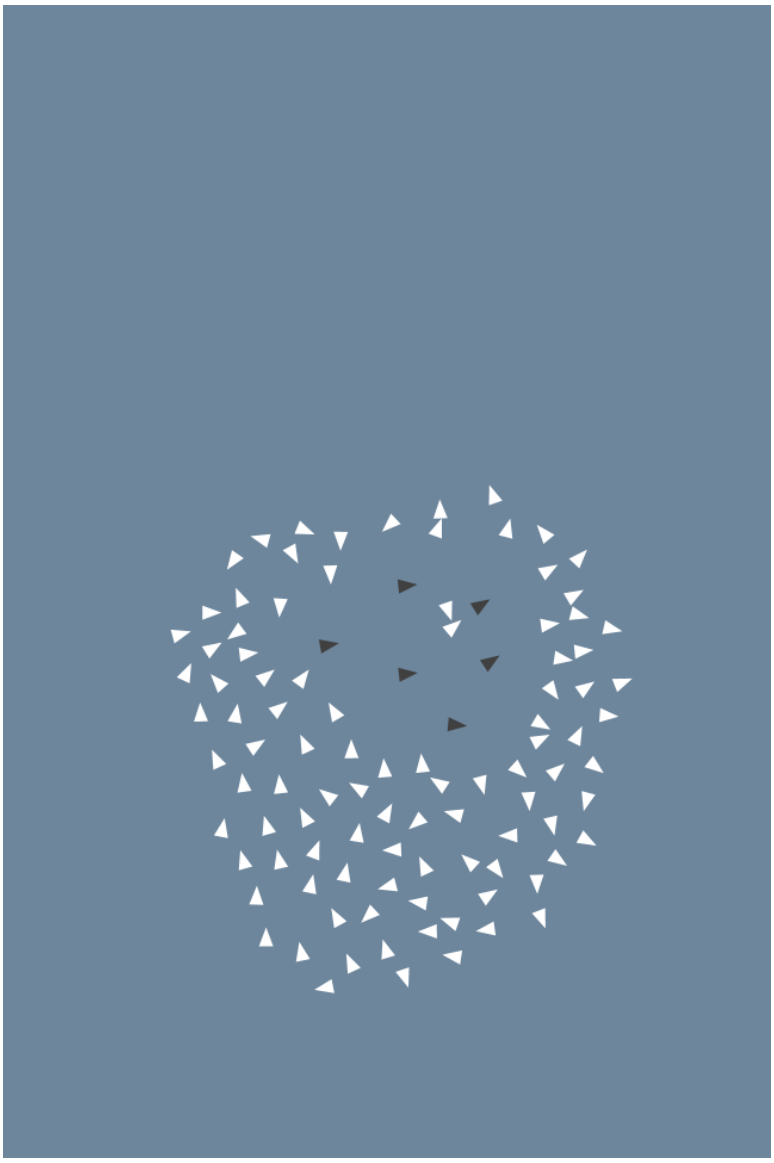


Follow the Leader

A PROJECT DOCUMENT

Quan Trung Nguyen | 793663 | Data Science | 2020-2025 | 25th April 2020

Contents

General Description

A 'Simon says' (follow the leader) simulation. The intention is that in a flock there is a leader who moves randomly, and others follow this individual without interfering his path or colliding with each other. The program visualizes the movement of the individuals by combining two different features appropriately:

- The leader aims for chosen places randomly (Seeking).
- The leader wanders aimlessly without doing sharp turns (Wandering).

The others in the flock can be implemented as follows:

- The individuals avoid excessive crowding or want to keep enough space around them (Separation).
- The individuals try to reach a certain point, slowing down as they approach this point. The target point of the movements can be somewhat behind the leader so that the members of the flock do not accidentally collide with the leader and make him more distinct (Arrival).
- Additionally, the individuals try to dodge the leader if they get on his way.

Based on Craig Reynolds' article" Steering behaviors for autonomous characters" http://www.red3d.com/cwr/steer/gdc99/.

In addition to the stated requirements, the program also allows for multiple boids in action and the force-vectors of the boids drawn. Not only do the program can read from files, it can also write to them. Overall, the author reckons that the implementation of the program is of a high-level difficulty.

User Interface

When the user first opens the program, there will be 2 dialogs:

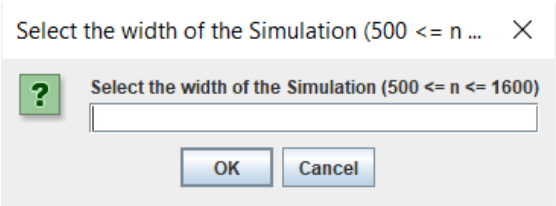


Figure 1. Width-selection dialog

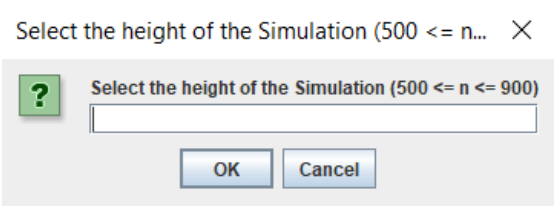


Figure 2. Height-selection dialog

Both of them ask the user to input the dimensions of the program. After that, the main interface of the program will be displayed:

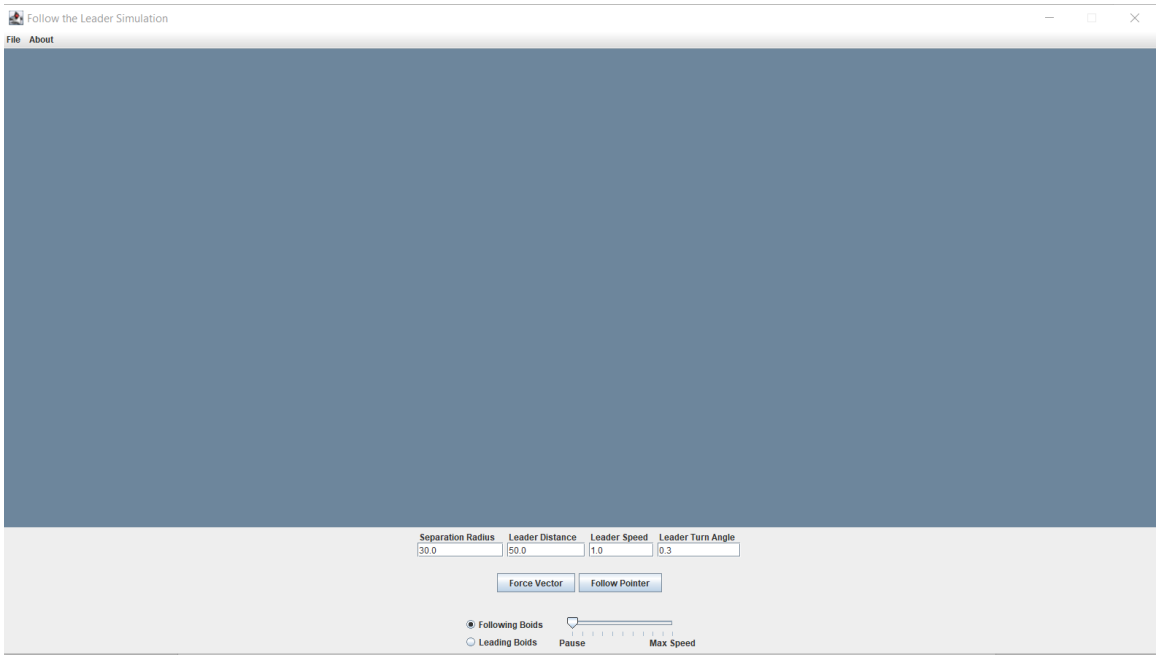


Figure 3. The main interface of the program

The biggest panel in the middle of the program is where the simulation of the Boids happens. The user can create new Boids by clicking on it. Below the panel is all the available buttons and text fields for the user to interact with the simulation:

- Separation Radius: the distance between the Following Boids when simulated.
- Leader Distance: how far the Following Boids should be behind the Leading Boids.

- Leader Speed: the speed of the Leading Boids
- Leader Turn Angle: how wide the Leading Boids should turn (Note: only work when the Boids are not following the pointer)
- Force Vector: add lines that stem from the Boids which represent the steering forces applied to them.
- Follow Pointer: make the Leading Boids follow the cursor.
- The two radio buttons named "Following Boids" and "Leading Boids": Change the kind of Boid the user creates when they click on the map.
- Slider: Change the speed of the simulation. The user can pause the speed of the simulation using this.

Above the main panel, there is a menu bar where users can open, save simulations as well as read an interesting text. The open simulation dialog interface resemble the Windows Explorer interface significantly. The save simulation counterpart, on the other hand, only has a text field for the user to input the name of the file. After finishing saving the program, the user can locate his or her saved file in the directory of the program.

Note: Saving simulation only means saving the current simulation and its boids, not the modes and parameters.

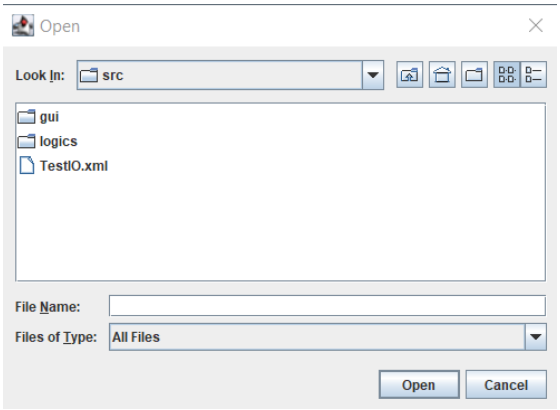


Figure 4. Open simulation Dialog

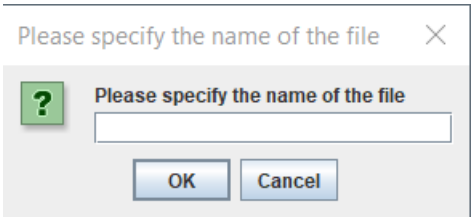
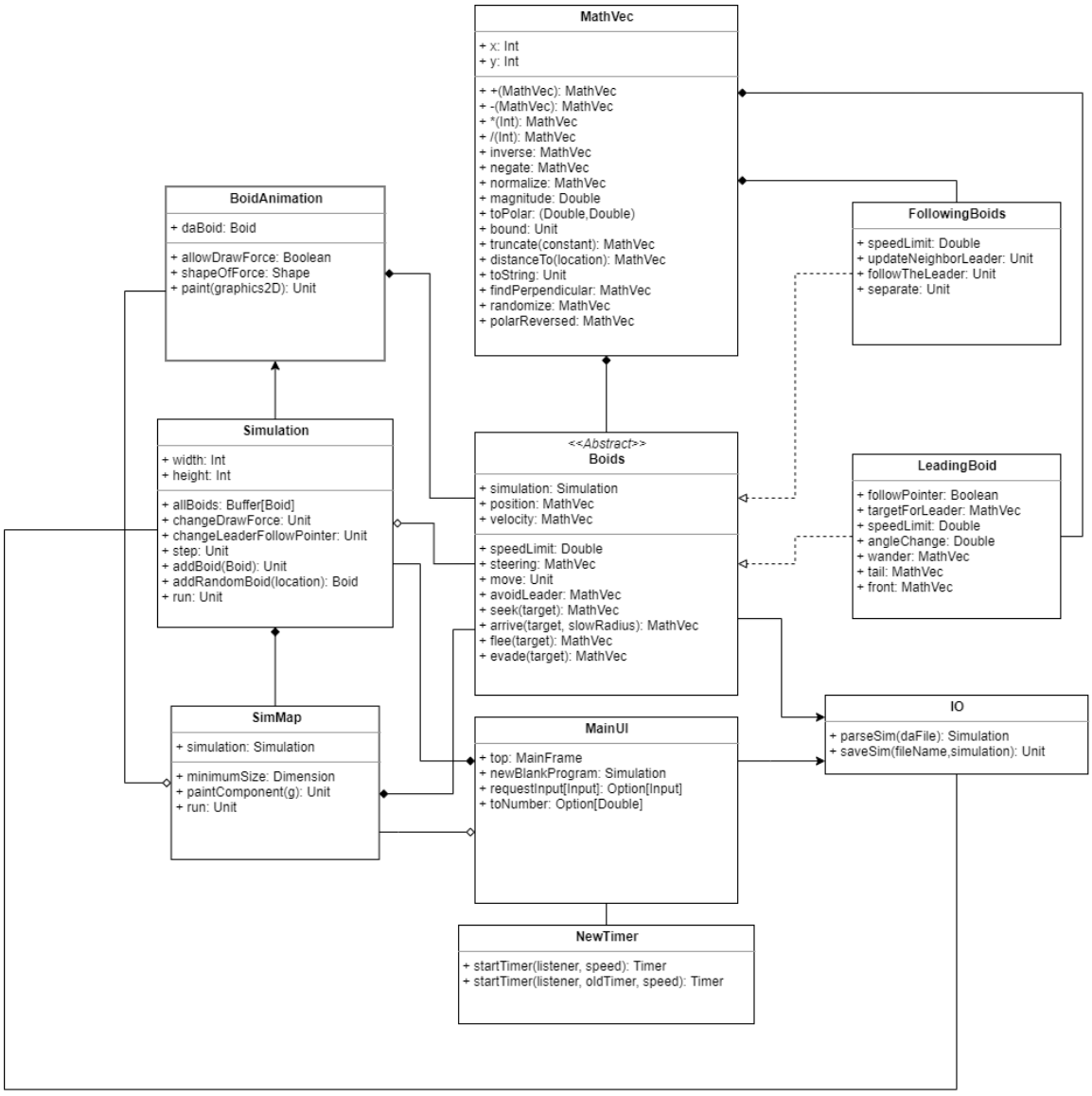


Figure 5. Save simulation dialog

Program Structure

The program is divided into 2 main parts: **GUI** and **logics**. The former consists of MainUI, BoidAnimation, SimMap classes, all of which output the interface of the program. The latter is composed of the classes: Simulation, Boids, MathVec, and IO, which dictate how the program works. The relationship between the classes are described as following:



- MathVec: Responsible for carrying out vector operations.

- Boids: The abstract class from which the 2 classes Following Boids and Leading Boids inherit.
- Following Boids: The class that implements the distinct methods of the mentioned Boids. Inside, the most important *separate* and *follow the leader* behaviors are implemented.
- Leading Boids: The class where the distinct behavior of Leading Boids is implemented: *wander.*
- IO: In charge of reading and writing simulations from and to files.
- BoidAnimation: Implement boids and their steering forces' graphical representation.
- SimMap: The area of simulation. When the program runs, a thread of this class is created, drawing the area on which the simulation all the boids within it operate.
- Simulation: The **logics** of the simulation itself. It contains all the constants used in the program. All the functionalities that concern with adding Boids use the methods of this class. The most important of which is the *step()* method which makes the Boids calculate their next position – a big part of how the program works. The class is initialized as a thread in the program.
- MainUI: The central part of the simulation. It contains most of the components the users interact with, as well as timers that render the program in motion. Events are also handled by this object.
- Timer: A secondary object used to implement the overloaded methods. All of which are responsible for triggering timers which set the simulation in motion.

In the planning phase, there are many decisions that had to be made regarding the structure of the program. The most important of which was whether to have 2 classes the Following Boids and Leading Boids inheriting a Boid abstract class, or just one all-encompassing Boids class. After careful deliberation, the author chose to follow the former approach, as those 2 classes are too divergent in its operations to be in the same class, but also share too many common methods and properties to be completely on their own.

Another one was about how to distribute threads, with the option of creating multiple threads, one for each boid and its animation. However, it was determined that this method may result in unwanted complications, such as the limit of threads, and bring about minimal benefits, thus not worth pursuing. SimMap class's existence was also put into question. Should they be integrated into the MainUI or constitute a separate class? This turned out to be a simple decision because eliminate the panel would deny the **logics** of the program the ability to thread. Aside from those, the use of the NewTimer

object was not thought of in the beginning but only implemented as a result of needs arose in the coding process.

Algorithms

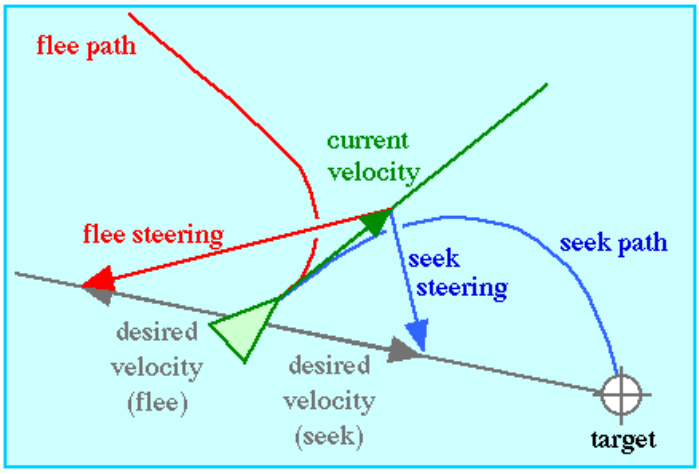
The Boids are based on a point mass approximation, defined by position and velocity. At each simulation step, behaviorally determined steering forces (limited by max_force) are applied to the boid's point mass. That acceleration is added to the old velocity to produce a new velocity, which is then truncated by max_speed. Finally, the velocity is added to the old position:

```
steering_force = truncate (steering_direction, max_force)

velocity = truncate (velocity + steering_force, max_speed)

position = position + velocity
```

There are 5 crucial Boids' behaviors in a flocking simulation:

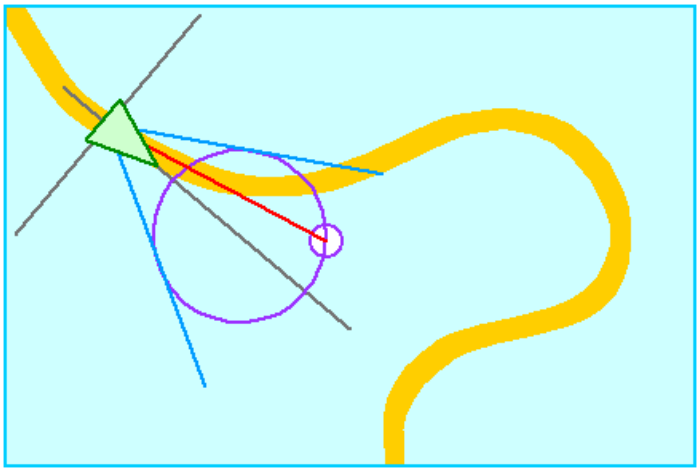
1. Seek and Flee: acts to steer the character towards or away a specified position in global space. This behavior adjusts the character so that its velocity is radially aligned towards the target or away from it:

```
desired_velocity = normalize (position - target) * max_speed

steering = desired_velocity - velocity
```



2. Wander: is a type of random steering. The boid retain steering direction state and make small random displacements to it each frame. The steering force is constrained to the surface of a sphere located slightly ahead of the character. The sphere's radius determines the maximum wandering "strength" and the magnitude of the random displacement determines the wander "rate."

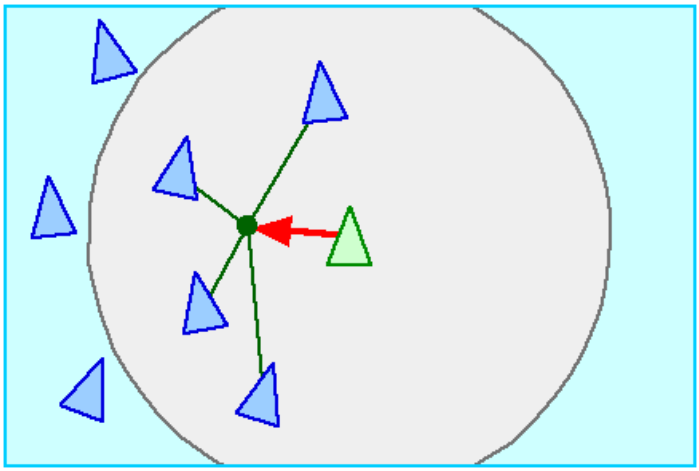
```
circle_center = normalize (velocity) * circle_distance

random_direction = random_vector * circle_radius

steering = circle_center + random_direction
```



3. Separation: gives a character the ability to maintain a certain separation distance from others nearby. This can be used to prevent characters from crowding together. To compute steering for separation, first a search is made to find other characters within the specified neighborhood. For each nearby character, a repulsive force is computed by subtracting the positions of our character and the nearby character, normalizing, and then applying a 1/r weighting.

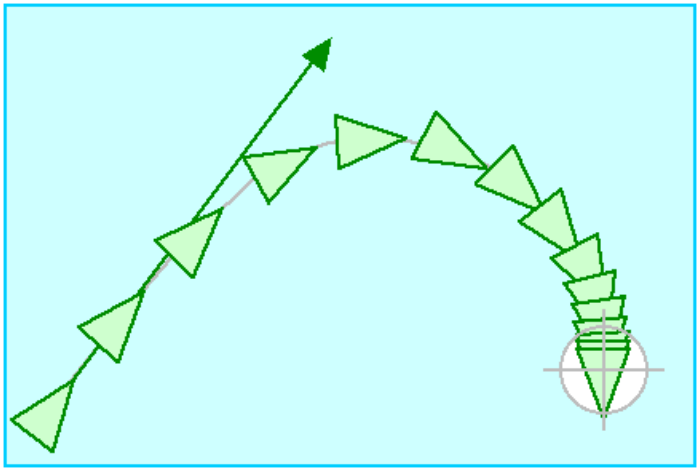
```
For (i <- allBoids)

{

if (i != thisBoid)

{

if (thisBoid.distanceFrom(i) < neighbor.radius)

{

v.x += boid.velocity.x;
```

```
v.y += boid.velocity.y;

neighborCount++;

}

}

}

v.x += i.x - thisBoid.x;

v.y += i.y - thisBoid.y;

v.x *= -1;

v.y *= -1;
```



4. Arrival: behavior is identical to seek while the character is far from its target. But instead of moving through the target at full speed, this behavior causes the character to slow down as it approaches the target, eventually slowing to a stop coincident with the target.

```
target_offset = target - position

distance = length (target_offset)

if (distance < slow_radius)

desired_velocity = target_offset * max_speed *
(distance/slow_radius)

steering = desired_velocity - velocity
```



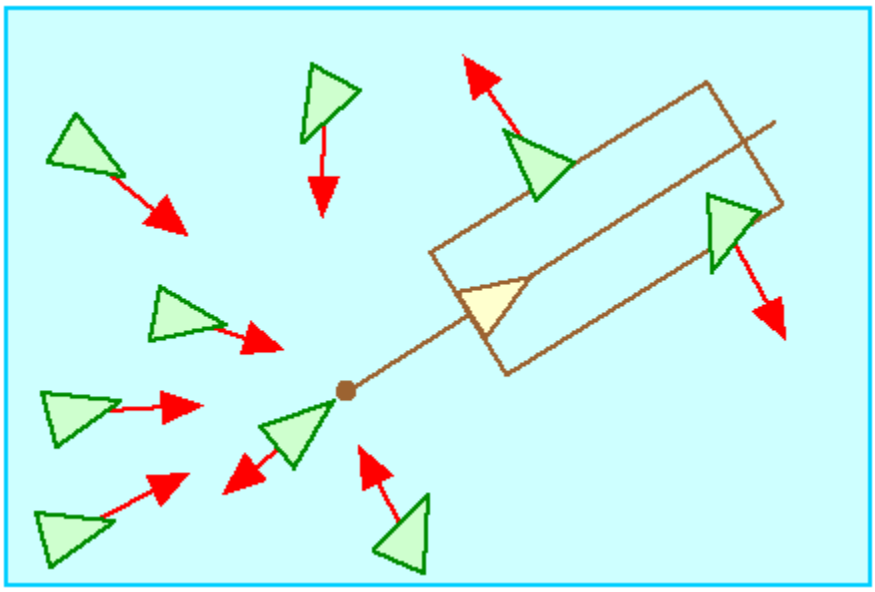
5. Leader Following: one or more character to follow another moving character designated as the leader. Generally, the followers want to stay near the leader, without crowding the leader, and taking care to stay out of the leader's way (in case they happen to find themselves in front of the leader). In the program, the leader following is achieved by having the followers (Following Boids) to *arrive* at a location behind the leader (the tail):

```
tail = leader_position - normalize(leader_ velocity) * constant

steering = arrive(tail)
```

And, leader avoiding is done by setting up a point in front of the leading boid and making every other Boids *flee* from that:

```
front = leader_position + normalize(leader_velocity)

if (this_distance(front) <= 50) flee(front)
```

To further avoid collisions, the program also includes a mechanism that make all boids flee the leader when it is too close to them. Overall, this behavior heavily relies on the implementation of *arrival* and *flee.*



In this particular program, the "Leading Boids" will employ 3 of them: *wandering* and *arrival* and *flee*. The "Following Boids" will utilize *separation* and *follow the leader.* Most of them are implemented as mentioned, except that the *seek* behavior of the Leading Boids will aim at the position of the cursor. This mode can be turned on by toggling the "follow pointer" button. For the simulation to work, the program uses a Timer with a minuscule delay. After each delay, the *step()* method is called, each Boid's position is updated, its animation is drawn, putting the program in motion.

With regard to functionalities, their inner-workings are relatively straightforward. The slider controls the simulation by manipulating the Timer, reducing the delay time when the user turns it up, and vice versa. When the speed level is dropped to 0, the method *Timer.stop()* is called, pausing the processing of the simulation. In addition, the "Force Vector" button draws the steering forces of the boids just by painting them as straight lines with one vertex fixed on the position of the Boids. Finally, the Text Fields can modify the simulation by changing the constants listed inside the Simulation classes, consequently changing the Boids' behaviors.

Data Structures

The program has a simplistic use of data structures. All the Boids are stored in Buffers, a mutable data structure, for flexible access, element-generating and removal.

Files and Internet Access

For loading and saving data into files, the program uses the XML format. This divergent from the plan which uses JSON was made because of practical reasons. In short, it is because of the much better compatibility of Scala with XML. The structure of a typical file written and readable by the program is as follow:

```
<simulation>
  <width> { simulation.width } </width>
  <height> { simulation.height } </height>
  <boid>
    <type> FollowingBoid </type>
    <position>
          <x> { x.position.x } </x>
          <y> { x.position.y } </y>
    </position>
    <velocity>
          <x> { x.velocity.x } </x>
          <y> { x.velocity.y } </y>
    </velocity>
```

```
</boid>
<boid>
  <type> LeadingBoid </type>
  <position>
        <x> { position.x } </x>
        <y> { position.y } </y>
  </position>
  <velocity>
        <x> { velocity.x } </x>
        <y> { velocity.y } </y>
  </velocity>
</boid>
</simulation>
```

Testing

1. Manual testing:

Due to the graphical nature of the program, manual tests are, in most of the cases, the most optimal approach. We can check if the basic parts of the simulation are working correctly just by carefully studying how the boids travel and interact with each other:

- The Boids should avoid the leaders quickly and responsively.
- The Leader Boids, in its default mode, should steer naturally.
- The Following Boids should maintain the correct distance from each other.

Considering functionalities, the number of ways that the program can be tested increases significantly:

Scenario 1: Every functionality work in normal conditions.

- Changes in parameters bring about expected results
- Speed can be changed easily
- The simulation can be paused anytime
- Saving and loading work smoothly
  Etc.

Scenario 2: When inputted boundary values, the program throws errors.

- Text fields, when given the wrong type of parameters, are expected to show error messages. The same thing applies to saving and opening simulations dialogs.
- When the simulation is paused, every other function still works correctly.

2. Unit Testing:

As manual testing already constitutes much of the testing process, the use of unit tests was not to a great extent. In particular, only 3 unit-tests that cover the most crucial aspects of the program were written:

- testTruncate: tests the *truncate* function in the MathVec class. The function, when called by a vector, should be able to return a new vector with the magnitude given by the function but still has the same direction as the old vector.
- testSeek: tests the *seek()* function in the Boids class. The function, when called by a boid, should return a vector aimed at the *target* parameter of the function, having taken into account the current velocity of the Boid.
- testUpdateNeighborLeader: tests the *updateNeighborLeader* function in the FollowingBoids class. The function, when called by a Following Boid, should return a buffer of the nearest Following Boids by a certain radius as well as its nearest leader.

Known Bugs and Missing Features

After the process of testing, it was determined that the program has 3 bugs:

1. This is the most significant error: opening a new simulation (whether from I/O or blank simulation) does not work for those which have the same size as the

current one. In detail, when the user tries to open a new simulation that has the same size as the current one, the program will be turned into a gray, inactive panel. The cause of this seems to stem from the conflict of size between 2 instances of the program. If the author was given more time, the *top()* method of the SimpleSwingScala will be more thoroughly investigated, prompting solutions. However, for now, one simple fix can already be carried out. We can make the program automatically make tiny changes to the size of the simulation. However, whether such practice is ethical is up for debate. Hence, the author refrained from implementing such a solution.

2. When the program is paused, it is not possible to create more than one boid at the same exact location. This is thought to be related to the nature of listener and reaction mechanism of Scala Swing. An approach to this problem might be replacing the reactions by another function, one similar to the method *findMouse().*
3. In some rare cases, the user may not be able to add Boids. This problem happens on such limited occasions that the author has not been able to trace its roots.

3 Best Sides and 3 Weaknesses

Strengths:

1. The project accomplished more than difficult requirements.
2. The source code was written with thought and intent, carefully organized and commented, and thus, easy to read and understand.
3. The author implemented creative ways to solve the problem. For example, the *avoidLeader* method deviated somewhat from the original papers by Craig Reynolds, using one dot and its radius to simulate the region in front of the Boid.

Weaknesses:

1. The program has a fatal flaw which is listed at the beginning of the previous section.
2. The program does not have a "wall" mechanism that, when it is turned on, prevents the Boids from going outside the boundaries of the simulation. Attempts

have been made to address this but resulted in unnatural behaviors on the Boids' parts, and thus, was withdrawn.

3. Twitching of the Boids is still a prevalent phenomenon, especially among the Following Boids. This is determined as a problem with parameters, not with the source code, and hence, it can be addressed with time.

Deviations from The Plan, Realized Process and Schedule

The process of implementing the program was radically different from the intended plan. The **GUI** was the first component introduced, instead of the **logics**. The time it took to create the user interface is significantly longer, compared to that of the **logics** of the program. This was because the program needs to have an UI first, which the **logics** can later be based and tested on to produce accurate results. Besides, the UI proved to be much more of a challenge because of its novelty to the author. Other than that, the process of generating classes in the logic parts unfolded as intended. This in itself is a very meaningful lesson that implementation needs to be in accordance with the type and structure of the program.

Final Evaluation

Overall, the program quality is satisfactory. Despite one major bug, most of it operates considerably smoothly. There are more than enough functionalities that make it an interesting program. The source code was written with thought and intent, carefully organized and commented, and thus, easy to read and understand. However, there is still much room for improvement. Creating a more advanced "wall" mechanism can be an example. The layout can be improved to look more like an industry product. Etc. The existing structure of the program is robust, thus, there is no need for drastic changes. However, if changes were to happen, they can still be carried out conveniently. The

author reckons that if the whole project is restarted, all the processes already undergone will be repeated, with no deviation.

References and Links

- Craig Reynolds (1999). Steering Behaviors for Autonomous Characters. Available at http://www.red3d.com/cwr/steer/gdc99/
- Vijay Pemmaraju (2013). 3 Simple Rules of Flocking Behaviors: Alignment, Cohesion, and Separation. Available at https://gamedevelopment.tutsplus.com/tutorials/3-simple-rules-of-flocking-behaviors-alignment-cohesion-and-separation--gamedev-3444
- Fernando Bevilacqua (2012). Understanding Steering Behaviors: Wander. Available at https://gamedevelopment.tutsplus.com/tutorials/understanding-steering-behaviors-wander--gamedev-1624
- Java Swing Library. https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html