Bytom, 26.05.2021

Computer Programming Laboratory
IV semester

Final Programming Project Report

Project topic: Boids Simulation

Author: Radosław Rzeczkowski
Email: radorze093@student.polsl.pl
Tutor: Roman Starosolski

1. Topic and description

The aim of this project was to create an interactive desktop application that simulates Boid behaviour.

The idea of Boids was proposed by Craig Reynolds in 1986. They create a computer model of coordinated animal motion such as bird flocks and fish schools. It is an example of a complex system where complex, organic and unpredictable behaviour stems from simple rules.
The three rules that each Boid has to obey are:

- Separation: steer to avoid crowding local flockmates.
- Alignment: steer towards the average heading of local flockmates.
- Cohesion: steer to move toward the average position of local flockmates.

Local here means that a Boid can only perceive other Boids that are within some distance from it. This addition is crucial to simulating a complex system since a unit element of such system should have limited knowledge of it's surroundings to achieve balance between chaos and predictability.
Steering is defined by Craig Reynolds in terms of two or more dimensional vectors such that the desired direction of movement (for example towards the average position of local flockmates) minus the current velocity vector gives the Steering vector that is treated as an acceleration applied to the boid. Many acceleration vectors can then be added together and finally are added to the current velocity of the boid. The velocity is treated like a displacement vector for a frame since it is just added to the current position of the boid. These rules create interesting patterns and behaviours when there is a large number of boids being simulated.

The user is able to observe the simulation via a GUI made using the SFML library. In a 2D view a Boid is represented by a triangle. All Boids are part of one of six Boid Groups, where each group has a designated colour by which it is distinguished.

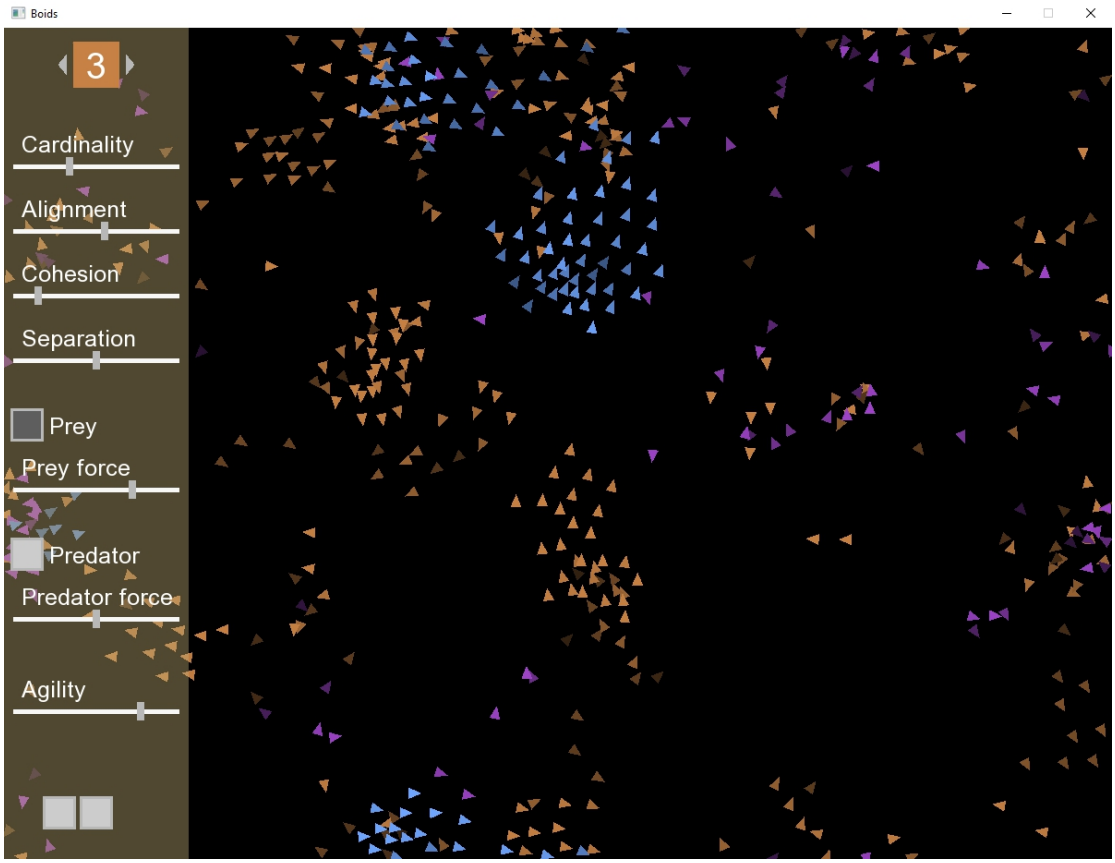
Basing on the original three rules one can quite easily introduce new, custom rules. In the application two such rules have been added: Predator and Prey rules. A Boid Group can be a predator group, prey group or neither. Boids from a prey group will move away, or flee, from Boids from a predator group. Boids from predator groups will move towards, or chase, Boids from prey groups.The behaviour can be further customized by changing the weights attributed to the rules. For example the prey rule can have a much higher weight than any other rule since it's critical to the figurative survival of a Boid.

The user has control over the simulation by the means of a control panel. There, he can change parameters of a selected Boid Group: weights of movement rules and where a group is a predator, prey or neither. The user can also click in the simulation area to attract or repel Boids relative to the cursor.

Because of the large numbers of Boids in simulations and the real time nature of the application a spatially-aware data structure is ideal. The bin-lattice spatial subdivision data structure has been chosen to optimize the „local" queries of each Boid.

2. External specification

After starting the program the simulation is stared immediately. Two groups of 100 Boids are created with default parameters. The user is now free to interact with the simulation and observe it.



At the top of the control panel is the group selection. Groups are identified by their unique colour. There are always 6 groups. At the start of the program groups 2-5 contain zero Boids. All groups have separate parameters. Boids from a given group do not interact with Boids from other groups, apart from a Predator - Prey dynamic.

Below there are four sliders:
The first slider controls the cardinality of a group. It ranges from 0 to 1000 Boids.
The following three sliders control weights of movement rules. The user is free to experiment with different combinations of these values and see how they affect the behaviour of a group.

Next are two pairs of controls. The check boxes can be clicked on to toggle them. A Boid Group cannot be Prey and Predator at the same time. Boids from Prey groups will run away from Boids from Predator groups. Boids from Predator groups will chase Boids from Prey groups. There needs to be at least one Prey and one Predator group for these settings to have an effect in the simulation. Associated sliders control the weights for running away and chasing. By default they are much higher than other rules and have a larger value range.

The last slider is the Agility slider. It controls the agility of a group. In reality it controls the maximum value of a force that a Boid can experience. All forces (movement rules) will be clamped to this value. In practice it controls how quickly a Boid can change it's course and accelerate.

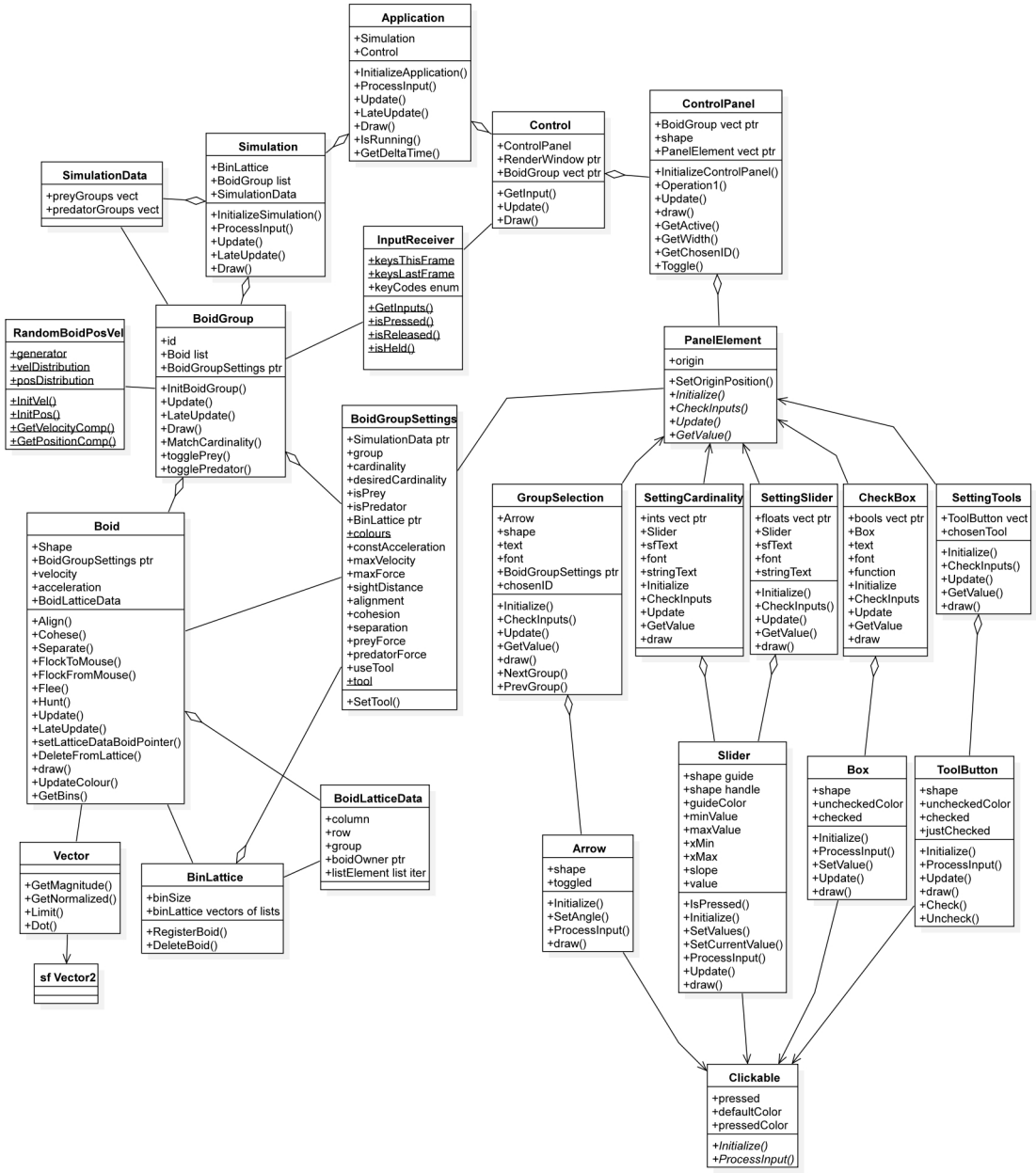
At the bottom of the panel there are two squares. These are tools that can be used when clicking in the simulation area. The left one is attraction, it is active by default. It will attract Boids to the cursor. The right one is repulsion.

The user can hide the control panel by pressing Space. If the user clicks in the simulation area (outside of the control panel) the selected Tool will be used. Left mouse button will use the tool within the selected Boid Group. Right mouse button will use the tool on all Boids.

Although it is possible to have six groups of 1000 Boids each such a large amount is not advised and will cause the program to lag significantly and could lead to a crash. The program can handle 1000 Boids in two groups with smooth framerates. At three full groups the application starts to lag. There can be easily six groups of 500 Boids each.

If compiled and run within Visual Studio it should be in Release configuration for best performance.

3. Internal specification



Full specification is attached below in the form of generated documentation. Chosen classes:

Boid

This class represents a Boid entity. It is visually represented on screen by a triangle that rotates to align with it's velocity vector. The color of the Boid indicates to which group it belongs. It dims and gets brighter slightly based on the rotation of the Boid to add some visual interest. The Boid moves according to it's movement rules. The rules are implemented as methods of the class. Most rules query the BinLattice for nearby Boids and then build an acceleration vector based on the position/heading of the Boids within range. The acceleration vectors from all rules are added together and form a final acceleration vector for a frame. This vector is then added to the current velocity vector. At the end of a frame the Boid is displaced by the velocity vector. Each Boid would normally have to check all other Boids to find the

ones within it's local radius. This leads to a time complexity of O(n^2) and is very bad for performance in this real time setting. That's why the BinLattice data structure is used.

BinLattice

This class implements a spatially aware bin-lattice spatial subdivision data structure. It speeds up the distance query by dividing the area of the simulation into equal, fixed bins. The Boids are assigned to bins based on their location each frame. When a Boid wants to find others within some radius from it it can check the Boids that are in nearby bins instead of all Boids present in the simulation. Given that the Boids are somewhat distributed in the simulation area and are not all covered by the nearby bins this data structure significantly improves performance. The class uses a three-dimensional vector to represent division of Boids by Group, Bin row and Bin column. At each of these coordinates there is an std::list. This data structure was chosen to allow for fast addition and deletion of elements without invalidating pointers and iterators to the elements. Each frame all Boids register to the BinLattice. Based on their coordinates a pointer to the Boid is placed in an according bin. The Boid updates information about it's current bin coordinates and an iterator to itself in the list in the BoidLatticeData class. Using this information it will be quick to remove the Boid from the current bin if it moves to an area covered by a different one. When a Boid wants to find nearby Boids it iterates over the Boids pointed to by pointers stored in lists in adjacent bins.

The project uses Templates in the Vector<T> class to implement a wrapper for the sf::Vector2<T>, a 2D vector class from SFML library. The Vector<T> class was created to extend the functionality of the basic SFML Vector2. Operations like vector addition, dot product, multiplication by scalar, and normalization were implemented. These operations are used extensively in the Boid class methods.

std::list was used from STL containers. It was chosen to store elements in places where deletion and addition needed to be as fast as possible. It's property to never invalidate pointers to stored objects regardless of deletions or additions of others was also very important. BinLattice class uses a list to represent a bin that stores pointers to Boids. Boids themselves are stored in lists in the BoidGroup class to ensure that a pointer to a Boid is valid as long as the Boid is not removed from the list.

STL algorithms were also utilized in this project, mainly to perform operations on the std::list such as constructing elements in place or removing them. An erase-remove idiom was also used to delete elements of a std::vector by value.

Smart pointers were used to provide access to some elements such as the BoidGroupSettings in BoidGroup class. In this case a unique_ptr was used because the object should be deleted once the BoidGroup that holds it is destroyed. It also ensured that regular pointers to BoidGroupSettings used extensively by both Boids and GUI elements were valid regardless of how and where the BoidGroup that holds it is stored.

General scheme of program operation:

First all elements are initialized with corresponding methods. Then the program goes into the main loop where it reads and processes user inputs, updates the simulation and GUI, performs a late update that includes operations like moving off-screen Boids to the opposite side of the screen, draws all drawable elements to the screen and calculates delta time (time between this and last frame). The loop can be exited by stopping the program by, for example, pressing the Close Window button. Then all objects are deconstructed.

4. Testing and debugging

The program was tested many times, effectively after the implementation of each functional feature. Thanks to this many bugs were caught and fixed early. One bug however was especially malicious and took a long time to find. At the beginning it would cause Boids to disappear seemingly randomly, but it was not noticed. It caused crashes due to illegal memory access once the BinLattice class was introduced. I assumed it was a problem with the new class and the pointers it uses, since Boid objects were having mangled fields (the memory access violation was happening because a BinLattice method was trying to access vector elements way beyond their actual size because the Boid's garbage location coordinates were being used as bin coordinates on the vector). After improving and assuring that the new class works properly I started to look in other classes that interact with Boids. After a long hunt I finally arrived at the source of the problem, which was laughably simple: division by zero. In the most low-level class, the Vector<T> wrapper, there were no zero-checks in methods like Normalize that divide a vector by it's length. That, under right and quite specific circumstances, led to division by zero and garbage output. This major mistake would've gone unnoticed and caused further problems if it wasn't for the BinLattice implementation. It was a good and simple lesson.
The program was also tested with large numbers of Boids and with different configurations of Prey and Predator groups.
The UI was tested and debugged while some behaviour was deemed sensible enough to be accepted, such as the sliders „sticking" to the cursor after LMB has been released outside of the control panel area.