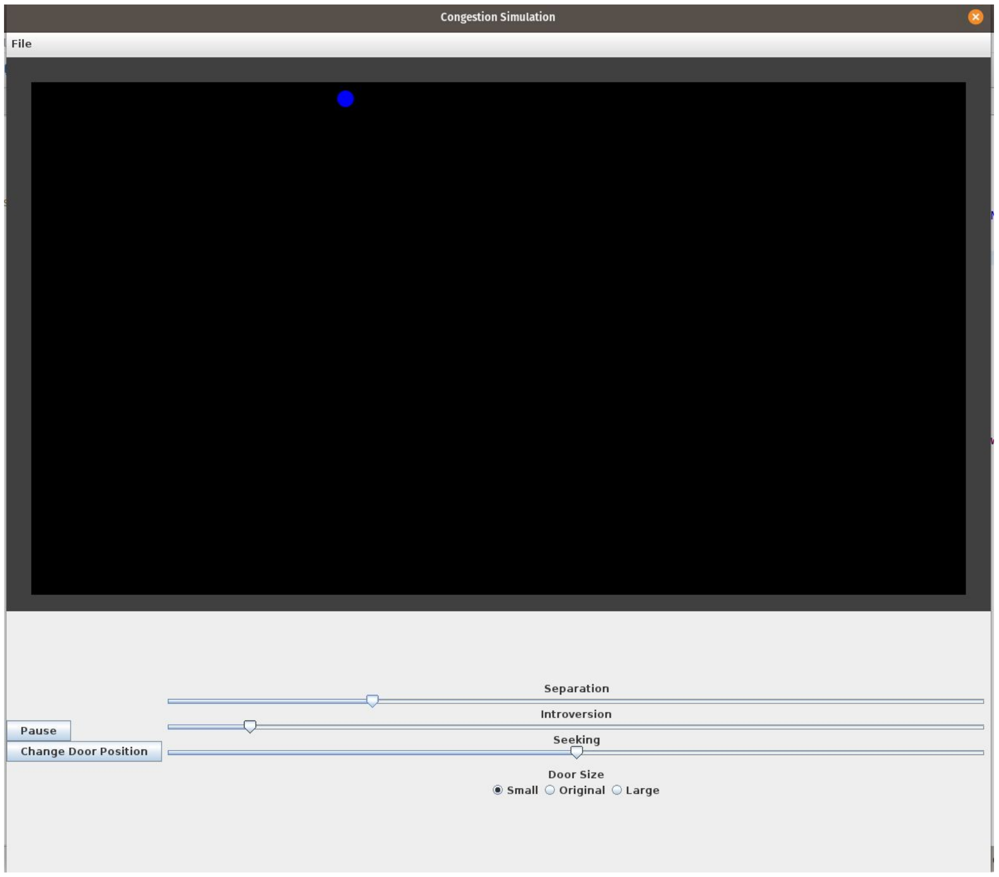
- Balance the work more evenly through the timeline by setting extremely specific goals.
- I wasted weeks on getting the right GUI. Next time, I would create a simple GUI structure quickly and spend more time tinkering with the motion of the boids. The ratio of work in GUI and motion was the opposite of what I wanted it to be.

References

- Steering Behaviour for Autonomous Characters
- Stack Overflow
- Stack Exchange
- OTFried GUI Examples
- Alvin Alexander Scala Resource
- tetrix in Scala — swing
- scala.swing
- Swing Examples
- A Concise Introduction to Scala GUIs
- Mark Lewis

Appendixes

Initial State



Separation

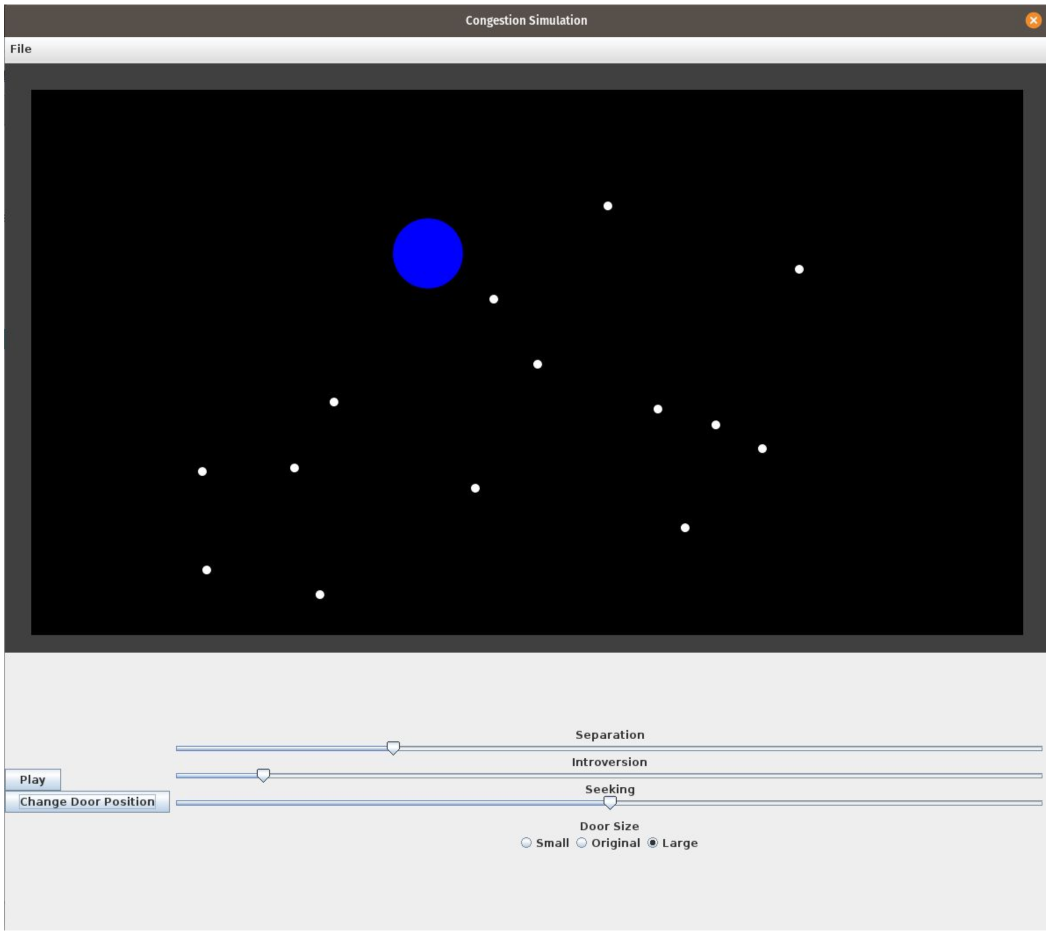
Introversion

Seeking

Door Size

[✓ Small] Original Large

Door Position and Size Changed + Boids heading to the Door



Separation

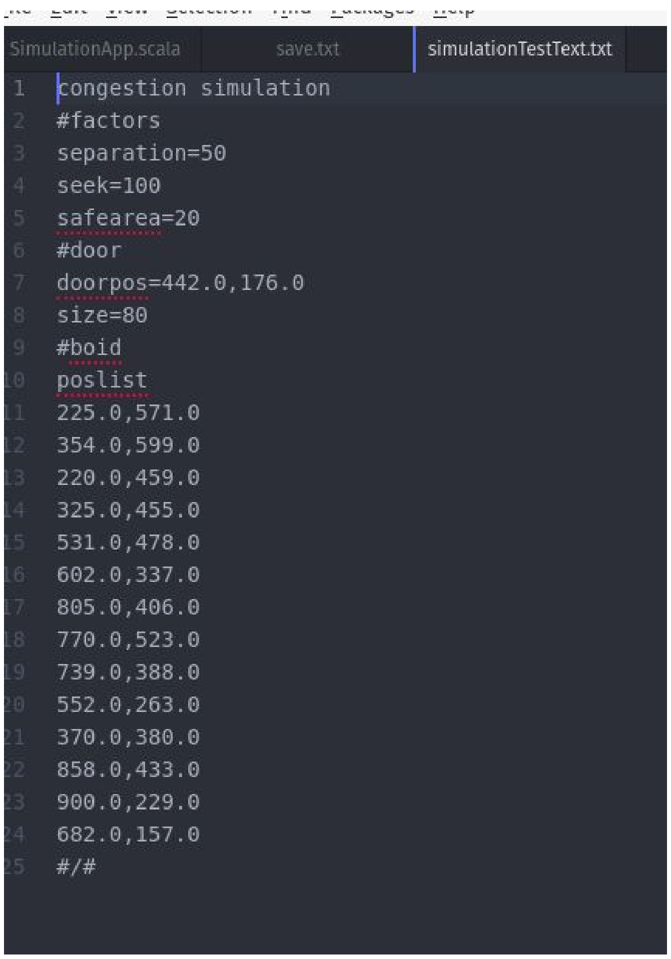
Introversion

Seeking

Door Size

Small Original [✓ Large]

The simulation showed in the previous image in text form



```
congestion simulation
#factors
separation=50
seek=100
safearea=20
#door
doorpos=442.0,176.0
size=80
#boid
poslist
225.0,571.0
354.0,599.0
220.0,459.0
325.0,455.0
531.0,478.0
602.0,337.0
805.0,406.0
770.0,523.0
739.0,388.0
552.0,263.0
370.0,380.0
858.0,433.0
900.0,229.0
682.0,157.0
#/#
```