Simulation of free shoals in the ocean

Aguiari Matteo, Branchesi Alice, Graglia Lorenzo, Panizza Giacomo

August 2022

1 Introduction

This is the report for the latest version of the simulation of free shoals. The version was updated in July, presented and corrected in the last oral exams' session by Panizza Giacomo. In the following lines are highlighted the part that are changed between the first version and this one.

The aim of this project is to simulate the behaviour of free shoals in the ocean and is achieved by recreating the movements of a lone fish (from now on called boid) and its interactions with other individuals close to him. We defined "close" any boid within a circle of radius d_{min} pixels from its neighborhood.

$$|\vec{x}_{b_i} - \vec{x}_{b_j}| < d_{min} = 145\ px$$

This math formula is the base computation used in one of the program's cardinal functions and is useful to discriminate whether or not other rules are in effect. It's also important to remark the fact that boids have an angle of view of 75° from both sides.
The whole simulation can be confined in a space with borders, resembling a huge aquarium, or, with periodic borders conditions activated, the boids are free to move in an seemingly infinite space, that teleports them on the other side when they reach a border. So, for example, if a boid exits from the left side of the screen, it comes back in from the right one, with the same speed and direction. The behaviour of the boids in the space is based on three fundamental rules described here, where v_i represents the component that will be added to the boids previous velocity as a result of the rule application:

- separation - when this rule is turned on the boids tend to step away from one another accordingly to the following formula:

$$\vec{v}_1 = -s(\sum \frac{1}{(\vec{x}_{b_j} - \vec{x}_{b_i})}) \ , \ |\vec{x}_{b_i} - \vec{x}_{b_j}| < d_b = 10\ px,\ j \neq i$$

- alignment - allows the creation of a shoal, a boid that comes across another fish tends to follow the trajectory of the neighbors as follows:

$$\vec{v}_2 = a(\frac{1}{n-1} \sum \vec{v}_{b_j} - \vec{v}_{b_i}),\ j \neq i$$

- cohesion - the cohesion rule implements "local" centers of mass, one for every fish and iteration, by taking the mean of the position-vectors of all the boids that are in its field of view and nearby, towards whom it will tend to move. The formula upon which this rule is implemented is:

$$\vec{v}_3 = c(\frac{1}{n-1} \sum \vec{x}_{b_j}),\ j \neq i$$

If desired, there is also the possibility to add predators to the simulation. These new elements resemble sharks, able to follow and eat fish. On the other end the prey will react to the hunters' presence by increasing their speed. As for boids, sharks also have a fixed angle of view. One last feature lets the user toggle ON/OFF a red cross, that represents the current position of the center of mass of the shoal.

2 How to use

Before compiling the program, it is important to turn on a graphic tool such as MobaXterm or Xlaunch, together with the graphic library of SFML.
To compile and execute the program, the suggested commands to input in the terminal are:

```
$ g++ main.cpp simulation.cpp shoal.cpp -o simulation -lsfml-graphics -lsfml-window
-lsfml-system -Wall -Wextra -O2

$ ./simulation
```

Some general instructions, together with the request to input personalized parameters will be printed on the terminal; at this point the user is requested to insert the initial speed of both fish and predators (it must be a positive value, we suggest "3"), the number of boids (it must be a positive and integer value, we suggest "50") and the maximum number of predators allowed (it must be a positive and integer value, we suggest "2"). The simulation will then start running with the current pre-settings. While the graphic interface is on, the user is allowed to add new boids anywhere within the window by clicking in the preferred location with the left mouse button. Respectively, with a right click, a predator will appear in the current mouse location. In order to broaden the user interaction, Some of the keyboard buttons are linked to specific functions that can be enabled and disabled. Here's the list of the possible options and the correspondent keys:

- S enables/disable the separation function;
- X enables/disable the alignment function;
- C enables/disable the cohesion function;
- P instantly kills all predators;
- K allows/denies sharks to eat boids;
- W switches between "aquarium mode" and periodic borders condition.
- M toggles on/off the red cross representing the shoal's center of mass.

3 How it's written

This program is made up of six files:

- **simulation.hpp** : it is the header file that contains the declaration of the class "Simulation" and its methods. In this file are included the standard library and the SFML dependencies, containing all the objects and functions needed. The purpose of this section of code is to operate for the most part the graphical interface, to accept all the inputs from the user, and to menage the evolution of the whole simulation. In this file also appears an #include guard, that prevents the program to break the "one-definition-rule" while compiling more .cpp files with the same inclusion simultaneously.

- **simulation.cpp** : in this second file are defined the methods of "Simulation" together with a template function, specifically designed to verify whether or not a specific user user input is valid. Upon creation of an object of the class Simulation, the constructor initializes the graphic window and numerous variables, among which there is an object of the class "Shoal". At every iteration the "Update" method will guarantee the flawless execution of all the functions needed.

- **shoal.hpp** : this file contains the declaration of the class "Shoal" and its methods. The purpose of this class is to manage the computational part of the simulation and to draw the entities on the graphic interface every updating cycle. The specifications of the single functions will be

further discussed later. “shoal.hpp” also contains the struct “Boid”, which is the building block of the whole program, being able to encapsulate two vectors, one for the position and one for the velocity of the represented entity, and a float for its rotation relative to the x-axis, counter-clockwise (an angle of 45° would be in the IV quadrant of a canonical Cartesian plane).

- **shoal.cpp** : containing the fundamental movement-related functions this file represents the real “brain” of the program. Every cycle fish’s and predators’ position, velocity and texture are updated in conformity with external inputs and fixed values initialised by the “InitShoal” method. Regarding the boids, seven movement functions are implemented, the first two dealing with the “aquarium” borders, “move_bouncing” in case of active walls and “move_pacman” in case of periodic border condition. The other five concern the actual position evolution rules: (“move_separation”, “move_alignment”, “move_cohesive”), mentioned above, “move_escape”, that allows boids to escape from visible predators when they’re closer than $d_e = 145\ px$ and “VCorrectionBoids” that, as suggested by the name, tends to “correct” the boids’ speed, slowing them down when they move too fast, and accelerating them when they move too slow, accordingly to the ”initial velocity” parameter chosen by the user. All this functions are called within the more generic method “moveBoids”. For the predators there are the same two functions for the basic movement, walls activated or deactivated, respectively, “move_bouncingPredator” and “move_pacmanPredators”, an additional “killing” function, that deletes a boid when the predator is close enough to eat it ($d_k = 25\ px$) and the “move_hunting” function. This last method allowing a predator to move towards the center of mass of the fish he sees, as long as it is within its radius of action. As for the boids there is a “VCorrectionPredator” function to change the speed of the sharks. All this functions are called in the more generic method “movePredators”.
Three more methods give some insights of the spatial relation between entities: “visible_b” checks if a boid enters the field of view of another fish, “visible_p” checks weather or not a predator sees a specific fish and “nearby” returns “true” whenever two entities are closer than a given radius.
One last portion of the code is dedicated to the evolution of the center of mass, represented on screen, when active, by a moving red cross.

- **vector.hpp**: this file contains the overloading of some basic operations for the vector-type sf::Vector2f, together with the implementation of a function that outputs the norm of a given vector, “norm”, one that outputs the relative rotation mentioned above, “getRotation”, and one that calculates the position of the center of mass of the shoal, “CalculateCM”.

- **main.cpp** : in this file is encoded the initial output printed on the terminal at execution-time, with the instructions the user must follow. An object of the “Simulation” class is then created, with a while loop where the method “update” is called, updating the graphic window until it is closed by the user, at which point the programs is terminated.

4 Test

All the tests are encoded in the “aquarium_test.cpp” file. They aim to verify the correct implementation of the methods defined in the class Floak and check the correctness of the additional functions and operators’ overloading encoded in the ”vector.hpp” file. To compile we suggest using the following commands:

```
$ g++ aquarium_test.cpp shoal.cpp simulation.cpp -o tests -lsfml-graphics -lsfml
-window -lsfml-system -Wall -Wextra

$ ./test
```

To facilitate the testing procedure, two new methods of the class Shoal are introduced: “spawnCustomBoid” and “spawnCustomPredator”.