Flocking system mimicking fish school behaviours

MSc Computer Animation and Visual Effects
Bournemouth University

Luke Bazalgette
i7902033

August 2016

Table of Contents

Abstract

In this thesis, a programmable flocking system was developed. The system is designed to emulate the movement patterns of fish in a body of water. Previous academic research into the field is reviewed and analysed through a critical lens, as a means of informing the design of the software. What follows is an extensive review of the implementation process, providing details on code structure, languages and libraries used to assist development. The resulting simulation is capable of interpreting flock behaviours through the use of a high level scripting environment which can be modified at run time. In order to demonstrate the functionality of the simulation, several scripts were produced. The default script emulates the behaviours of fish when confronted with danger through the formation of a defensive bait ball; the ball alters its shape based on its proximity to a moving point described as a predator. Upon analysing the results of the implementation, a reciprocal velocity based flocking system is proposed and the results of the implementation are presented and discussed.

Keywords: Baitball, scripting, agent

Concepts: Reciprocal velocity obstacle, Reynolds flocking, Dynamic scripting

1 Introduction

This thesis describes the development process of the underwater flocking system. This section introduces the problems associated with developing such a system and the proposed solution. Existing research and the development of the solution are discussed in later sections.

In nature, some intelligent species form aggregations with the intent to increase chances of survival. These groups are given different names based on their species and the nature of the social group. In computer science, a congregation of intelligent agents is known as a flock as defined by Reynolds (1987). A flock exhibits the illusion of motion alignment, position cohesion and minimum separation. When applied to the field of entertainment, this typically refers to the simulation of large crowds of virtual actors. These systems are commonly employed when constructing a large scale battle or a herd of animals. This approach is effective when constructing a rehearsed scene and when assuming that the agents have a set path or environment to follow. However, there are few examples of flocking behaviours being applied to dynamic real time environments. In nature, an intelligent being would be expected to react to changes in the environment around it.

Although real-time flocking systems have been designed to simulate birds, packs of predators or groups of people, there are fewer examples of flocking systems concerned with modelling the behaviours of groups of aquamarine life. Although birds act in a similar manner due to their airborne nature, fish have developed unique evolved behaviours and movement patterns. The denser atmosphere of their environment is also a factor when modelling school movement patterns.

This thesis aims to explore the nature of the movement of fish through the creation of an intelligent system which is capable of running in real time and outputs simple vector values representing the position of its agents in three-dimensional (3D) space. As the reactions of the flock are expected to be dynamic, an efficient means of controlling them is also required. Through the use of basic conditions, the emergent behaviour should exhibit properties close to its corporeal reference.

2 Related Work

2.1 Fish behaviour

In order to effectively model the behaviour of fish in a simulated environment, direct visual observation is necessary. When considered as a collective, a group of fish that stay together for social reasons is known as a shoal. If the collective shows deliberate and structured movement, it is described as a school of fish. Schools are typically made of fish of a similar size or species and are formed out of survival instinct, increasing chances of finding food, reproducing and avoiding danger, supporting Reynolds' observations. Lauder (2008) notes that fish operating as a school gain hydro-dynamic advantage, reducing the cost of locomotion using the wakes created by their neighbours, directly mirroring Reynolds' theory of cohesion in a flock.

Baitballs describe the natural phenomenon in a school of fish form a spherical vortex in an attempt to deter predators from approaching them. When observing baitballs through visual documentation; they are prone to change shape, divide and separate in reaction to stimuli. Although tightly packed, the fish in a baitball maintain minimum distance from each other and reciprocate the forces of their wake to neighbours.

2.2 Flocking systems

A great deal of research into the field of modelling natural flocking patterns in virtual space derives from the paper Flocks, Herds and Schools: A Distributed Behavioural Model by Reynolds (1987). The flocking system devised by Reynolds is elaborated from a particle system; each agent in the flock is treated as a particle. The overall behaviour of the flock is dictated by the algorithm itself. The behaviour of the flock is dictated by simple movement scripts, the environment being navigated and the laws of simulated physics also influence the movement of the agents. Reynolds notes that one common element among all natural flocks is the impression of a centralised control among all the individual members. One might argue it is merely the result of co-dependency between the members of the herd.

Reynolds describes an Actor in a flocking system as such:

> "An actor is essentially a virtual computer that communicates with other virtual computers by passing messages." (Reynolds, 1987)

Each actor has its own set of computational abstractions which handles process, reasoning, procedure and states. Actors in flocking systems are capable of organising themselves and determining their own objectives. The modern term for a member of a flock is the word Agent. Cunningham and Cunningham (2010) compare and contrast Actors and Agents. Actors will only recognise an obstacle or fellow Actor in a simulation if explicitly wired to a framework. Although Actors can exhibit greater diversity and functionality than Agents, the continuous nature of the latter is more suited for creating naturally evolving simulations and assessing behaviours at runtime.

A common means of simulating this effect of agent awareness is through the use of continuous fields. The continuous field method assesses the distance between two individual agents and applies a dampening force to each agent, preventing a collision. According to Schwab (2011) this derives from a state space style of AI design, in which agent behaviours are isolated and predicable, as a result the main drawback of this method is its inability to take into account the agent’s surroundings and appropriately navigate. van den Berg, Patil, Sewall, Manocha, and Ling (2008) believe that the navigation problem can be solved by selecting a new velocity for each agent in each cycle.

> “Our goal is to choose this velocity such that the agent will not collide with other agents moving in the same environment. This is a challenging problem, as we only know the current velocities of the other agents, and not the future ones. Also, the agents are not able to communicate to coordinate their navigation.” (van den Berg et al., 2008)

2.3 Reciprocal velocity obstacles

One solution considered is to linearly extrapolate the velocity from neighbouring agents to predict future motion. This approach selects a new velocity for the agent AI and implicitly assumes that all other agents use similar reasoning. This means that the effects of collision avoidance are halved to compensate for the reasoning capabilities of the other agent. This is known as Reciprocal Velocity Obstacles (RVO). This Reciprocal Velocity method guarantees to avoid oscillatory behaviour of the agents. The reciprocal velocity of agent A to agent B is defined as the set of velocities available to A that will result in a collision with agent B at some point in time given their present velocity. This means collision can be avoided should both agents choose a value outside their RVO, provided both employ the same algorithm.

Snape, Guy, Lin, Manocha and van den Berg (2012) followed up their work with an approach to multi-agent comprised of two levels, one dealing with goal path planning, the other addresses local collision and avoidance navigation. Other agents are not considered when planning the initial path to the goal. Only large objects like buildings and mountains. Each of the agents probes a constant number of candidate velocities and evaluates them against the RVO of the previous agents. The entire computation is reduced to a linear runtime by only selecting a restricted subset of nearby agents.

Snape and Manocha (2012) use RVO as a means of simulating autonomous aircraft (Figure 2.3.1). Their solution aims to build on van den Berg's previous solution by factoring 3D space into the equation. The kinematic and dynamic model was simplified to simulate vehicle control, without the ability to reverse. Variables such as altitude and speed were not fixed and allowed to vary continuously. This three dimensional approach is similar to a conceptual system of modelling fish behaviours underwater.

Previous models for collision free navigation were typically limited to 2D space and did not consider kinematic and dynamic restraints in their motion. This approach extends the Optimal Reciprocal Collision Avoidance (ORCA) algorithm to include 3 dimensional workspaces and performs local collision avoidance for polygonal objects. The agent behaviour is inspired by the Dubins (1957) car, which is constrained to forward motion with a fixed speed. Although these constraints do not apply to fish, the theory of object avoidance still stands.

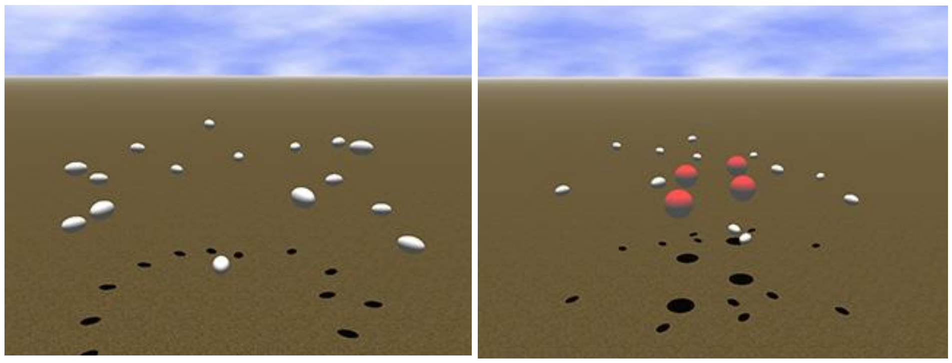

Figure 2.3.1 Snape and Manocha's simulated airplanes**.**

Whereas the previously discussed solutions are concerned with controlling agents following their own set paths, they do not exhibit group behaviour.

3 Software design

This section presents a breakdown and discussion of the solution design and a summation of the underlying mathematics.

3.1 Class structure

For ease of development, each component of the program contained within a dedicated class. Values are passed between classes through the use of includes and class pointers. Directly divorcing each component enables reuse, porting and readability.

3.1.1 NGLScene

The NGLScene class (Figure 3.1.1.1) acts a simple means of visualising the simulation and managing update frequency. As all values returned from Flock are stored as vector arrays, it is possible to loop through each element and draw an object at each vector. This class inherits from the QOpenGLWidget library.

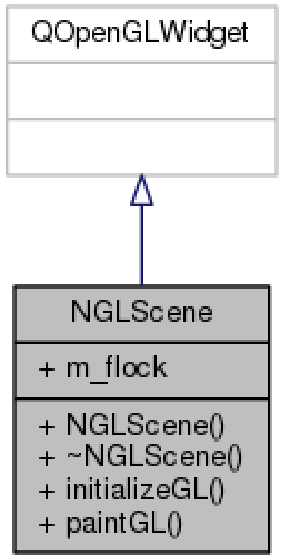

Figure 3.1.1.1 NGLScene inheritance diagram

3.1.2 LuaInterpreter

LuaInterpreter (Figure 3.1.2.1) contains the simulation object and get set functions designed to control the behaviours of the agents in the simulation. The class uses a command pattern, where functions are called by the simulation when needed. By decoupling the functions in a separate class, this reduces the amount of code used in the class bridging Lua to the Cpp environment.

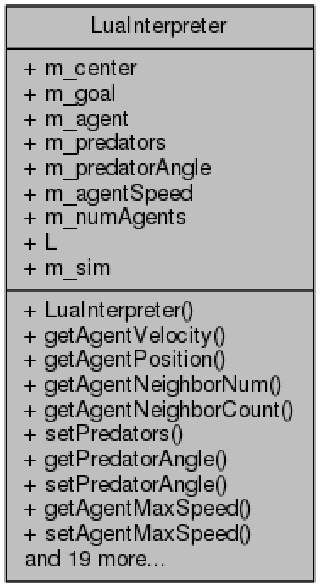

Figure 3.1.2.1 LuaInterpreter UML Diagram

3.1.3 Flock

The simulation is then initialised inside the Flock class (Figure 3.1.3.1). Flock sets the default state of the simulation and is responsible for spawning agents and dictating the program loop. The external Lua script is called from this class and executed as part of the setPreferredVelocities function.

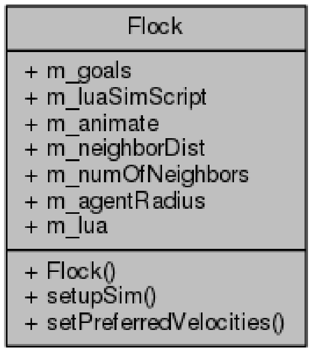

Figure 3.1.3.1 Flock UML Diagram

As Lua does not directly support Cpp functions, a global variable is set pointing to LuaInterpreter. This enables Cpp functions to be called through Lua via the 'interpreter:' prefix. Lua needs to be told what do execute when a function is called; the LuaFunctions class facilitates this. The class defines a meta-table, from which functions derived from the LuaInterpreter are registered with the Lua environment. As LuaFunctions requires access to functions from the simulation class, a static pointer was created within Flock giving the program access to otherwise unavailable functions. From there, a bridge between the Lua stack and cpp is created, meaning values can be passed in and out of the program as if operating as a single entity. The simulation is then directly controlled by Lua.

3.2 Interface

To increase usability and provide a simplified method of interacting with the solution, two graphical user interface (GUI) classes were created (Figure 3.2.1). The UserInterface acts as a main window for the solution and is responsible for creating the OpenGL instance from which simulation data is rendered. Users can directly interact with the simulation from this window. The performance of the simulation can be adjusted through the manipulation of agent attributes. The resources used by the simulation scale linearly with the attributes. It is also possible for users to load their own scripts into the simulation through the use of a file dialog. The file dialog appears on launching the program and provides an interface for locating, editing and writing Lua scripts locally.

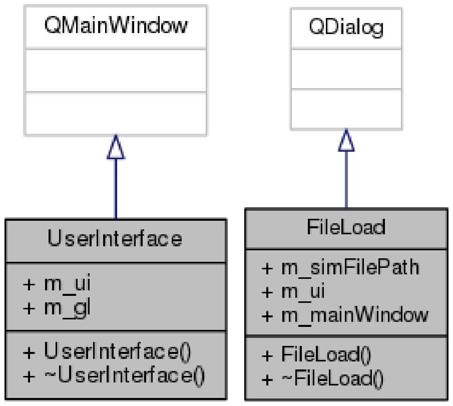

Figure 3.2.1 UML Diagrams of User Interface classes

3.3 Application

To demonstrate the capabilities of the simulation, several behaviour scripts were produced. The equations used to dictate the behaviours can be found in the appendices.

3.3.1 Bait ball

The default script which is loaded on start-up is designed to emulate the effects of a bait ball (Figure 3.3.1.1). The velocities of all active agents in the scene are set to orbit the centre of the swarm by default. Should the flock come too close to a point defined as a predator; the agents within the radius of the predator will retreat to a safe distance whilst maintaining the bait ball. If a predator enters the swarm, the agents will abandon formation and retreat away from the predator.

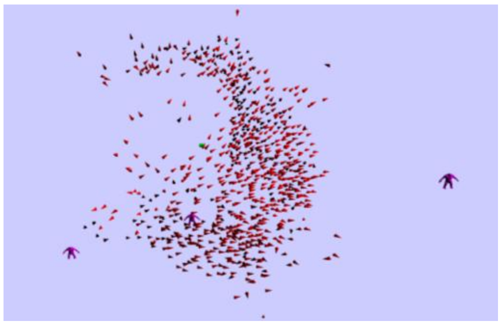

Figure 3.3.1.1 Bait ball script

3.3.2 Agent flow

All active agents follow an oscillating path. This is intended to mimic the effects of mass migration along a sea current (Figure 3.3.2.1).

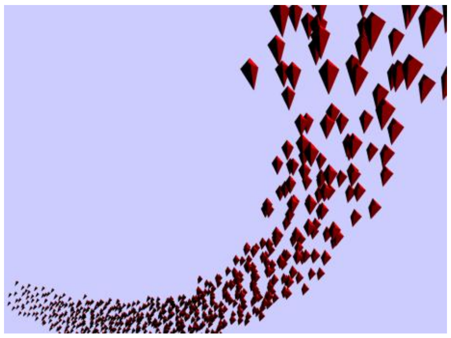

Figure 3.3.2.1 Agent flow script

3.3.3 Feeding effect

This script was the result of an unsuccessful attempt to recreate the obstacles seen in the work of Snape et al. Agents will be drawn to points along a set path, forming independant bait balls, not unlike fish school feeding patterns. Should an area become too crowded, the fish will move on to the next feeding point (Figure 3.3.3.1).

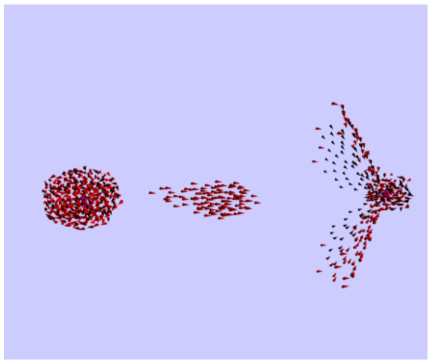
Figure 3.3.3.1 Feeding effect script

3.3.4 Fish tornado

This script creates a large vortex around a given point in space. All active agents will eventually be drawn in to form a single pillar. The agents will oscillate through the body of the pillar from their point of origin.

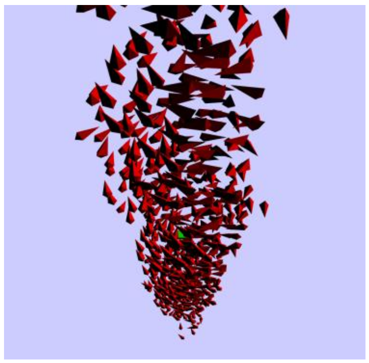
Figure 3.3.4.1 Agent tornado script

3.3.5 Goal tracking

This simple script demonstrates the behaviours discussed by Reynolds, where fish will move to different goals as a group whilst maintain alignment and cohesion (Figure 3.3.5.1). When the collective flock reaches the goal, the position will be set to a new random value.

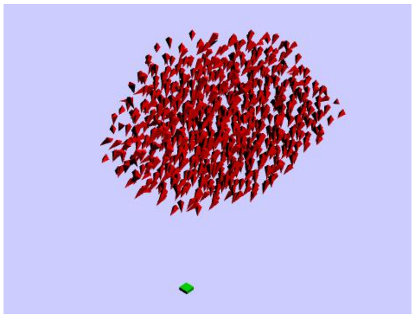
Figure 3.3.5.1 Bait ball script

3.3.6 Opposite goals

This script isolates two groups of agents and assigns adjacent start positions and goal vectors. This causes two separate streams of agents to cross paths (Figure 3.3.6.1). When assigned differing max speeds, the stream containing the slower agents will be bisected and have members carried away by the faster agents, demonstrating the effects of the RVO algorithm.

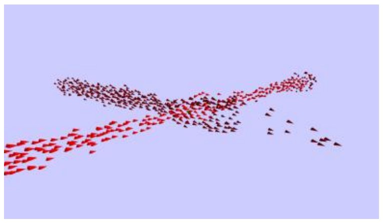

Figure 3.3.6.1 Bait ball script

3.3.7 Script documentation

This file contains a full list of compatible functions for use in Lua. This is intended to act as a reference for the development of new scripts.

3.3.8 Starting template

This file contains a basic script with commonly used functions. When creating new scripts it is advised that this file structure be used as a starting point.

When assigning values to each agent in the simulation, a loop is created inside the script with a number ranging from 0 to a figure set by the user. This sample number acts as a constraint for how many agents to assess during the execution of the script. Increasing this number results in a linear increase in program execution time, but returns an accurate result. Through testing and revision it is noted that past a certain sample point, there is a negligible change in the behaviour of the flock.

3.4 Libraries

A selection of public code libraries were used to assist the development of the solution. This section details their function and application to the solution.

3.4.1 NCCA Graphical Library (NGL)

NGL provides an OpenGL based solution for drawing the objects in the solution in a visual format. Each point value returned by the simulation is represented by an object on screen. This served well during the debugging process and provided a direct visual reference of the flock behaviours.

3.4.2 RVO2-3D

The RVO2-3D library provides existing functionality with regards to the reciprocation of agent velocity. It was devised by Snape and Manocha (2010) as a platform for developing new methods of navigation for autonomous agents using the RVO method. This makes it a logical choice for procedurally modelling the movement of fish. The RVO2 library computes the OPC for each agent and intersects the half planes to form a region of permitted velocities for the virtual agent. The HRVO algorithm is similar but uses a clear path geometric algorithm instead when computing the new velocities. The RVO is not considered when avoiding local collisions. This is to eliminate agents which are too far way to affect the velocity and path of an agent, significantly reducing the running time of the algorithm. The agent will only consider agents it can immediately see. The first step of integration is to compute a preferred velocity for each agent $\boldsymbol{v}^{pref}_{i}$. This vector has a magnitude equal to the preferred speed of the agent in the direction of the next node along the agent's global path to its goal. If the agent is close to the goal a preferred velocity is set to the NULL vector.

The new velocity $\boldsymbol{v'i}$ selected for agent Ai will ideally be as close to $\boldsymbol{v}^{pref}_{i}$ as possible whilst remaining outside the RVO of Ai and inside the admissible new velocities. This may cause the environment to become crowded with agents, meaning the RVO is populated with a full set of admissible velocities, causing a deadlock in agent movement. In order to address this, the algorithm selects a velocity within the RVO but will be penalised for this choice.

The factor wi can vary from agent to agent, reflecting differences in aggressiveness and shyness. The time to collision can be easily calculated based on RVOs. A new velocity can be selected with minimal penalty among all velocities. This minimum is approximated by sampling the number N of velocities evenly distributed over a set of admissible velocities. The structure of the program guarantees that the agent will assess a new velocity should the choice be unsafe. As the selection of neighbours is imperative for the functionality of the solution the orientation of the agent is factored into the selection. With this considered, the orientation of the agent is updated in each cycle by inferring it from the motions followed by the agent.

In most cases the orientation of the agent is determined by the velocity vector. However to deal with crowded situations, the orientation is computed as a weighted average of the preferred velocity and the actual velocity.

3.4.3 Luawrapper

The Luawrapper class was created by Ames (2014) as a means of executing Lua scripts within a Cpp environment. The wrapper is capable of feeding common variable types to the Lua script as well as returning them back to the CPP environment. The class is easily extendable by way of the Luawrapperutil which allows the user to specify new variable types to be read in from Lua. For the purposes of the simulation, a Vector3 template was created as a means of reading float values in and out of the Lua script with correct labels. When passed into Lua, the Vector 3 is treated as a table; the x, y and z values can be operated on as if they were floating point numbers. New Vector3 tables can also be defined within Lua by structuring a table with X, Y and Z axes.

When using Lua within Cpp, values can be passed in by pushing them onto the Lua stack. Each slot in a stack can hold any given value, including strings, tables and entire functions. Values can be retrieved from the stack in Cpp and assigned as variables. Typically, within the solution any values retrieved from the stack are passed as arguments for functions in LuaInterpreter. These functions are either used to update an aspect of the simulation or to return a new value back to Lua.

3.5 Performance

As the program continually assesses the simulation to determine the new velocities of its active agents, the simulation script is run multiple times to produce accurate results for each returned frame. Some values are updated independent of the agents, such as the predator positions; if updated at the same rate as the agents, a dissonance is created between the position of the model representing the predator and the values in use by the simulation for calculating the new velocities of the given agents. This provides enough time for the agents to assess the environment and respond appropriately. The length of time to calculate the frame is based on a number of conditions, the first being the number of agents being sampled in a given loop through the script. As this number increases so too does the time required to calculate a frame. This time is scaled according to how many agents are given at that point in the simulation. As the ORCA method in use in the library only considers obstacles within a set view of the active agent, an effect on performance is only notable when all active agents are closely congregated together.

3.6 Mathematics

3.6.1 Velocity

The basis for flocking systems comes from calculating the change in position of a given agent on each update executed. This employs the real world principal of velocity. The velocity of an active agent is described in Figure 3.6.1.1.

$$v_i = p_i / \Delta t$$

(3.6.1.1)

Where ***p*** is the three-dimensional vector representing the agent's point in 3D space and Δt represents the time frame of change.

3.6.2 Reciprocal velocity obstacle

The RVO algorithm is used by the simulation when assessing the positions and the preferred velocities of each agent in an individual update. The $RVO^i_j(v_j, v_i)$ of agent ***j*** to agent ***i*** is defined as the set of velocities (v_i) available to Ai that will result in a collision with agent ***j*** at some point in time given their present velocity. It is defined by van den Berg et al. (2008) for an agent ***i***, and another agent ***j*** which is regarded as a moving obstacle maintaining its current velocity v_j (Figure 3.6.2.1).

$$RVO^i_j\ (v_j, v_i) = \{v\grave{}_i \mid 2v\grave{}_i - v_i \in VO^i_j\ (v_j)\}$$

(3.6.2.1)

Where $VO^i_j\ (v_j)$ is the list of available velocities for agent ***j***. The agent is required to pick a velocity outside of this set to avoid a collision. van den Berg describes the Velocity Obstacle equation in Figure 3.6.2.2.

$$VO^i_j\ (v_j) = \{v_i \mid \lambda(p_i, v_i - v_j) \cap j \oplus -i \neq \emptyset\}$$

(3.6.2.2)

The solution used by Snape et al. makes use of the alternative HRVO (Hybrid reciprocal Obstacle Velocity Obstacle) method. This variation takes into account that an object is already in constant motion. The velocity obstacle of active agent ***i*** with a position p_i induced by a moving obstacle ***j*** with position p_j, is the set of all velocities that will result in a collision between the agent and the moving obstacle within a short time frame, assuming that the dynamic obstacle maintains constant velocity v_j.

The HRVO method is defined in Figure 3.6.2.3.

$$HVO^i_j\ (v_j) = \{v_i \mid \exists t > 0 :: t(v_i - v_j) \in D(p_j - p_i, r_a + r_b)\}$$

(3.6.2.3)

Where ***D(p,r)*** is an open disc of radius ***r*** centred at ***p***. It follows that if agent ***i*** selects a velocity within the ***VO*** region, a collision may potentially occur.

3.6.3 Constraints

The agents are also subject to Kinodynamic constraints in addition to Geometric ones. When acting together, this restricts the set of admissible new velocities for the agent, given its current velocity $\boldsymbol{v_i}$ and possibly the orientation $\boldsymbol{\theta_i}$. This set is denoted in Figure 3.6.3.1.

$$\boldsymbol{K(v_i,\ \theta_i,\ \Delta t).} \tag{3.6.3.1}$$

It may have any shape depending on the nature of the agent. If maximum values are set for velocity and acceleration, the velocities available are shown in the results of Figure 3.6.3.2.

$$\boldsymbol{K(v_i,\ \theta_i,\ \Delta t) = \{\grave{v}_i | \, \|\grave{v}_i\| < v^{max}_i \wedge \|\grave{v}_i - v_i\| < a^{max}_i \Delta t\}} \tag{3.6.3.2}$$

The variable $\boldsymbol{a^{max}_i}$ assumes that the virtual agent has a maximum acceleration constraint. Complicated constraints may restrict the set of admissible values with regard to the orientation; this would prove useful for simulating car mechanics and kinematics.

3.6.4 Optimal reciprocal collision avoidance

This avoidance method addresses the oscillation potentially caused by HRVO. This method was introduced by van den Berg (2011) as a means of overcoming the condition dependant limitations of collision avoidance. This augments the velocity obstacle with a half-plane that defines a set of velocities that are both collision free and guarantee smooth agent motion outside of dense scenarios.

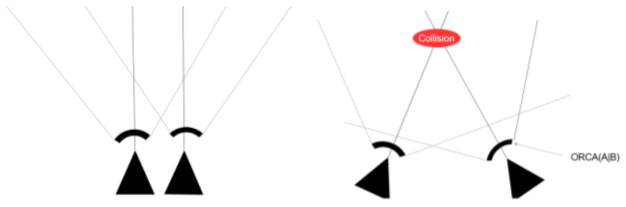

Figure 3.6.4.1 UML Diagrams of User Interface classes

Figure 3.6.4.1 shows a set of ORCA lines defining the minimum and maximum values and a third showing the agent's current velocity. The point of collision is shown where the current velocity ORCA lines intersect.

Each agent must re-sample their velocities until the lines are no longer intersecting, avoiding a collision. The RVO is not considered when avoiding local collisions. This is to eliminate agents which are too far way to affect the velocity and path of an agent, significantly reducing the running time of the algorithm. The agent will only consider agents it can immediately see (Figure 3.6.4.2).

$$ORCA^{i}_{j}=\{ v \mid (v-(v_i + 0.5u)) \,.\, n \geq 0\}$$

(3.6.4.2)

In this equation devised by Snape, Guy and van den Berg (2011), U represents the vector of from the relative velocity $v_i - v_j$ of the agents in motion to the closest point of the active agent's RVO boundaries (Figure 3.6.4.3).

$$u = (min(v \in VO^{i}_{j}) \mid\mid v - (v_i - v_j) \mid\mid) - (v_i - v_j)$$

(3.6.4.3)

n describes the outward normal of the boundary p_f the velocity obstacle at $va-vb + u$. With that u becomes the smallest change required to the velocity to avoid collision. When both agents are capable of avoidance this means that each agent needs to adjust its path by $u/2$. Therefore permitted velocities are in a half pipe shape relative to n beginning at $v_a+(u/2)$.

3.6.5 Bait ball

In order to form a bait ball formation, the equation in Figure 3.6.5.1 is used.

$$p_n = cos(\theta) * (p_x - p_{origin}) - sin(\theta) * (p_x - p_{origin}) + p_n$$

(3.6.5.1)

Where n is an axis of the agent's position space and θ is the angle set each update to the algorithm. When applied as an update to velocity rather than the position of the object, the equation becomes the Figure 3.6.5.2.

$$v_i(x) = cos(\theta) * (v_x - p_{origin}) - sin(\theta) * (v_z - p_{origin}) + v_x$$
$$v_i(y) = sin(\theta) * (v_x - p_{origin}) - sin(\theta) * (v_z - p_{origin}) + v_y$$
$$v_i(z) = sin(\theta) * (v_x - p_{origin}) - sin(\theta) * (v_z - p_{origin}) + v_z$$

(3.6.5.2)

3.6.6 Reynolds behaviours

These equations are hard coded into the Cpp environment rather than scripted. Cohesion between agents describes the behaviour of a flock that attempts to stay as close to the centre of the collective as possible. The cohesion equation is described by Hartman and Benes (2006) in Figure 3.6.6.1.

$$c_i = \sum_{\forall n \in v_{j,\ldots n}} (p_j/m)$$

(3.6.6.1)

Where ***m*** refers to the number of agents recognised as neighbours of Agent i. Alignment ensures that all active agents maintain a uniform velocity with its immediate neighbours while travelling in a linear fashion. The steering equation is defined in Figure 3.6.6.2.

$$m_i = \sum_{\forall n \in v_{j,\ldots n}} (v_j/m)$$

(3.6.6.2)

Rather than use a script method, the constant nature of these steering behaviours and the complex algorithm required, meant they are better suited as functions in C++ rather than encoding them directly in Lua. These values are then called into the script for the purposes of calculating new velocities.

3.7 Functionality

Provided that the code is valid, the solution is able to run any given Lua script. If an invalid script is loaded, the program will default to a basic simulation whereby agents will swarm to the centre of the 3D environment before having their velocities nullified. Although a Lua debugger would have been ideal, the limitations of the platform and the use of unique Lua functions would have been problematic when integrating existing solutions. The functions made available to Lua provide high levels of control over the behaviour of the simulation without the need for extensive knowledge of object orientated programming languages.

The Cpp environment has been structured in a way which allows developers to easily extend the system and add new functionality by following the existing structure. All significant functions are appropriately named and commented. An external developer should be able to add new functionality to the simulation by adding to the LuaInterpreter and LuaFunctions without needing to interact with any other components.

3.8 Bugs and limitations

Due to the nature of the timer event in NGL, the script is reviewed multiple times during a single frame; as a result any functions which return values to an array in the Cpp environment are called multiple times resulting in overpopulated arrays. The solution to this issue relies on a check back to the program checking of the vector pointer is already populated. If so the function is not called.

One significant bug that manifests when compiled through Qt is a broken File Dialog box that appears when an OpenGL scene is active at the same time. This can be avoided by deactivating the OpenGL window while it is open. However the objective of the solution is to provide an effective simulation. Rather than retrieving the file path string using a Dialog box, an alternative method whereby the user can manually enter the string of the file to be loaded was included instead.

4 Conclusions

The final solution produced provides a scriptable, intelligent flock of agents, simulating the behaviours of real-world aquamarine life. The script syntax is friendly to users whom have previous experience working with scripting languages and provides examples as a template for other users to follow. Debugging can be performed at runtime by modifying the external Lua script directly, simplifying the interface, minimising development time and assisting in the development of complicated algorithms. Objects available to the users include flocking agents derived from the RVO2-3D Library, a goal point which can be manipulated by both script and user using the middle mouse. The data returned by the simulation is easily applicable to other systems and can easily be integrated into other systems if needed.

Additional milestones achieved include a graphical user interface in which users can adjust the default variables of the agents at run time; optimising the algorithm and further refining the behaviours. Users are able to load and manipulate their own external scripts through the use of an external dialog box, providing all the tools required to control all open aspects of the simulation. To better demonstrate the functionality of the simulation, a collection of scripts showcasing possible behaviours were produced. Each script aims to show a different aspect of fish behaviour, such as a bait ball and following a current flow. The movement exhibited by the agents in the final result can be closely compared to the behaviours of real world aquamarine life. As with real fish, agents exhibit avoidance which can be described as frenetic and reactive. They will exhibit formation even without explicit instruction from the script. In some cases, the fish show some unnatural movement such as orientation and velocity snapping rather than smooth steering. This may be attributed to the active script rather than the simulation itself.

Some planned components and features were not implemented in the final solution. A dedicated Obstacle class intended to handle the ability for agents to navigate a static environment; this was abandoned as the functionality of the RVO2-3D library did not provide the necessary logic for calculating the adjustments in agent velocity. Instead, a similar approach was used to simulate the proximity of danger within the Lua script itself, albeit not as complicated. The flocking system could be improved through the use of a class designed to read in geographical data to act as constraints on the velocities available to active agents. Agents would respond to this data as if navigating through a coral or rocks. By decoupling the classes dedicated to handling the simulation and the interpretation of Lua functions, the solution could be converted into a plugin for visual effects software, renders, game engines and other computer generated animation platforms. As the solution is specialised to simulate aquamarine life, further study may include similar systems for other types of flocks. This may include birds, insects and mammals, each with their own subtleties and simulated behaviours. As the initial logic of flocking systems derives from particle systems, a similar solution could be used for simulating common particle effects such as rain and other weather phenomena. This is feasible given the point based output of the simulation.

Acknowledgements

Elias Daler for advice on implementing Lua with Cpp.
Alex Ames for assistance with Luawrapper.

References

AMES, A., 2014. *LuaWrapper.* [library]. Available from: https://bitbucket.org/alexames/luawrapper/overview [Accessed 10 August 2016]

CUNNINGHAM AND CUNNINGHAM INC. 2010. *Actor vs agent*. Portland: Cunningham & Cunningham. Available from: http://c2.com/cgi/wiki?ActorVsAgent [Accessed 10 August 2016].

DUBINS, L. E., 1975. *On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents*. American Journal of Mathematics. 79 (3). 497–516.

HARTMAN, C. AND BENES, B., 2006. Autonomous Boids. *Computer Animation And Virtual Worlds* [online], 17(3-4), 199 – 206. Available from: http://onlinelibrary.wiley.com/doi/10.1002/cav.123/pdf [Accessed 10 August 2016]

LAUDER, G. V., 2008. *Schooling behaviour in fishes.* Cambridge: Lauder Laboratory, Harvard University. Available from: http://www.people.fas.harvard.edu/~glauder/SchoolingbehaviorInFishes.htm [Accessed 10 August 2016]

REYNOLDS, R., 1987. Flocks, Herds, and Schools: A Distributed Behavioural Model. Computer Graphics, 21(4). *ACM SIGGRAPH Conference Proceedings*, 27-31 July 1987. Anaheim: ACM. Available from: http://www.cs.toronto.edu/~dt/siggraph97-course/cwr87/ [Accessed 10 August 2016].

SCHWAB, B., 2009. *AI Game Engine Programming*. 2nd ed. California, USA: Delmar.

SNAPE, J. AND MANOCHA, D., 2010. Navigating multiple simple-airplanes in 3D workspace. *International Conference on Robotics and Automation*, 3 – 8 May 2010. Anchorage. Available from: http://gamma.cs.unc.edu/S-AIRPLANE/S-AIRPLANE.pdf [Accessed 10 August 2016].

SNAPE, J., GUY, S. AND VAN DEN BERG, J., 2010. Independent Navigation of Multiple Robots and Virtual Agents. *9th International Conference on Autonomous Agents and Multiagent Systems*, 10-14 May 2010. Toronto. Available from: http://gamma.cs.unc.edu/INDNAV/INDNAV.pdf [Accessed10 August 2016].

SNAPE, J., GUY, S.J., LIN., M.C., MANOCHA, D. AND VAN DEN BERG, J., 2012. Reciprocal collision avoidance and multi-agent navigation for video games. *AAAI Workshop - Technical Report.* 22–26 July 2012. Toronto: Sheraton Centre. Available from: https://www.aaai.org/ocs/index.php/WS/AAAIW12/paper/download/5247/5645 [Accessed 10 August 2016].

VAN DEN BERG, J., 2011. Reciprocal n-body collision avoidance. *International Symposium on Robotics Research,* 28 August - 3 September 2011. Flagstaff: Little America. Available from: http://gamma.cs.unc.edu/ORCA/publications/ORCA.pdf [Accessed 10 August 2016].

VAN DEN BERG, J., LIN, M., AND MANOCHA, D., 2008. Reciprocal velocity obstacles for real-time multi-agent navigation. *Proceedings - IEEE International Conference On Robotics And Automation*, 19-23 May 2008. Pacedena: Pasadena Conference Center. Available from: http://gamma.cs.unc.edu/RVO/icra2008.pdf [Accessed 10 August 2016].

VAN DEN BERG, J., PATIL, S., SEWALL, J., MANOCHA, D. AND LING, M., 2008. Interactive navigation of individual agents in crowded environments. *Proceedings of the 2008 symposium on Interactive 3D graphics and games*. 15-17 February 2008. Redwood City: Electronic Arts Campus. Available from: http://rll.berkeley.edu/~sachin/papers/Berg-I3D2008.pdf [Accessed 10 August 2016].

Appendix

Lua Scripts

Agent Flow

```
-- Agent Flow --
--Demonstrates how the constant flow of agents can be changed over time
maxAgent=interpreter:getNumAgents()

zero={x=0.0,
        y=0.0,
        z=0.0}

-- Retrieve current point of predator movement cycle
newAngle=interpreter:getPredatorAngle()

-- ensure angle doesn't exceed 360.
ifnewAngle>100thennewAngle=0
end

-- Interate on angle for staggered animation
interpreter:setPredatorAngle(newAngle+0.001)


fori=0,400,1
do
-- First group of agents
agent=interpreter:getAgent()

-- If agent count exceeds maximum number of active agents a segfalt occurs.
-- Reset to zero to prevent this.
if agent >maxAgent
then
agent=0
end

position=interpreter:getAgentPosition(agent)
positionAbsolute=interpreter:getAbsoluteValue(position)

firstGoal={x=-100,
                y=0,
                z=-100}

goalVector=interpreter:subVectors(firstGoal, position)

cohesion=interpreter:getCohesion(agent)

velocity=interpreter:addVectors(goalVector,cohesion)

amp=100.0

-- Determin the curve that the velocity is modified by
curveX=(amp*math.cos(newAngle*2))
curveY=(amp*math.sin(newAngle*2))

-- Cohesion provides steering effect
velocity.x = curveX + cohesion.x
velocity.y = curveY + cohesion.y

interpreter:setSingleAgentVelocity(agent, velocity)

-- Reset the positions of the agents
ifposition.z<firstGoal.z+10
then
startPoint={x=-100-math.random(),y=0,z=100}

interpreter:setAgentPosition(agent,startPoint)
end

interpreter:setAgent(agent +1)


end
```

Bait Ball

```
-- Bait ball simulation --

-- Brief: This file is intended for use with Luke Bazalgette Thesis 2016
-- Agents are repelled away from predators
-- Simulates the effects of real world bait balling in schools of fish.

-- Gobal variables

-- Total number of agents for loop condition
maxAgent=interpreter:getNumAgents()

-- Vector3 format
zero={x=0.0,
        y=0.0,
        z=0.0}

-- Retrieve current point of predator movement cycle
newAngle=interpreter:getPredatorAngle()

-- Interate on angle for staggered animation
interpreter:setPredatorAngle(newAngle+1.0)

-- ensure angle doesn't exceed 360.
ifnewAngle==360thennewAngle=0
end

-- convert angle into radians
predatorAngle=math.rad(newAngle)



-- bait ball returns velocity required to circle point in a sphere formation
functionbaitBall(angle, position,center,centerVector, agent)

-- The following section is from :-
-- @gamefromscratch (24 November 2012). GAMEDEV MATH RECIPES: ROTATING ONE POINT AROUND ANOTHER POINT
[online]. [Accessed 2016].
--  Available  from:  <http://www.gamefromscratch.com/post/2012/11/24/GameDev-math-recipes-Rotating-one-
point-around-another-point.aspx>.
newVelocity={x=(math.cos(angle)*(position["x"]-center["x"])-math.sin(angle)*(position["z"]-center["z"])-
center["x"])*math.random(),
               y=(math.sin(centerVector["y"])*(position["x"]-center["x"])-
math.sin(centerVector["y"])*(position["z"]-center["z"])-center["y"])*math.random(),
               z=(math.sin(angle)*(position["x"]-center["x"])-math.sin(angle)*(position["z"]-
center["z"])-center["z"])*math.random()}
-- end of Citation

cohesion=interpreter:getCohesion(agent)
alignment=interpreter:getAlignment(agent)

steer=interpreter:addVectors(cohesion,alignment)

newSpeed=interpreter:getAgentMaxSpeed(agent)-0.05
ifnewSpeed<0.4thennewSpeed=0.4end
interpreter:setAgentMaxSpeed(agent,newSpeed)

final=interpreter:addVectors(centerVector,interpreter:addVectors(newVelocity,steer))

return final
end


-- Change second value to increase Velocity sample rate
fori=0,70,1
do
-- Get the current agent
agent=interpreter:getAgent()

-- If agent count exceeds maximum number of active agents a segfalt occurs.
-- Reset to zero to prevent this.
if agent >maxAgent
```

```
then
agent=0
end

-- Get Vector data for current agent
velocity=interpreter:getAgentVelocity(agent)
position=interpreter:getAgentPosition(agent)

-- Get the middle of the agent swarm
center=interpreter:getCenterOfSwarm()

-- Get position of goal in simulation
goal=interpreter:getGoal()

-- return the value to Cpp for use in functions
interpreter:setCenter(center)

-- Get the distance from agent to center of swarm
centerVector=interpreter:subVectors(center, position)

-- Initial position of predator
predator={x=40,y=0,z=30}

-- Initial position of predator
predator={x=40,y=0,z=30}

-- Point predator orientates around
predatorOrigin={x=40,y=0,z=30}

-- Move in an irregular pattern
predator["x"]=(predatorOrigin["x"]+math.cos(predatorAngle*2)*60)
predator["z"]=(predatorOrigin["z"]+math.sin(predatorAngle/1.5)*60)
predator["y"]=(predatorOrigin["y"]+math.cos(predatorAngle*3)*60)


otherPredator={x=10,y=0,z=10}

predatorOrigin={x=20,y=20,z=50}

otherPredator["x"]=(predatorOrigin["x"]+math.cos(predatorAngle/2.5)*100)
otherPredator["z"]=(predatorOrigin["z"]+math.sin(predatorAngle/6)*40)
otherPredator["y"]=(predatorOrigin["y"]+math.cos(predatorAngle*2)*20)

finalPredator={x=10,y=0,z=10}

predatorOrigin={x=10,y=20,z=15}

finalPredator["x"]=(predatorOrigin["x"]+math.cos(predatorAngle*0.2)*80)
finalPredator["z"]=(predatorOrigin["z"]+math.sin(predatorAngle/3)*70)
finalPredator["y"]=(predatorOrigin["y"]+math.cos(predatorAngle*0.2)*95)

-- You can set any given Vector as a predator
predatorArray={otherPredator,finalPredator, predator}

-- Get absolute values of Vectors
positionAbsolute=interpreter:getAbsoluteValue(position)
predatorAbsolute=interpreter:getAbsoluteValue(predator)
centerAbsolute=interpreter:getAbsoluteValue(center)

-- Absolute of predatpor vector
predatorVectorAbs=interpreter:getAbsoluteValue(interpreter:subVectors(position, predator))

-- Angle of velocity adjustment agents undergo
angle=math.rad(math.pi*45)

-- Distance agents detect predators
distance=80

-- Previous predator to be detected
-- required in order for agents to respond to multiple predaotrs.
oldPredator={x=0,y=0,
    z=0}
```

```
-- Number of predators in the range of the bait ball
predatorsInRange=0

-- Set agent to bait ball before searching for predators.
-- This will maintain bait ball shape.
interpreter:setSingleAgentVelocity(agent,baitBall(angle, position,center,centerVector, agent))

-- Loop through list of predators and adjust velocity of each agents
fork,vinpairs(predatorArray)
do
-- Get current predator and absolute position
newPredator=predatorArray[k]
predatorAbsolute=interpreter:getAbsoluteValue(newPredator)

-- Check if predator is in range of the bait ball
ifnewPredator["x"]- distance < position["x"]and position["x"]<newPredator["x"]+ distance and
newPredator["z"]- distance < position["z"]and position["z"]<newPredator["z"]+ distance and
newPredator["y"]- distance < position["y"]and position["y"]<newPredator["y"]+ distance
then

-- Increase speed as predator approaches bait ball
newSpeed=interpreter:getAgentMaxSpeed(agent)
+(math.abs(predatorAbsolute-positionAbsolute)*0.015)


ifpredatorAbsolute-50<positionAbsoluteand
predatorAbsolute+50>positionAbsolute
then

-- Add to value if a predator is within range. Used to scale magnitude.
predatorsInRange=predatorsInRange+1

-- Cap speed at 2.0
ifnewSpeed>2.0thennewSpeed=2.0end
interpreter:setAgentMaxSpeed(agent,newSpeed)

-- Update center vector
centerVector=interpreter:subVectors(center, position)

-- Get velocity to achieve cohesion between Agents
cohesion=interpreter:getCohesion(agent)

-- Update the velocity of the agent
oldVelocity=interpreter:getAgentVelocity(agent)

-- Check that there is a recorded value in OldPredator
ifoldPredator.x~=0
then
oldVelocity=interpreter:subVectors(position,oldPredator)
end

-- Scale vector according to how many predators are in range
newPredator=interpreter:divideVectors(newPredator,predatorsInRange)

-- Agent moves away from predator
interpreter:setSingleAgentVelocity(agent,
interpreter:addVectors(oldVelocity,interpreter:subVectors(cohesion,newPredator)))

-- Split away from center if predaotr comes too close to the center
-- Split away from center if predaotr comes too close to the center
ifpredatorAbsolute-25<positionAbsoluteand
predatorAbsolute+25>positionAbsolute
then

interpreter:setSingleAgentVelocity(agent,
interpreter:addVectors(oldVelocity,interpreter:subVectors(position,newPredator)))
interpreter:setAgentMaxSpeed(agent,3.0)

end

-- Set position of current vector to be used in next looop if needed
oldPredator=interpreter:addVectors(oldPredator, v)
oldPredator=interpreter:divideVectors(interpreter:addVectors(oldPredator, v),predatorsInRange)
```

```
end
end
end

-- Iterate agent count and return to Cpp
interpreter:setAgent(agent +1)

end

numPredators=interpreter:getPredators()

-- Pass positions of predators back into program
ifnumPredators==0then
fork,vinpairs(predatorArray)
do
interpreter:setPredators(v)
end
end
```

Feeding effect

```
-- Feeding simulation --

-- Fish will swarm around predators until moving on to next point.

-- Gobal variables

-- Total number of agents for loop condition
maxAgent=interpreter:getNumAgents()

-- Vector3 format
zero={x=0.0,
        y=0.0,
        z=0.0}

-- Retrieve current point of predator movement cycle
newAngle=interpreter:getPredatorAngle()

-- Interate on angle for staggered animation
interpreter:setPredatorAngle(newAngle+1.0)

-- ensure angle doesn't exceed 360.
ifnewAngle==360thennewAngle=0
end

-- convert angle into radians
predatorAngle=math.rad(newAngle)


-- Change second value to increase Velocity sample rate
fori=0,200,1
do
-- Get the current agent
agent=interpreter:getAgent()

-- If agent count exceeds maximum number of active agents a segfalt occurs.
-- Reset to zero to prevent this.
if agent >maxAgent
then
agent=0
end

-- Get Vector data for current agent
velocity=interpreter:getAgentVelocity(agent)
position=interpreter:getAgentPosition(agent)

-- Get the middle of the agent swarm
center=interpreter:getCenterOfSwarm()
```

```
-- Get position of goal in simulation
goal=interpreter:getGoal()
goalVector=interpreter:subVectors(goal, position)

-- return the value to Cpp for use in functions
interpreter:setCenter(center)

-- Get the distance from agent to center of swarm
centerVector=interpreter:subVectors(center, position)

-- Initial position of predator
predator={x=100,y=0,z=0}

otherPredator={x=250,y=0,z=0}

finalPredator={x=500,y=0,z=0}


-- You can set any given Vector as a predator
predatorArray={ predator,otherPredator,finalPredator}

-- Get absolute values of Vectors
positionAbsolute=interpreter:getAbsoluteValue(position)
predatorAbsolute=interpreter:getAbsoluteValue(predator)
centerAbsolute=interpreter:getAbsoluteValue(center)


-- Angle of velocity adjustment agents undergo
angle=math.rad(math.pi*45)



-- Distance agents detect predators
distance=70

-- Previous predator to be detected
-- required in order for agents to respond to multiple predaotrs.
oldPredator={x=0,y=0,z=0}

-- Number of predators in the range of the bait ball
predatorsInRange=0

objective={x=600,y=0,z=0}

objectiveVector=interpreter:subVectors(objective,position)

-- Set agent to bait ball before searching for predators.
-- This will maintain bait ball shape.
interpreter:setSingleAgentVelocity(agent,objectiveVector)
-- Loop through list of predators and adjust velocity of each agents
fork,vinpairs(predatorArray)
do
-- Get current predator and absolute position
newPredator=predatorArray[k]
predatorAbsolute=interpreter:getAbsoluteValue(newPredator)

-- Check if predator is in range of the bait ball
ifnewPredator["x"]- distance < position["x"]and position["x"]<newPredator["x"]+ distance and
newPredator["z"]- distance < position["z"]and position["z"]<newPredator["z"]+ distance and
newPredator["y"]- distance < position["y"]and position["y"]<newPredator["y"]+ distance
then

ifpredatorAbsolute-20<positionAbsoluteand
predatorAbsolute+20>positionAbsolute
then

-- Add to value if a predator is within range. Used to scale magnitude.
predatorsInRange=predatorsInRange+1

-- Update center vector
centerVector=interpreter:addVectors(center, position)
```

```
-- Update the velocity of the agent
oldVelocity=interpreter:getAgentVelocity(agent)

-- Check that there is a recorded value in OldPredator
ifoldPredator.x~=0
then
oldVelocity=interpreter:subVectors(position,oldPredator)
end


-- Scale vector according to how many predators are in range
newPredator=interpreter:divideVectors(newPredator,predatorsInRange)

predatorVector=interpreter:subVectors(newPredator, position)

oldVelocity=interpreter:addVectors(oldVelocity,predatorVector)

oldVelocity=interpreter:divideVectors(oldVelocity,2)

-- Agent moves away from predator
interpreter:setSingleAgentVelocity(agent,oldVelocity)

-- Set position of current vector to be used in next looop if needed
oldPredator=interpreter:addVectors(oldPredator, v)
oldPredator=interpreter:divideVectors(interpreter:addVectors(oldPredator, v),predatorsInRange)

end
end
end

-- Iterate agent count and return to Cpp
interpreter:setAgent(agent +1)

end

numPredators=interpreter:getPredators()

-- Pass positions of predators back into program
ifnumPredators==0then
fork,vinpairs(predatorArray)
do
interpreter:setPredators(v)
end
end
```

Fish tornado

```
-- Fish Tornado

-- Simulates a shoaling style exhibited by fish schools.
-- Resembles a vortex
maxAgent=interpreter:getNumAgents()

zero={x=0.0,
        y=0.0,
        z=0.0}

-- Retrieve current point of predator movement cycle
newAngle=interpreter:getPredatorAngle()

-- ensure angle doesn't exceed 360.
ifnewAngle>360thennewAngle=0
end

interpreter:setPredatorAngle(newAngle+0.01)

-- convert angle into radians
predatorAngle=math.rad(newAngle)
```

```
firstGoal={x=-100,y=0,z=-100}

-- Point predator orientates around
flowOrigin={x=60,y=0,z=10}

speed=0.1
magintude=100
-- Move in an irregular pattern
firstGoal["x"]=(flowOrigin["x"]+math.cos(newAngle*speed)*100)
firstGoal["z"]=(flowOrigin["z"]+math.sin(newAngle*speed)*100)


interpreter:setGoal(firstGoal)

fori=0,20,1
do
-- First group of agents
agent=interpreter:getAgent()

-- If agent count exceeds maximum number of active agents a segfalt occurs.
-- Reset to zero to prevent this.
if agent >maxAgent
then
agent=0
end


position=interpreter:getAgentPosition(agent)

cohesion=interpreter:getCohesion(agent)

interpreter:setAgentMaxSpeed(agent,0.5)



goalVector=interpreter:subVectors(firstGoal, position)


velocity=interpreter:addVectors(goalVector,cohesion)

-- Fish occilate relative to their point of origin.
-- Pattern is perturbed further through neighbour awareness
curveY=(2.0*math.sin(newAngle*2*math.pi))

-- Cohesion needs to be readded to the velocity to produce formation
velocity.y=curveY+cohesion.y

interpreter:setSingleAgentVelocity(agent, velocity)


interpreter:setAgent(agent +1)


end
```

Goal Tracking

```
-- Goal Tracking --

maxAgent=interpreter:getNumAgents()

zero={x=0.0,y=0.0,z=0.0}

-- Agents circle around a given point
functionbaitBall(angle, position,center,centerVector, agent)
newVelocity={x=(math.cos(angle)*(position["x"]-center["x"])-math.sin(angle)*(position["z"]-center["z"])-
center["x"])*math.random(),
               y=(math.sin(centerVector["y"])*(position["x"]-center["x"])-
math.sin(centerVector["y"])*(position["z"]-center["z"])-center["y"])*math.random(),
               z=(math.sin(angle)*(position["x"]-center["x"])-math.sin(angle)*(position["z"]-
center["z"])-center["z"])*math.random()}

cohesion=interpreter:getCohesion(agent)
alignment=interpreter:getAlignment(agent)

steer=interpreter:addVectors(cohesion,alignment)

final=interpreter:addVectors(centerVector,interpreter:addVectors(newVelocity,steer))

return final
end

fori=0,200,1
do
agent=interpreter:getAgent()

if agent >maxAgent
then
agent=0
end

velocity=interpreter:getAgentVelocity(agent)
position=interpreter:getAgentPosition(agent)

center=interpreter:getCenterOfSwarm()
interpreter:setCenter(center)
centerVector=interpreter:subVectors(center, position)

goal=interpreter:getGoal()
goalVector=interpreter:subVectors(goal, position)


ifinterpreter:getAgentNeighborCount(agent)>0
thenneighbor=interpreter:getAgentNeighborNum(agent,0)end

positionAbsolute=interpreter:getAbsoluteValue(position)
goalAbsolute=interpreter:getAbsoluteValue(goal)

cohesion=interpreter:getCohesion(agent)

alignment=interpreter:getAlignment(agent)

angle=math.rad(math.pi*10)

bait=baitBall(angle, position, goal,goalVector, agent)

finalVelocity=interpreter:addVectors(zero,goalVector)

-- Add modifiers to velocity
finalVelocity=interpreter:addVectors(finalVelocity, cohesion)

finalVelocity=interpreter:addVectors(finalVelocity, alignment)

finalVelocity=interpreter:addVectors(finalVelocity, bait)

interpreter:setSingleAgentVelocity(agent,finalVelocity)
```

```
distance=50.0

ifgoalAbsolute-10.0<positionAbsoluteandpositionAbsolute<goalAbsolute+10.0and
goal["x"]- distance < position["x"]and position["x"]< goal["x"]+ distance and
goal["z"]- distance < position["z"]and position["z"]< goal["z"]+ distance and
goal["y"]- distance < position["y"]and position["y"]< goal["y"]+ distance
then
-- Generate random number in codition to improve performance
randomX=math.random(0,150)-math.random(0,150)
randomY=math.random(0,150)-math.random(0,150)
randomZ=math.random(0,150)-math.random(0,150)

goal.x=randomX
goal.y=randomY
goal.z=randomZ

interpreter:setGoal(goal)
end

interpreter:setAgent(agent +1)


end
```

Opposite Goals

```
-- Opposite goals

-- Script is designed to show how two groups of agents interact with each other when travelling to
opposite goals.
-- Faster agents are more aggressive to therefore slower agents will be forced to adjust their velocity.

-- Also serves to demonstrate the effects of cohesion on an agent's path.

maxAgent=interpreter:getNumAgents()

zero={x=0.0,
        y=0.0,
        z=0.0}


firstGoal={x=-100,
             y=0,
             z=-100}


fori=0,400,1
do
-- First group of agents
agent=interpreter:getAgent()

-- If agent count exceeds maximum number of active agents a segfalt occurs.
-- Reset to zero to prevent this.
if agent >maxAgent
then
agent=0
end

interpreter:setAgentMaxSpeed(agent,2.0)

position=interpreter:getAgentPosition(agent)

goalVector=interpreter:subVectors(firstGoal, position)

cohesion=interpreter:getCohesion(agent)

velocity=interpreter:addVectors(goalVector,cohesion)

interpreter:setSingleAgentVelocity(agent, velocity)

-- Reset position of agents past a certain point
ifposition.x<firstGoal.x
```

```
then
startPoint={x=100+math.random(),y=0,z=100}

interpreter:setAgentPosition(agent,startPoint)
end


--Second group
agent= agent+1

if agent >maxAgent
then
agent=0
end

interpreter:setAgentMaxSpeed(agent,0.5)

position=interpreter:getAgentPosition(agent)

secondGoal={x=100,
                 y=0,
                 z=-100}

goalVector=interpreter:subVectors(secondGoal, position)

cohesion=interpreter:getCohesion(agent)

velocity=interpreter:addVectors(goalVector,cohesion)

interpreter:setSingleAgentVelocity(agent, velocity)

ifposition.x>secondGoal.x
then
startPoint={x=-100-math.random(),y=0,z=100}

interpreter:setAgentPosition(agent,startPoint)
end



interpreter:setAgent(agent +1)


end
```

Script Documentation

```
-- Luke Bazalgette Thesis Lua Syntax Guide --

-- WARNING
-- This code is not intended to be run within the simulation
-- It is meerely to document a list of valid syntax

-- The functions listed in this docuyment should be safe to cut and paste into a valid Lua script.

--1. Interpreter variables

-- luawrapper already provides functionality for multiple variable types
-- Integers, floats and strings are straightforward to return

agentNum=0
float=0.0
stringSample="Hello word!"

-- RVO::Vector3
-- These are written as tables within Lua.
-- Please follow the format below
```

```
vector={x=0,y=0,z=0}

-- Contents of X,Y and Z are float values
-- Returned as RVO::Vector3
-- please remember to include X,Y and Z, otherwise it will be treated as a regular table.

-- Values of the Vector can be accessed in Lua as such

vector.x=vector.y

--2. Interpreter functions

--Full list of valid interpreter functions and arguments
-- The following functions are Getters and Setters for variables sotred within Cpp.
-- This is neccessary as Lua flushes all variable values on completing script execution.

agentNum=interpreter:getAgent()--Get agent value stored in LuaInterpreterCpp class
interpreter:setAgent(interger)--Set the agent value to an integer from Lua

goal=interpreter:getGoal()--Get position of goal stored in LuaInterpreterCpp class
interpreter:setGoal(vector)--Set position of goal stored in LuaInterpreterCpp class

center=interpreter:getCenter()--Get vector value of center stored in LuaInterpreterCpp class
interpreter:setCenter(vector)--Get vector value of center stored in LuaInterpreterCpp class

predatorCount=interpreter:getPredators()--Get size of predator array. Returned as Integer.
interpreter:setPredators(vector)--Push a Vector value into m_predators in Cpp. Used to draw predators in
NGLScene.
--Put in a loop to push multiple values in a table.

predatorAngle=interpreter:getPredatorAngle()--Get  the  angle  of  predator  movment  from  LuaInterpreter
class.
--Since it is intended to iterate with each run of the script it is important to return the value.
interpreter:setPredatorAngle(float)--Set value in Cpp. It is advised to be used as an iterator.

-- The following functions retrieve values to be used as constraints within the script.
agentCount=interpreter:getNumAgents()-- Get number of active agents in the simulation
interpreter:getAgentNeighborCount(agentNum)--retrieve total number of neighbors for a given agent.

--Without these, certain loops may cause a segmentation fault


-- Reynolds behaviours
-- These are global behaviours calculated in Cpp
-- They are common to most flocking algorithms as such simplify the code required.
cohesion=interpreter:cohesion(agentNum)--returns Vector velocity that achieves cohesion
alignment=interpreter:alignment(agentNum)-- returns Vector velocity that achieves alignment

-- Agent operations
-- set agent attributes using these functions.
velocity=interpreter:getAgentVelocity(agentNum)-- get the current velocity of a given agent
position=interpreter:getAgentPosition(agentNum)-- get the current position of a given agent

interpreter:setAgentPosition(agentNum, vector)-- set the position of an agent to a specified vector

interpreter:setSingleAgentVelocity(agentNum, vector)-- set the preferred velocity of a given agent
intepreter:setAllAgentVelocities(vector)-- Set a new preferred velocity for the entire swarm

maxSpeed=interpreter:getAgentMaxSpeed(agentNum)-- Get the agent's current max speed.
interpreter:setAgentMaxSpeed(agentNum, float)--Set agent max speed to a given float.

flockCenter=interpreter:getCenterOfSwarm()--Get the average position of all agents in the simulation.

-- Vector operations
vector=interpreter:addvectors(vector,vector)--Adds the corresponding axis of each Vector to one another
vector=interpreter:subVectors(vector,vector)--Subtract  the  right  hand  vector  from  the  left  and  return
the value

vector=interpreter:multiplyVector(vector,1)--Scale the vector values by the argument on the right.
vector=interpreter:divideVector(vector,1)-- divide the values in the given vector by the argument on the
right.
```

```
bool=compareVectors(vector,vector)--Find out if two vectors have the same values. returns 1 if true,
otherwise 0.
--Useful as an If condition

absolute=interpreter:getAbsoluteValue(position)-- Returns absolute value (length) of a given vector.
Returned as float.
power=interpreter:getPowerOf(float,2)--Returns the power of a given float value to the right hand
argument. Lua doesn't have this functionality.


-- 3. Useful code

-- This section is dedicated to pieces of useful code that is applicable to most scripts

-- If you have an angle iterator you won't want it to exceed 360.
-- This function resets the value to 0, place at the begginning of your code and before interacting the
angle
ifnewAngle==360
thennewAngle=0
end

-- Printing Vectors:
-- To print a vector it must be placed within a loop like this.
fork,vinpairs(vector)
do
printk,v
end

-- Vector to a point.
-- If you want to make an agent travel to a given point...
-- you will need to find the distance between it's current position and the goal.

-- Get the distance from agent to center of swarm
centerVector=interpreter:subVectors(center, position)
interpreter:setSingleAgentVelocity(agentNum,centerVector)

-- This applies to any given point

-- This is a typical loop for applying new Velocities to all active agents

fori=0,50,1-- Set second value to decide how many agents to sample during a given loop
do
-- Get the current agent from LuaInterpreter
agent=interpreter:getAgent()

-- If agent count exceeds maximum number of active agents a segfalt occurs.
-- Reset to zero to prevent this.
if agent >maxAgent
then
agent=0
end

-- Iterate agent number for next loop
-- Also ensures that each agent is assessed in order
interpreter:setAgent(agent +1)

end
```

Starting Template

```
-- Starting template --
-- Use this script as a strarting point for new behaviours
-- Contains various useful values that are common to most scripts

maxAgent=interpreter:getNumAgents()

zero={x=0.0,
        y=0.0,
        z=0.0}


fori=0,200,1
do
-- First group of agents
agent=interpreter:getAgent()

-- If agent count exceeds maximum number of active agents a segfalt occurs.
-- Reset to zero to prevent this.
if agent >maxAgent
then
agent=0
end

position=interpreter:getAgentPosition(agent)

goal=interpreter:getGoal()
goalVector=interpreter:subVectors(goal, position)

center=interpreter:getCenter()
centerVector=interpreter:subVectors(center, position)

cohesion=interpreter:getCohesion(agent)
alignment=interpreter:getAlignment(agent)

velocity=interpreter:addVectors(goalVector,cohesion)

-- Retrieve current point of predator movement cycle
newAngle=interpreter:getPredatorAngle()

-- ensure angle doesn't exceed 360.
ifnewAngle>360thennewAngle=0
end

interpreter:setSingleAgentVelocity(agent, velocity)


interpreter:setAgent(agent +1)


end
```