Flocking Simulation

Project Documentation

Table of contents

General description

Overview

Flocking Simulation is a graphical 2d simulation written by Sergey Zakuraev
as a part of Aalto University Programming 2: Studio course, where a user-selected number of vehicles (boids) move in a life-like flocking manner. Each vehicle has a number of parameters such as a position in the 2d world and a velocity, and each vehicle follows four rules to achieve realistic behavior. The app has been implemented at the difficult level, all the required features have been implemented, along with several additional ones.

Details

The app is a scala application that can be run through Eclipse integrated development environment (IDE), it is written in the object-oriented, imperative style. It has a number of classes and singleton objects that interact with each other to create a graphical 2d simulation which can be controlled by the user. Each vehicle follows the following 4 rules to achieve realistic behavior:

1. Separation - avoiding collision with other vehicles
2. Cohesion - aiming for the center of mass of nearby vehicles
3. Alignment - aligning itself with direction of nearby vehicles
4. Obstacle avoidance - avoiding collision with obstacles

The rules are based on Craig Reynold's article "Steering Behaviors For Autonomous Characters" [1],[2] and are described in detail in the *Algorithms* section. Additionally, the simulation features a number of other parameters such as vehicle detection radius and top speed.

The app includes all of the features required for the difficult level:

- A graphical user interface (GUI) accessible through Eclipse IDE. It can be run as a Scala Application.
- Each new vehicle initially moves in a random direction
- A simulation state can be loaded from a text file of custom format
- Weight of the four rules can be adjusted live (i.e. without stopping the simulation)
- Other adjustable parameters such as top speed of vehicles and their detection radius

Additionally, the app includes several extra features:

- Simulation features obstacles and obstacle avoidance behavior
- Obstacle avoidance controls that can be shown or hidden
- Obstacles can be added by user
- The simulation state can be saved into a text file
- App features 6 presets with different starting conditions
- App window can be resized live

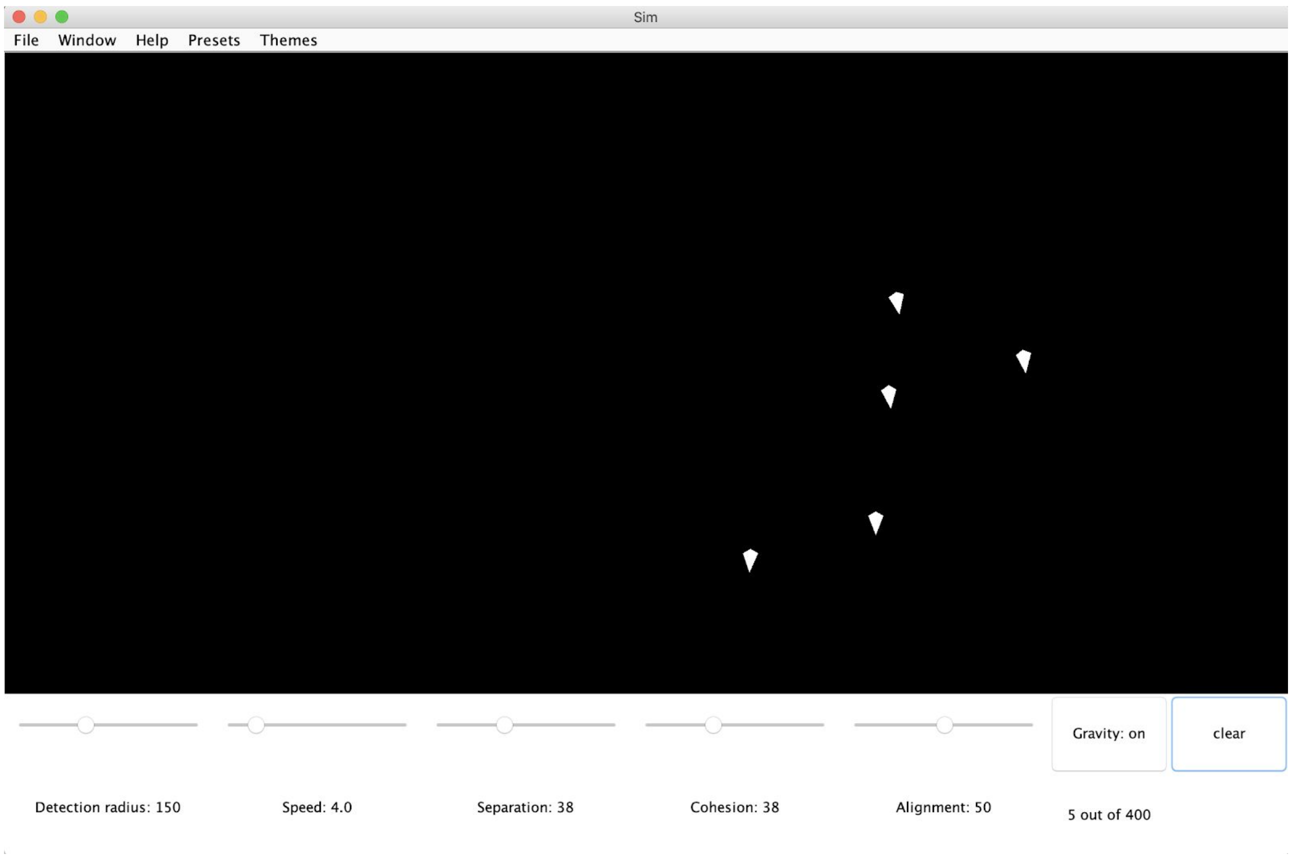
- Simulation has 3 color themes that can be changed live
- There is a help section that opens pop-up windows with instructions
- Simulation features a gravitational pull that attracts vehicles towards the center of simulation, and can be toggled
- Simulation can be reset into the default state

The development process and final app do differ somewhat from the initial plan. The class structure has been expanded to increase code readability and potential reusability. It has been split into GUI and logic parts. GUI has been altered somewhat: for example the user does not type in the number of vehicles to be added, but instead clicks on the simulation panel. The schedule was followed rather loosely, but it did not come to the situation, where the entire project had to be implemented in a short period of time.

User interface

Overview

The program can be launched through the Eclipse IDE by running FlockSimulationApp class inside the gui package. The user can add new vehicles and alter the state of the simulation. The GUI is implemented with scala.swing and java.awt libraries. It appears as a single resizable window with a menu bar, a simulation field where vehicles and obstacles appear, and a controls panel.



Detection radius: 150
Speed: 4.0
Separation: 38
Cohesion: 38
Alignment: 50
5 out of 400

Image 1: program GUI

Details

In order to run the program, the user needs to have Java SE Development Kit 8 (https://www.oracle.com/java/technologies/javase-jdk8-downloads.html) and Scala IDE for Eclipse (http://scala-ide.org/download/sdk.html) installed on their computer. The program can be imported into Eclipse as an archive. It can be launched by running the FlockSimulationApp.scala class inside the gui package as a Scala Application.

The program appears as a single window, resizeable up to a limit. It can be maximized, minimized and closed. The upper part of the window contains a menu, the middle is an empty simulation panel, where vehicles can be added, and the bottom is a controls bar with a set of sliders, labels and buttons, allowing for control of the simulation.

Initially, the simulation contains no vehicles, and parameters are adjusted for most realistic behavior. The user can add new vehicles by left-clicking on the black simulation panel, or by choosing a preset from the “Presets” menu. The number of vehicles inside the simulation, the maximum possible number, as well as the value of behavior parameters can be seen and adjusted in the controls bar. The user can remove all vehicles and obstacles by pressing the “clear” button in the controls bar or the “File” -> “Reset” options in the menu. Note that the reset also returns all the parameters to initial values. An artificial pull towards the center of simulation can be toggled through the controls bar. It makes the simulation appear more dynamic.

The menu bar features the following sub-menus:

- *File*
  Through the ‘File’ section of the menu bar, the user can reset the simulation, close it, save and load the simulation state into/from a text file. More details on this can be found in the *Files and Internet Access* section.

- *Controls*
  The user can show or hide additional controls for obstacle avoidance behavior and a button that toggles effects of mouse clicks on the panel between adding vehicles and adding obstacles.

- *Help*
  The user can open pop-up windows with instruction on usage of the app and loading/saving.

- *Presets*
  The user can select a simulation preset, which resets the simulation, sets parameters to specific values, and adds vehicles and in some cases obstacles.

- *Themes*
  The user can change the visual appearance of the app, by selecting one of the 3 themes.

Program structure

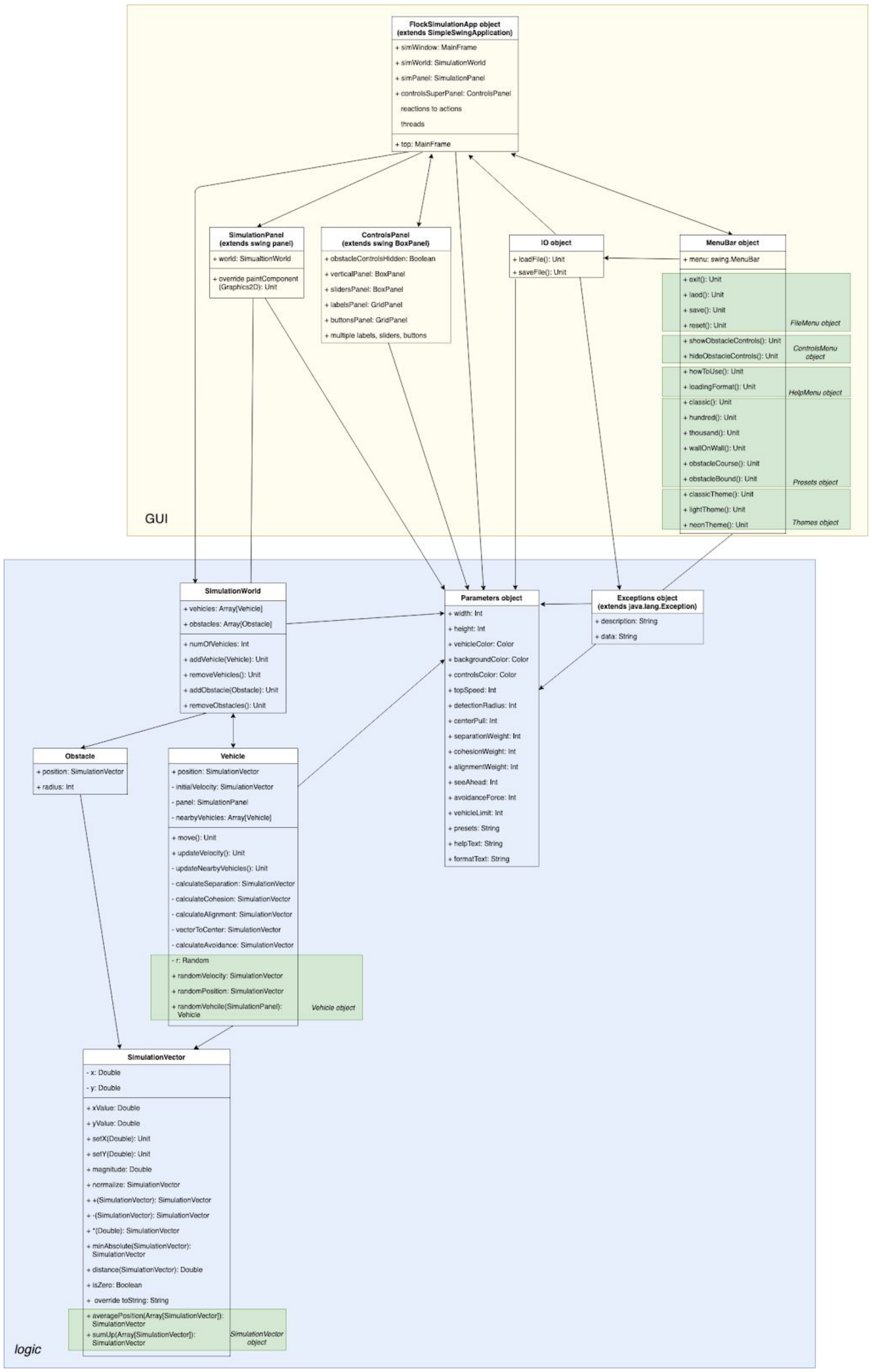

FlockSimulationApp object (extends SimpleSwingApplication)
+ simWindow: MainFrame
+ simWorld: SimulationWorld
+ simPanel: SimulationPanel
+ controlsSuperPanel: ControlsPanel
reactions to actions
threads
+ top: MainFrame
SimulationPanel (extends swing panel)
+ world: SimualtionWorld
+ override paintComponent (Graphics2D): Unit
ControlsPanel (extends swing BoxPanel)
+ obstacleControlsHidden: Boolean
+ verticalPanel: BoxPanel
+ slidersPanel: BoxPanel
+ labelsPanel: GridPanel
+ buttonsPanel: GridPanel
+ multiple labels, sliders, buttons
IO object
+ loadFile(): Unit
+ saveFile(): Unit
MenuBar object
+ menu: swing.MenuBar
+ exit(): Unit
+ laod(): Unit
+ save(): Unit
+ reset(): Unit
FileMenu object
+ showObstacleControls(): Unit
+ hideObstacleControls(): Unit
ControlsMenu object
+ howToUse(): Unit
+ loadingFormat(): Unit
HelpMenu object
+ classic(): Unit
+ hundred(): Unit
+ thousand(): Unit
+ wallOnWall(): Unit
+ obstacleCourse(): Unit
+ obstacleBound(): Unit
Presets object
+ classicTheme(): Unit
+ lightTheme(): Unit
+ neonTheme(): Unit
Themes object
GUI
SimulationWorld
+ vehicles: Array[Vehicle]
+ obstacles: Array[Obstacle]
+ numOfVehicles: Int
+ addVehicle(Vehicle): Unit
+ removeVehicles(): Unit
+ addObstacle(Obstacle): Unit
+ removeObstacles(): Unit
Parameters object
+ width: Int
+ height: Int
+ vehicleColor: Color
+ backgroundColor: Color
+ controlsColor: Color
+ topSpeed: Int
+ detectionRadius: Int
+ centerPull: Int
+ separationWeight: Int
+ cohesionWeight: Int
+ alignmentWeight: Int
+ seeAhead: Int
+ avoidanceForce: Int
+ vehicleLimit: Int
+ presets: String
+ helpText: String
+ formatText: String
Exceptions object (extends java.lang.Exception)
+ description: String
+ data: String
Obstacle
+ position: SimulationVector
+ radius: Int
Vehicle
+ position: SimulationVector
- initialVelocity: SimulationVector
- panel: SimulationPanel
- nearbyVehicles: Array[Vehicle]
+ move(): Unit
+ updateVelocity(): Unit
- updateNearbyVehicles(): Unit
- calculateSeparation: SimulationVector
- calculateCohesion: SimulationVector
- calculateAlignment: SimulationVector
- vectorToCenter: SimulationVector
- calculateAvoidance: SimulationVector
- r: Random
+ randomVelocity: SimulationVector
+ randomPosition: SimulationVector
+ randomVehcile(SimulationPanel): Vehicle
Vehicle object
SimulationVector
- x: Double
- y: Double
+ xValue: Double
+ yValue: Double
+ setX(Double): Unit
+ setY(Double): Unit
+ magnitude: Double
+ normalize: SimulationVector
+ +(SimulationVector): SimulationVector
+ -(SimulationVector): SimulationVector
+ *(Double): SimulationVector
+ minAbsolute(SimulationVector): SimulationVector
+ distance(SimulationVector): Double
+ isZero: Boolean
+ override toString: String
+ averagePosition(Array[SimulationVector]): SimulationVector
+ sumUp(Array[SimulationVector]): SimulationVector
SimulationVector object
logic

Image 2: program UML (Also attached in appendixes)

Overview

The program was developed using both top-down and bottom-up design. It is split into 3 packages: gui with visual components, threads and event handling, logic with mathematics, behavior algorithms, and tests with unit and functional tests. Packages contain classes and singleton objects which interact with each other through public interfaces and alter each others' variables.

Details

Gui package

- *FlockSimulationApp (*singleton object)
  This object launches the app itself. It creates instances of SimulationWorld and instances of SimulationPanel and ControlsPanel, uses MenuBar menu variable to create the 3 GUI components. It also launches two threads that separate calculations and graphics.

- *MenuBar (singleton object)*
  Contains a variable of scala.swing.Menubar that represents the top menu inside the app. Contains helper objects inside it, that control actions of menu items. MenuBar is an object and not a class, as it does not require parameters and does not get instantiated multiple times. It is split into inner helper objects for better readability and extensibility.

- *SimulationPanel (class)*
  Extends swing panel, and is responsible for visually representing vehicles and obstacles. It receives information about them through the SimulationWorld, instantiated inside the FlockSimulationApp and passed into the SimulationPanel as a parameter.

- *ControlsPanel (class)*
  Extends swing BoxPanel and is responsible for the controls and labels part of GUI. Instantiates these controls from the swing library and also takes care of their reactions.

- *IO (singleton object)*
  Responsible for loading and saving files using scala io. Has two methods which take care of the respective operations. Is an object and not a class, as it does not take in parameters, has no public variables and is only used throughout the 2 methods it has.

logic package

- *SimulationWorld*
  Represents the inner state of the simulation. Stores information about vehicles and obstacles, has methods for adding and removing these. Can be accessed by SimulationPanel and FlockSimulationApp.

- *Vehicle*
  Class that when instantiated, represents individual vehicles. Each vehicle knows what SimulationWorld it is inside in order to identify nearby vehicles, stores and updates its own position, velocity, nearby vehicles, and has private methods for calculating vectors that

enforce the 4 rules. Also has a companion object that allows creation of random velocities, random positions, and vehicles with random velocities and positions.

- *Obstacle*
  Instances of obstacles, created by simulationWorldrepresent individual obstacles. Each obstacle has a position and a radius.

- *SimulationVector*
  Instances of this class represent positions and velocities within the simulation. The class has a number of public methods for mathematical operations such as addition, multiplication and normalization. Also has a companion object with utility methods.

- *Parameters*
  This object stores the variables that get changed by the user such as weights of rules, and gui parameters such as colors and dimensions, as well as constants help text. It can be accessed by almost all other objects and classes and thus makes their customization easy. Is an object and not a class as it does not need to be instantiated multiple times and is only needed for its mutable & immutable variables.

- *Exceptions*
  This object stores case classes of exceptions used in the project and a method for showing a pop-up window, that appears in case of loading issues.

The test package contains a class of unit tests for the SimulationVector and a class of functional tests for vehicles and simulation panels. More information on the tests can be found in the *Testing* section.

The class structure loosely resembles the initially planned one: there is still the FlockSimulationApp class that represents the simulation, which has an instance of swing MainFrame inside it, but it has been supplemented with a number of other GUI classes and objects to increase readability and extensibility. All these classes and objects could have been inside the FlockSimulationApp and initially that was the case, but it has been split up for reasons stated above.

The planned SimulationWorld, Vehicle and Obstacle were implemented roughly as planned. Additionally, a custom SimulationVector class and Parameters object have been added for convenience, as well as the Exceptions object for handling errors.

One possible change to the program structure could be removal of the parameters object and incorporation of its data into the FlockSimualtionApp or elsewhere as a companion object. This would however require almost all the classes to be able to communicate with the FlockSimualtionApp. Another option would be to move the drawing of vehicles and obstacles from the SimulationPanel to vehicles and obstacles themselves, but that would mix up the GUI and logic parts of the program.

Algorithms

Overview

The app uses vector operations and arithmetics to create life-like behavior in vehicles. Each vehicle has a position and a velocity. It keeps track of nearby vehicles and calculates 4 vectors acting on its velocity every 10 milliseconds. The algorithms which are more or less complex include mathematical formulas, while the trivial ones do not.

Vector algorithms

Calculations inside the simulation are used with instances of SimulationVector. Note: Position refers to position vector, i.e. the ‘tip’ of a vector. Simulation vector has a number of algorithms that use basic mathematics and linear algebra:

- *Magnitude of a vector*
  Program calculates the magnitude of a vector using Pythagor’s theorem
  m = sqrt(x^2 + y^2)

- *Normalizing a vector*
  Program scales the vector’s magnitude to 1, preserving its direction, by dividing its xValue and yValue by magnitude
  n = (x/magnitude, y/magnitude)

- *Distance between two vectors*
  Program calculates the distance between the ‘tips’ of two vectors using Pythagor’s theorem
  d = sqrt((x1-x2)^2 + (y1 - y2)^2)

- *Adding two vectors*
  Program adds two vectors by combining their xValues and yValues

- *Subtracting two vectors*
  Program subtracts two vectors by subtracting their xValues and yValues

- *Multiplying two vectors*
  Program multiplies two vectors by multiplying their xValues and yValues

The companion object of the class has the following algorithms:

- *Calculating average position for a sequence of vectors*
  Program calculates the average position of a number of vectors by combining their xValues and yValues and then dividing by the number of vectors

- *Summing up multiple vectors*
  Program adds all the vectors within a list

Vehicle algorithms

Each vehicle inside the simulation has two parameters represented by instances of SimulationVector: position and initialVelocity.

- *Finding nearby vehicles*
  Each vehicle can find a list of nearby vehicles within a constant radius by checking if the distance between itself and the other vehicle is less than the detection radius for every vehicle in the SimulationPanel that the vehicle belongs to.

- *Moving*
  Each vehicle can move, i.e. update its position by adding its velocity to the current position. Additionally, if the vehicle's position gets out of the simulation's bounds, the move() method resets its position to one on the opposite side of the simulation by resetting the vehicle position's x or y values.

- *Limiting own speed*
  Each vehicle can limit its top speed by normalizing its velocity and scaling by topSpeed.

- *Calculating separation*
  Each vehicle calculates its separation, i.e. a vector that repulses it from other vehicles. Initially, the repulsion vector has values (0, 0). For each nearby vehicle, the position of 'our' vehicle is subtracted from the position of the nearby one, normalized and then added to the repulsion vector. The repulsion vector is then normalized, to later be scaled by a weight. The result is a vector that faces as far away as possible from all nearby vehicles.

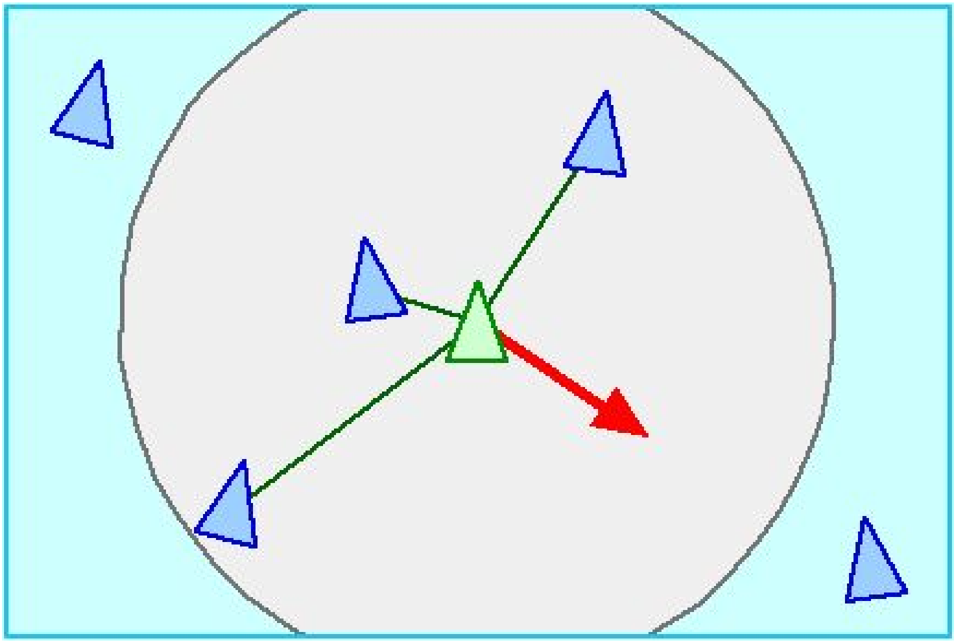


Image 3: separation [1]

- *Calculating cohesion*
  Each vehicle calculates cohesion, i.e. a vector that attracts it to the ‘center of mass’ of nearby vehicles. This is achieved by creating a sequence of nearby vehicles’ positions, and then calculating their average position. ‘Our’ vehicle’s position is then subtracted from this center of mass. The result is normalized to later be scaled by a weight.

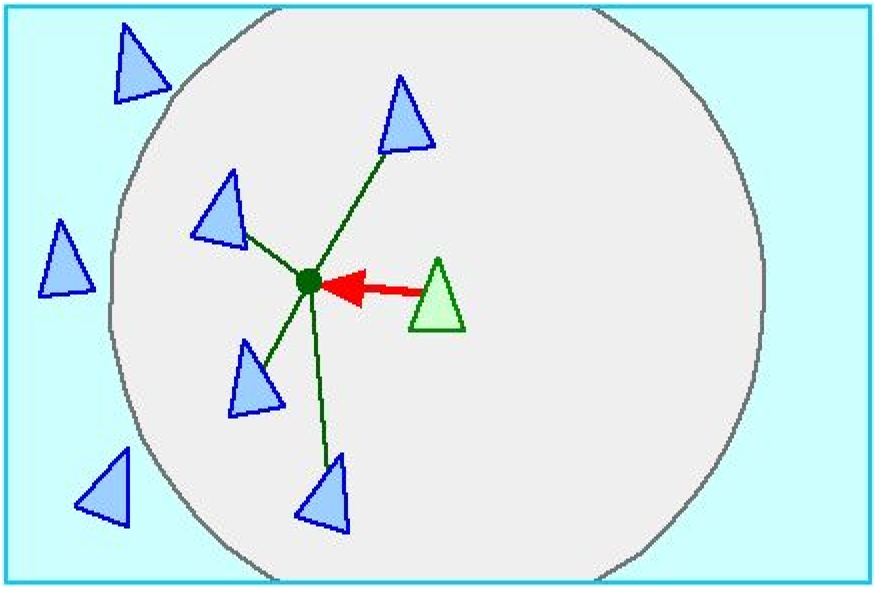


Image 4: cohesion [1]

- *Calculating alignment*
  Each vehicle calculates alignment, i.e. a vector that steers it in the same direction as nearby vehicles. This is done by calculating the average position of nearby vehicles’ velocities, instead of their positions as in *calculateCohesion.* The obtained vector is then normalized to later be scaled by a weight.

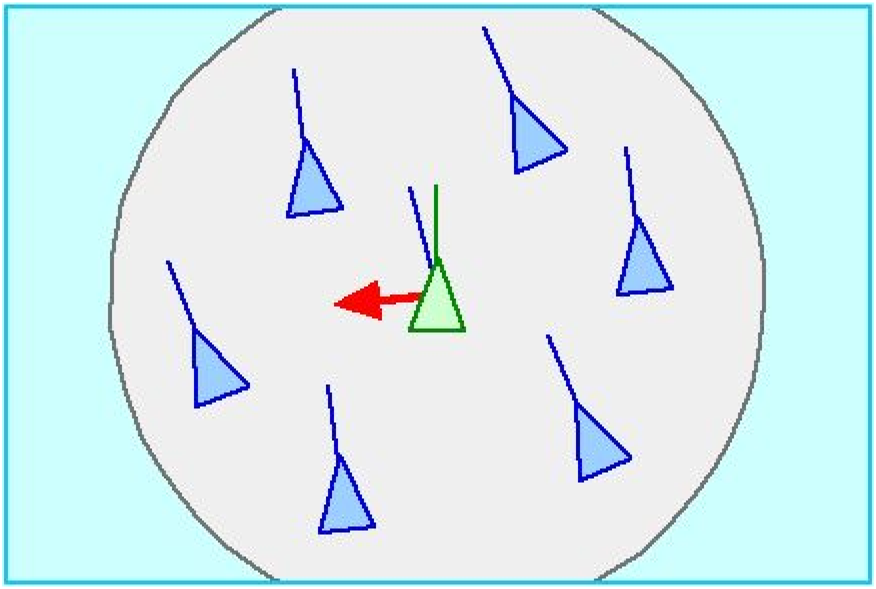


Image 5: alignment [1]

- *Calculating obstacle avoidance*
  Each vehicle calculates a vector repulsing it from the nearest obstacle in its line of sight. The vehicle creates a list of all obstacles that lie along its velocity vector scaled by seeAhead (green dashed line on image 6), by checking if the trajectory tip is within the radius of each obstacle. It then selects the nearest one to its position as the most threatening one. If such a threatening obstacle is found, the repulsion vector is calculated by subtracting the obstacle's position from the trajectory vector. This repulsion force is later added to the vehicle's current velocity. If the most threatening obstacle is not found, then a (0,0) avoidance vector is returned.

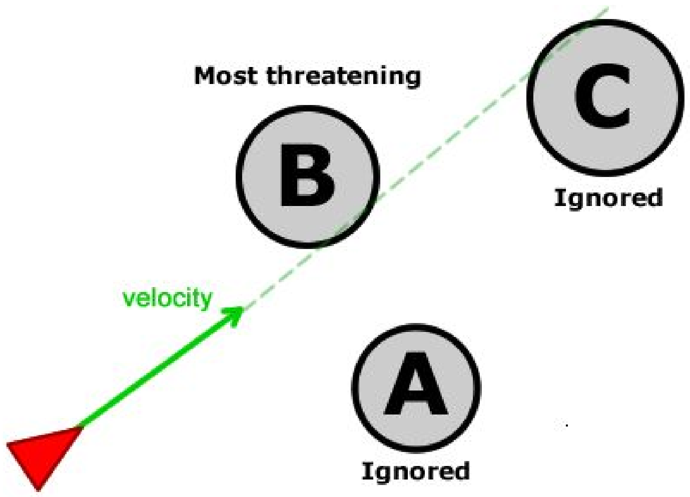


Image 6: obstacle avoidance [3]

- *Calculating gravitational pull*
  Each vehicle calculates how strongly it is pulled towards the center of simulation, i.e. a vector that pulls it towards the center. This is done by subtracting the vehicle's position from the center of simulation and then normalizing the result to later be scaled by a weight.

- *Updating velocity*
  Each vehicle updates its velocity by summing up its current velocity, with the four rule vectors above, each scaled by a constant weight, and the gravitational pull, then normalizing the result and scaling it by top speed.

The companion object of the class has the following algorithms:

- *Creating a random velocity vector*
  Creates a vector of random direction. The x component is calculated by taking the cosine of a random number between 0 and 360 degrees and then turning it to radians (degrees are used for readability), the y component is found the same way, but using sine.

- *Creating a random position vector*
  Generates a vector with random x and y coordinates within the range of the simulation's width and height.

SimulationPanel algorithms

- *Calculating coordinates for polygon models of vehicles*
  Each vehicle's polygon model has 4 corners, each corner has a pair of x and y coordinates. The tip of the vehicle is the sum of its position and it's velocity scaled by a weight. The back is the same, but velocity is subtracted from the position. The side corners of the polygon are found by finding vectors orthogonal (perpendicular) to velocity. The left side has velocity's x coordinate scaled by a positive weight and y coordinate scaled by a negative weight. The right side is the reverse. This is in other words a 90 degree clockwise and counterclockwise rotation:

$$\mathbf{a}^{\perp} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{a} = \begin{bmatrix} -a_y \\ a_x \end{bmatrix}.$$

  Image 7: counterclockwise [4]

FlockSimulation algorithms

- *Saving Vehicle coordinates and velocities*
  The program stores all the vehicles' x- and y- coordinates of position and velocities into arrays of pairs. It then concatenates each element of each pair into a string.

- *Loading files*
  The program splits the text file into an array of lines, and splits each line at every space. It then sets simulation parameters to components at specific indexes for each line and thus deals with all the parameters except vehicle coordinates and velocities. Vehicle coordinates and velocities are cleaned of all punctuation, and put into a single array. They are then grouped into two arrays, containing velocities and positions by supplementing the initial one with indexes and filtering them by parity. Finally, vehicles are added with positions from the first array and velocities from the second one.

Data Structures

Overview

The program uses Arrays and custom SimulationVector data types.

Details

Array:
Arrays are mutable, indexed and more efficient than other collections such as buffers and lists. For example List.apply() is an O(n) operation, taking linear amount of time depending on length of collection, while Array.apply() is a constant operation, taking some small amount of time regardless of the array's size. Arrays' mutability is used for updating each vehicle's nearby vehicles, I/O and testing. Their indexing is used for loading information from files.

Sets could have been used for similar purposes where indexing is not needed, but after testing, the simulation's performance did not improve, so the idea was discarded.

SimulationVector:
A custom class representing a mathematical vector in 2 dimensions. A vector is mutable, i.e. its x and y coordinates can be changed. It is not exactly indexed, instead it only has 2 elements, that can be accessed by .xValue and .yValue methods, similar to a pair.

The vectors are used to represent vehicles' and obstacles' positions in the world, as well as vehicles' velocities and forces acting upon them. In other words, each of the 4 guiding rules is a new vector that gets added to the vehicle's current velocity. The custom class is useful as It has specific methods useful for the simulation such as normalize and distance.

An alternative could have been the scala mutable Buffer collection, pairs or immutable Vector, but working with these without custom classes would have been more tedious.

Files and Internet access

Overview

The program has no access to the internet. It uses .txt files for loading and saving simulation states, .scala files for classes, .jar files for referenced libraries and auxiliary eclipse files for work inside the IDE.

Details

The user can save the current state of the simulation by clicking the 'File' menu and then 'Save'. They are asked where the file is to be saved via a pop-up window. The file should be named and saved with a .txt extension. Once done, a file of the following format is created:

```
speed: 3
detectionRadius: 50
centerPull: 1
separationWeight: 5
cohesionWeight: 4
```

alignmentWeight: 6
Vehicles: (897.314869260016 333.00721001729784, -2.9909340837994085 0.2330521537488805); (885.5050397256837 281.04858653562803, -2.925073219346687 -0.6662932248348375); (809.1875247277576 368.77595672833115, -2.8255575388211858 -1.0080796569769495);

The file stores information about certain parameters of the simulation, locations and velocities of every vehicle as pairs of Doubles. In the above example, the simulation has 3 vehicles. The first one has, approximately, the position (897, 333) and velocity (-3, 0.23). An actual .txt example can be found in the appendix section.

The user can edit a saved file or write an entirely new one and then load it into the simulation using the 'File' -> 'Load' menu. This will remove all the vehicles previously inside the simulation. All values should be integers. Value for speed and detectionRadius should be above zero, values for other parameters should be non negative, and vehicle coordinates should be within the width and height of the simulation. If incorrect values are given, loading fails and the opens a pop-up window with help tips. If a vehicle is added with position outside the simulation, it will be placed at a random position within the simulation. This is done to avoid issues with window resizing. For example if the simulation is saved at width of 1500, then window is resized to width 800 and the same file is loaded, then vehicles that are outside the new border will simply be added to random positions.

Testing

Overview

The program has unit tests for the SimulationVector class, functional test for vehicles and their SimulationPanels. Additionally, the program's GUI and I/O have been tested manually.

Details

During development, the first thing implemented was a rough GUI, so that any additional components could be tested visually. Thus, creation, movement and interaction of vehicles and obstacles was initially tested visually and through the Scala interpreter.

Unit tests were implemented for the SimulationVector class using the junit-4 testing library. The tests assert that operations like normalization, addition and multiplication work correctly for a range of values.

Functional tests were used for Vehicle and SimulationPanel classes using junit-4 testing library. The tests assert that vehicles and obstacles are added and removed from the panel, move correctly and calculate 3 main rule vectors correctly, check that vehicles added outside simulation boundaries are handled correctly, and are capped in their top speed. All the tests pass 100% of the time. The test can be run by right clicking on the tests package and selecting ‘Run As’ -> ‘Scala JUnit Test’.

Additionally, I/O has been tested by saving files, changing the simulation state and loading them again, asserting that the state returned to the same one as at the moment of saving. Loading files were written manually and tested. Saved files have been altered and loaded. Incorrectly formated files have been tested.

Interestingly, the program achieved somewhat realistic flocking behavior around the middle of the development process. At late stages, it was discovered that the program breaks at speed 3 for an unknown reason, and works on all other speeds. After a thorough debugging process across all the classes, a tiny issue with the normalization of vectors has been discovered. After removing half a line of code, the whole simulation not only started working at speed 3, but also displayed a much more realistic and lifelike behavior, started supporting vehicle counts upwards of 500 vehicles and obstacle avoidance started to work correctly. If speed 3 was not tested during I/O tests, it would have never been discovered that the simulation was suboptimal.

Known bugs and missing features

Overview

The program contains a number of minor shortcomings.

Details

- *Simulation slows down, v at high simulation counts:*
  Above 1000 vehicles, the simulation slows down when vehicles get close to each other. Threads did not help much with this; however, If threads were implemented more thoroughly, simulation could possibly handle higher vehicle counts.

- *GUI might not work properly on Linux:*
  The program was developed on Mac OS, and when tested on a linux laptop, vehicles left traces behind them. This could be fixed by testing and debugging across different platforms or switching the GUI library.

- *At minimum window size, GUI is not displayed well:*
  When the window is resized to small sizes, some of the controls are hidden. This could be fixed with optimization of the ControlsPanel class.

- *High speeds cause unnatural behavior:*
  At speeds above 6 vehicles sometimes rush past each other without adjusting. On the other hand, this might happen in the real world, if birds were to fly at 400 km/h.

- *Vehicles pass boundaries by a little:*

When a vehicle reaches the simulation boundary, because of its velocity, its position reaches a little past the boundary, for example to values like -0.3 on the x axis. If the simulation state is saved at such a moment and then loaded, then the method addVehicle of SimulationWorld adds this vehicle at a random position within the simulation instead. This could be fixed by adjusting the move() method of the Vehicle class.

- *Magic numbers*
  There are a few ‘magic numbers’ across the program. They can be replaced with new variables in the Parameters object to improve readability and customizability.

- *Slider values*
  It is possible that with loading/change of presets some slider values do not visibly change. This could be fixed by debugging the GUI classes.

- *Obstacles and teleportation*
  Right after a vehicle teleports across the edge of the simulation, it can not detect obstacles right away and sometimes passes through them. This could be fixed by further developing the obstacle avoidance logic.

- *Passing through obstacles*
  Rarely, vehicles not at the edge of the simulation pass through obstacles. This is caused by the simplistic implementation of obstacle avoidance, and could be fixed by developing the algorithm further

3 Best sides and 3 weaknesses

3 best sides

- *Extensibility:*
  Program structure allows for easy addition of new features, as it is split into the GUI and logic parts which represent life-like objects. For example if a new class of ‘Hunter’ vehicles was to be added, it could simply extend the vehicle class and fit inside a SimulationWorld. The program is not a final thing and can easily be developed much further.

- *Code readability:*
  For the most part, code is rather intuitive and not cryptic, so if someone had the task of understanding the program logic in a short period of time, they could do it without too much effort.

- *Ease of use:*
  Program’s controls are more or less intuitive and don’t require a long time to get used to. The window can be resized, stretching the borders of the simulation.

3 weaknesses

- *Simplification:*
  Simulation logic leads to simplistic behavior that resembles that of real life birds or fish, but only up to a limit. Swing GUI limitations sometimes make vehicles overlap.

- *Scalability:*
  The program would not work for simulating, for example, 2000 vehicles, as it substantially slows down at high vehicle counts. This could possibly be fixed with better threads or refined logic.

- *Cross-platform use:*
  The program has been tested on 2 apple computers, but shows problems on Linux and has not been tested on Windows. It requires an IDE and java SDK installed in order to be launched.

Deviations from the plan, realized process and schedule

Overview

The project loosely followed the initial schedule. First, the topic of 2d simulations was researched, a simple GUI was developed, logic was implemented and tested, and then the project was polished.

Details

The initial stages of the project were the most difficult, as research had to be done on an unfamiliar topic and a GUI library learned. There were many issues with finding good material about the scala swing library as it is not widely used. At some point, scalaFX library was tested, but it also had little material available and required a steep learning curve. This process took the initial 2-3 weeks, as anticipated in the plan.

The middle part of the project went slowly: not much work was performed, but the key classes and logic were roughly implemented as planned. Deviating from the plan, threads were not implemented until late into the project. Obstacles were also implemented at the latest stages.

Most work has been done in the 3 weeks leading up to project submission, including optimization of vehicle behavior, debugging and documentation, program restructuring for separation of gui and logic and development of additional features.

Final evaluation

Overview

The project turned out decent and appears to meet the criteria, but it nevertheless does not reach the level of simulations used in real life. Vehicles display flocking behavior and are a little hypnotizing to look at, but behavior is simplified. The program could definitely be developed further with addition of more realistic behavior, new features that make simulations more interesting and better scalability. Luckily, the program structure enables such extensions.

Details

As mentioned in the previous sections, the program has a number of well-working features as well as a number of shortcomings. If those will be addressed, it could actually become an interesting tool for analysis of real life flocking behavior. The program runs smoothly and is not difficult to use. Nevertheless, it is not on the same level as commercial applications.

Scala language seems like a good choice for the project, as it is scalable for large volumes of data, but a better GUI library could be chosen. The inner logic seems to work fairly well, but could be developed further for more realistic behavior. Some of the Array data structures could be replaced with more optimal choices for example. Additional features such as ‘predator’ and ‘prey’ vehicles, selection of individual vehicles to see their parameters and more could be developed.

If the project would have started anew, a similar path would have been followed. One difference is that a better GUI library could have been chosen. Other than that, much has been learned along the way about object oriented programming, class structure and more, so the journey was certainly worth it.

Appendixes

Files inside the zip archive include:

1. The Eclipse project
2. Project document
3. App GUI screenshots
4. Examples of saved files
5. UML diagram of program structure

References

Overview

The project is based on Craig Reynold's research and features techniques on character behavior from several sources. The project has been developed based on knowledge learned in the Programming 2: studio course, with topics like threads and graphical user interfaces.

Details

[1] Reynolds, Craig W. *Steering Behaviors For Autonomous Characters*, Sony Computer Entertainment America , www.red3d.com/cwr/steer/gdc99/.

[2] Reynolds, Craig W. *Boids (Flocks, Herds, and Schools: a Distributed Behavioral Model)*, Sony Computer Entertainment America , www.red3d.com/cwr/boids/.

[3] Bevilacqua, Fernando. "Understanding Steering Behaviors: Collision Avoidance." *Game Development Envato Tuts* , 13 May 2013, gamedevelopment.tutsplus.com/tutorials/understanding-steering-behaviors-collision-avoidance--gamedev-7777.

[4] Weisstein, Eric W. "Perpendicular Vector." *From Wolfram MathWorld*, mathworld.wolfram.com/PerpendicularVector.html.

[5] "Lesson: Using Swing Components." *Lesson: Using Swing Components (The Java™ Tutorials > Creating a GUI With JFC/Swing)*, docs.oracle.com/javase/tutorial/uiswing/components/index.html.

[6] "Trail: 2D Graphics." *Trail: 2D Graphics (The Java™ Tutorials)*, docs.oracle.com/javase/tutorial/2d/index.html.

[7] "Programming 2: Studio" Course 2020, Aalto University.