School of Computing

FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Final Report

3D Simulation of fish schools using the Boids algorithm

Christian Constantin Pustianu

Submitted in accordance with the requirements for the degree of
MEng, BSc Computer Science with High-Performance Graphics and Games Engineering

2022/23

COMP3921 Individual Project

The candidate confirms that the following have been submitted*:*

Items	**Format**	**Recipient(s) and Date**
Final Report	*PDF file*	*Uploaded to Minerva (01/05/23)*
Link to online code repository	*URL*	*Sent to supervisor and assessor (01/05/23)*
User manuals	*Readme.md*	*In online code repository*
Demo video	*URL*	*Sent to supervisor and assessor (01/05/23)*

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student) Christian Constantin Pustianu

Summary

This paper describes in detail the process of creating a realistic 3D fish simulation using Craig Reynolds' Boids behavioural model (1987) and other related work. It outlines the steps involved in constructing the required models, implementing the algorithm, incorporating environmental factors such as collision detection and avoidance, and designing a GUI that provides the user with an easy way to tune parameters. It also provides a comprehensive evaluation and analysis of the results, demonstrating the effectiveness of the approach in generating natural schooling behaviour.

Acknowledgements

I would like to acknowledge my supervisor, Professor Hamish Carr, for his support and advice throughout this project, and my assessor, Professor Marc de Kamps, for valuable mid-term feedback.

Table of Contents

Chapter 1
Introduction and Background Research

1.1 Introduction

Flocking is an emerging collective behaviour that appears in nature for many species of animals. Examples of this include swarms of insects, flocks of birds, mammal herds, and many others. Fish schools in particular, are one fascinating example of this collective behaviour, and by understanding the factors that influence this behaviour, we can replicate it in simulations.

Achieving natural behaviour in animals is a process often performed manually in animation, but it quickly becomes unreasonable when it involves a large group of animals. Therefore, the need for automation appears, which can be fulfilled by applying principles of AI agents. The animals can be modelled as intelligent agents with rules to follow, that guide their actions based on environmental factors (van der Vaart and Verbrugge, 2008, Macal and North, 2005). One way to model this behaviour is through the algorithm developed by Reynolds (1987), which by using three simple rules, results in complex and seemingly natural movements for the group as a whole. In the same way a fish only sees its immediate surroundings, the agents move based on the state and actions of close-by neighbours.

1.2 Literature Review

1.2.1 Craig Reynolds' original paper

Flocks, Herds, and Schools: A Distributed Behavioral Model published by Craig Reynolds' in 1987 was a breakthrough for all graphics-related industries as it not only automated the process of animating large groups, but also provided a solid foundation for modelling behaviour. At its core the paper discusses how controlling individual behaviour can lead to emerging collective behaviour. Given a simple set of rules that simulates their perception, the agents, or *boids* (derived from *birdoid*), dynamically form and dissipate groups based on their environment. These rules are defined as *cohesion* (steer towards the centre of mass of neighbours), *alignment* (align direction with that of neighbours) and *separation* (avoid

Figure 1.1 Example of boids simulation from Reynolds' paper (1987, p.33)

collisions with neighbours). The author also tackles many issues that can appear in implementing this model, such as obstacle avoidance, maintaining correct orientation and algorithmic considerations.

The model has seen many applications, especially throughout the video game and film industries, but also in other unexpected areas. It has been used to control bird-like creatures (unsurprisingly named *Boids*) in the 1998 video game *Half-Life*, flocks of fish in the 1987 animation *Stanley and Stella in: Breaking the Ice* and even for swarms of bats and armies of penguins in Tim Burton's 1992 feature film *Batman Returns*. A more recent application appears in populating the virtual world with fish and crowds in *Disney's Raya and the Last Dragon* (2021) as presented by Ros et al. (2021). Moreover, it has seen applications in swarm robotics, for controlling both ground and aerial unmanned vehicles (Saska et al., 2012).

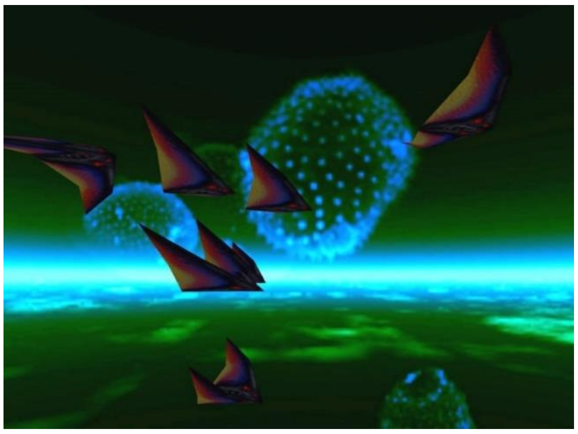


Figure 1.2 Standard Boids on Xen in Half-Life (1998).

Another potential benefit of boids simulations besides automating animations could be in scientific research. As outlined in Reynolds' paper, it could be a lot easier for biologists to use a simulation of flocks, herds, or schools, instead of having to observe the behaviour of real animals. Doing it in person might also disturb the natural behaviour that is being studied and thus decrease the reliability of the results. With a good implementation that takes into consideration all factors of the animals' perception, and more closely resembles the real world, it could prove invaluable to researchers as they could not only study realistic behaviour, but also replicate it any number of times.

1.2.2 Limitations of the model

On the other hand, since its release, the model has seen some amount of criticism due to its limitations. One of the main issues is that of oversimplification of animal behaviour. While the three rules used in the model result in believable emergent behaviour, in a real scenario there would be a multitude of factors and variables that could dynamically change the outcome. The implementation also does not take into account any predator-prey interactions which could alter the movement patterns of the flock, or even more so, individual differences since real animals' fitness and characteristics can differ. Even aspects such as social dynamics can interfere in a real flock, where the animals could form hierarchies or exhibit social learning behaviours.

Another issue is the lack of realism when it comes to the boids' perception of the flock. The model assumes every boid has perfect knowledge of the position and velocity of every other member of the flock, when searching for neighbours. This not only contradicts with the perception of real-life flocking animals, but also creates issues for the applicability of the model. Since every individual must check with every other individual to decide if they are neighbours or not, this makes the algorithm unsuitable for very large crowds where the computational resources would be significant. Reynolds (1987) also argues that instead of using centralized processing, a distributed processing approach would be desirable since the real animals use the same concept, but it unfortunately would also be quite impractical as the problem size correlates to the number of processing units required.

1.2.3 Recent work on Boids

Some of the recent papers related to Boids have investigated different improvements and adaptations of the original model. These include coherent group pathfinding, using genetic algorithms and particle swarm optimisations, space partitioning and parallelization.

1.2.3.1 Coherent Group pathfinding

As presented by Kamphuis and Overmars (2004), coherent group pathfinding involves simulating large crowds and how they navigate around obstacles without separating, in order to get to a location. Instead of guiding the group towards a point in space while relying on collision avoidance to help create the path around obstacles, this approach uses an algorithm for creating a *backbone path* for a single unit and guiding the others based on this path. It involves choosing a path that is wide enough to create a corridor allowing for multiple units to pass, while also assuring the lateral and longitudinal dispersion is within some boundaries. In other words, the units don't move too far from the group and wait for each other, thus maintaining the group's integrity. Applications of these include not only games and virtual environments (when simulating large armies), but also crowd control when analysing safety measures for buildings and public places.

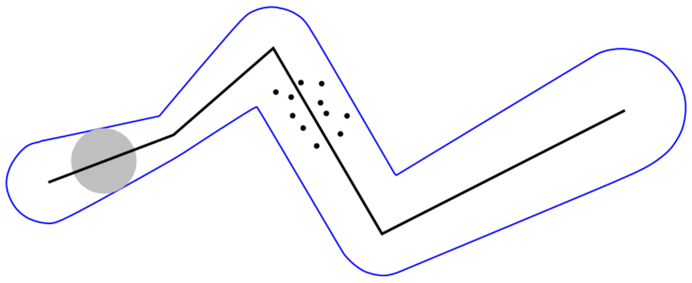


Figure 1.3 Formation of corridor from circles around backbone path (Kamphuis and Overmars, 2004, p.23)

1.2.3.2 Optimization Algorithms

A comparison of two different search algorithms, presented by Alaliyat et al. (2014), shows how the precision of the boids model can be improved in various ways. Both optimizations rely on adjusting the weights of the 3 rules (or 4 if the model includes an additional rule to “follow a leader”) so that their movement is as natural as possible. A cost function composed from the weights is used as a measure for evaluation, since minimizing it results in maximizing the fitness of the boids.

The first method presented is using a *Genetic Algorithm (GA)* to evolve and adapt the flock over multiple iterations based on the strongest individuals. The population is initialized with random genes (weights for the movement vectors) and for each iteration, some individuals get selected based on their fitness. They are then mutated amongst themselves to form a new population that is used to create the next generation.

The second method consists of using *Particle Swarm Optimisation (PSO)*, where each particle (or boid in our case) adjusts its velocity and flight pattern according to the positions in the search space with the best fitness values. Each particle keeps a record of its personal best position (*pbest*) based on fitness and has knowledge of the best position in its neighbourhood (*gbest*). With every iteration, each individual tries to guide its path using the two best position values, based on previous iterations.

Both algorithms conclude either by reaching a certain threshold for the cost function or by reaching a maximum number of iterations.

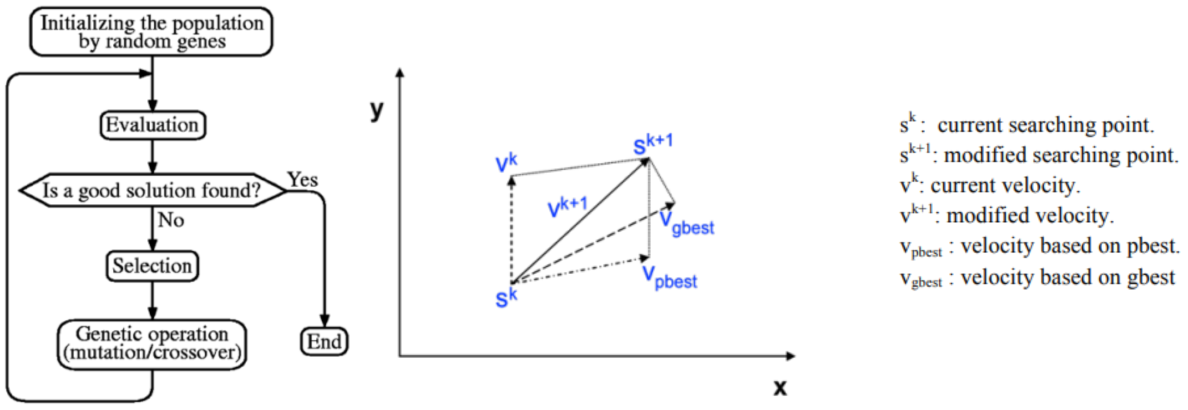

Initializing the population by random genes
Evaluation
Is a good solution found?
Yes
No
Selection
Genetic operation (mutation/crossover)
End
y
x
v^k
s^{k+1}
v^{k+1}
v_{gbest}
v_{pbest}
s^k
s^k: current searching point.
s^{k+1}: modified searching point.
v^k: current velocity.
v^{k+1}: modified velocity.
v_{pbest} : velocity based on pbest.
v_{gbest} : velocity based on gbest

Figure 1.4 (left) Flowchart of GA. (right) Concept of particle position modification by PSO (Alaliyat et al., 2014, p.647)

1.2.3.3 Space partitioning and Parallelization

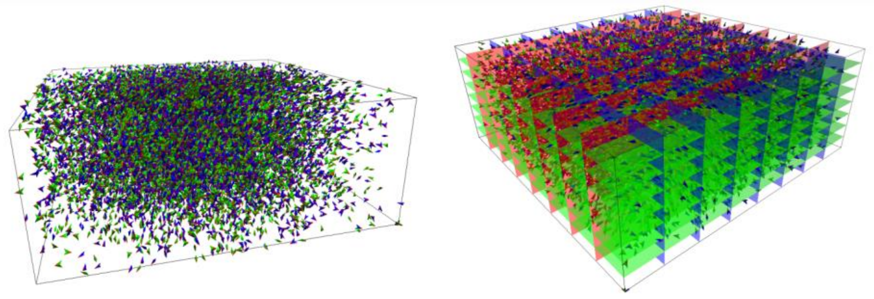
There are many algorithms for partitioning a simulation space in particle simulations and interaction-based models, such as Barnes-Hut treecodes (Barnes and Hut, 1986), Fast Multipole methods (Darve et al., 2011), and tree organisational methods: Octrees (Jackins and Tanimoto, 1980), K-D trees (Yang et al., 2011), and Adaptive Refinement Trees

(Kravtsov et al., 1997). Most of these are commonly used in computational physics and astrophysics simulations to efficiently calculate interactions between large numbers of particles or objects. More importantly for this project, they can also be applied to particle systems in computer graphics.

Space partitioning can provide a substantial improvement to the boids model by reducing computation if the problem is treated as a particle simulation. In the original implementation, each boid must loop through all other boids in the simulation space, which results in quadratic complexity $O(N^2)$ that becomes too computationally expensive for larger size problems. Dividing the simulation space means that each boid only checks its local and neighbouring space partitions, thus localizing the problem and greatly reducing computation.

Since the boids model performs the same computations for different data (for each boid), parallelization would be another optimization method. Moreover, due to its lack of data dependencies, it has even been categorized as an *embarrassingly parallel problem*.

Parallelization can also be combined with space partitioning, when using an algorithm that responds well to parallelization, to further accelerate the simulation. As explored by Hussleman and Hawick (2012), Grid-boxing in combination with some variations and improvements to GPU parallelization can lead to an overall complexity of the boids model of $O(N \log N)$, which is a great improvement over the original $O(N^2)$.



(a) Screenshot of our Boids simulation.

(b) Screenshot of the same simulation state with visualisation of the grid-box algorithm superposed.

Figure 1.5 Visualisation of the same set of agents with and without a uniform grid. (Hussleman and Hawick, 2012)

1.2.4 Related work

There are other concepts that can be of use when building a simulation of interactive objects. Some of the techniques used in this project that do not strictly relate to the boids model include obstacle collision detection and terrain generation.

1.2.4.1 Collision detection

The matter of collision detection can be tackled with a range of approaches with different advantages and disadvantages. There are simpler methods that use Bounding-Volumes (Sulaiman et al., 2015) which require significantly reduced computation, but are less accurate, or complex methods that check primitive-primitive intersections such as ray casting (Vlasov et al., 2012, Hermann et al., 2008) that is more computationally expensive, but produces better results.

The bounding-volumes generally used include Axis-Aligned Bounding Boxes (AABB), Oriented Bounding Boxes (OBB) and Spheres. AABB refers to rectangular boxes aligned with the xyz axes of the world’s coordinate system that surround the objects in the simulation and can easily be checked for intersections. OBB is a variation of AABB that allows for rotated boxes so that they better fit the objects. Meanwhile, the Sphere volume method involves surrounding the objects by spheres and using the radius for intersection checking.

While mostly applied in lighting models such as ray tracing, ray casting can also be applied to collision detection. Determining ray casting intersections involves casting a line from a point towards a direction and measuring the distance at which it intersects other objects. When used on moving objects, it can be cast in the direction of the movement to detect collisions with close-by objects before they occur. In the case of the boids model, each boid can shoot rays at incremental angles from its current direction until it finds a clear path that does not intersect with any obstacle.

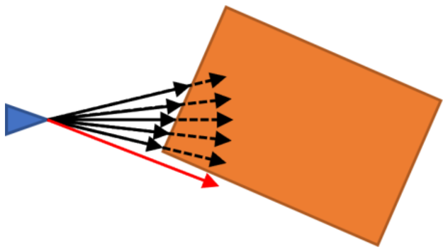
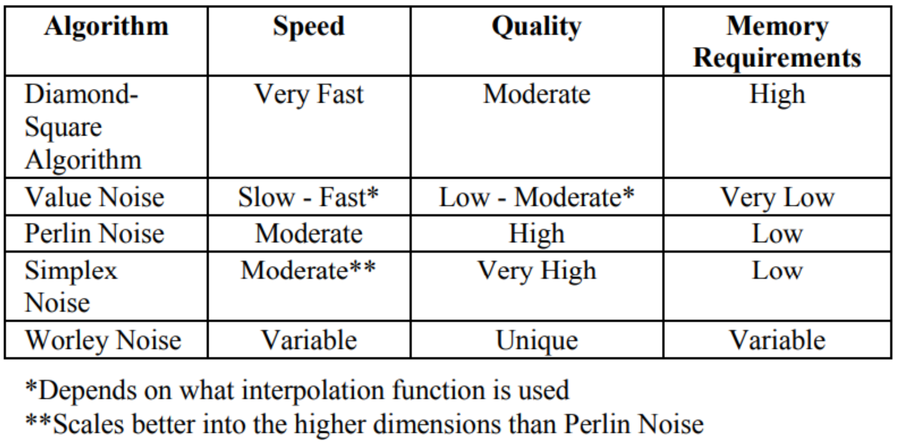


Figure 1.6 Obstacle avoidance using ray casting for collision detection.

1.2.4.2 Terrain generation

An important aspect of 3D simulations is that of terrain generation, where a realistic landscape is created to replicate a natural environment. Terrain is generally represented as a mesh that is procedurally generated using different algorithms, such as fractals and noise functions (Rose and Bakaoukas, 2016), or genetic algorithms (Ong et al., 2005). The usage of each of these depends highly on the desired speed, quality, and memory requirements, as they vary in their strengths and weaknesses.

The Diamond-Square Algorithm (Miller, 1986), a fractal-based terrain generation method, iteratively subdivides a square into four smaller squares, calculating the height of each new vertex based on the neighbouring vertices plus a random value. Multiple iterations of this subdivision create a fractal pattern of increasingly smaller squares, thus generating a complex terrain. The random values add some chaos into the mix, resulting in seemingly natural landscapes. Noise functions, on the other hand, generate random or pseudo-random values that are evenly distributed within a 2D space. Examples of these include Value Noise, Perlin Noise, Simplex Noise and Worley Noise, each suitable for different applications depending on the desired outcome.



Algorithm	Speed	Quality	Memory Requirements
Diamond-Square Algorithm	Very Fast	Moderate	High
Value Noise	Slow - Fast*	Low - Moderate*	Very Low
Perlin Noise	Moderate	High	Low
Simplex Noise	Moderate**	Very High	Low
Worley Noise	Variable	Unique	Variable

*Depends on what interpolation function is used
**Scales better into the higher dimensions than Perlin Noise

Table 1.1 Comparison of fractal-based algorithm and noise functions (Rose and Bakaoukas, 2016)

Next, a Genetic Algorithm can be used to iterate on any features of an initial terrain map in order to achieve the intended result. The features can be enhanced according to genes formed from arbitrary parameters, such as smoothness, matrix transformations or even some pre-defined complexity parameter. By setting up fitness evaluation metrics, these genes can then be mutated to produce the best results.

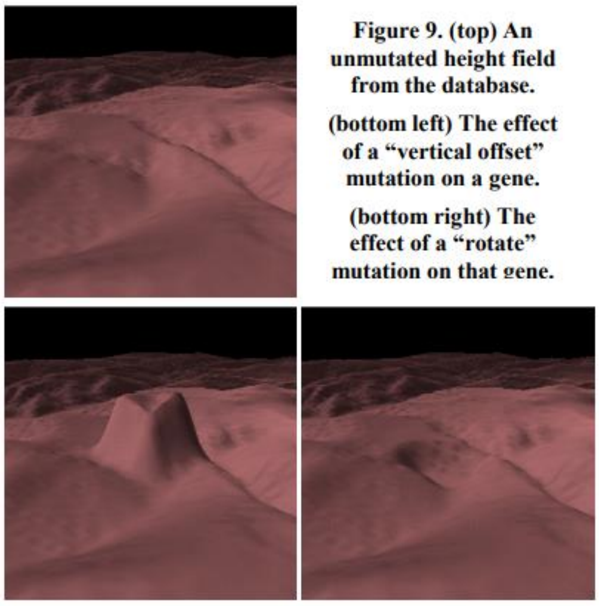


Figure 9. (top) An unmutated height field from the database.
(bottom left) The effect of a "vertical offset" mutation on a gene.
(bottom right) The effect of a "rotate" mutation on that gene.

Figure 1.7 Effects of mutations in a GA (Ong et al., 2005)

All these methods can be used to create 2D terrain maps suitable for generating terrain data. This data can then be represented as *height fields*, i.e., a scalar function $h = f(x, y)$ of a pair of coordinates corresponding to an elevation value (Ong et al., 2005), which usually represents the pixel value at those coordinates. Heigh fields are then used to generate the terrain mesh, that can later be modified by developers to add detail and textures.

Chapter 2
Methods

2.1 Solution Design

2.1.1 Geometric Flight

The basis of the implementation relies in having agents that continuously move towards a direction and changing that direction based on the algorithm. Therefore, to achieve realistic movement for our agents, we need to use what Reynolds (1987) refers to as *Geometric Flight*, which describes the movement of an object along a path, tangent to a 3D curve.

In practice, in the graphics rendering pipeline outlined by Akenine-Möller et al. (2008, pp.12-14) in their book, there is an Application Stage where the developer sets up the logic for the movement, followed by a Geometry Stage where the vertices in the object get transformed as desired, and a Rasterization Stage where the pixels are drawn to the framebuffer before outputting them to the screen. The Geometric Flight setup is implemented into the Application and Geometry Stages by defining the transformation matrix that is applied to the object when rendered.

This can be done by composing the *model to world matrix* (that is applied to vertices before the view and projection matrices) from a translation and a rotation matrix.

2.1.1.1 Boid Rotation

The rotation matrix can be implemented with a form of Rodrigues' rotation formula (or the equivalent Euler-Rodrigues formula) (Dai, 2015), to create a rotation matrix on a custom axis. Therefore, the final formula used in the implementation for the rotation matrix is:

$$R = \begin{bmatrix} \cos\theta + u_x^2(1-\cos\theta) & u_x u_y(1-\cos\theta) + u_z\sin\theta & u_x u_z(1-\cos\theta) + u_y\sin\theta \\ u_y u_x(1-\cos\theta) + u_z\sin\theta & \cos\theta + u_y^2(1-\cos\theta) & u_y u_z(1-\cos\theta) + u_x\sin\theta \\ u_z u_x(1-\cos\theta) + u_y\sin\theta & u_z u_y(1-\cos\theta) + u_x\sin\theta & \cos\theta + u_z^2(1-\cos\theta) \end{bmatrix}$$

where θ is the angle and $\boldsymbol{u} = (u_x, u_y, u_z)$ is the normalized axis vector.

In order to apply it, the angle and axis have to be computed in the following way: First, get the initial direction vector $\boldsymbol{d}_i$ of the boid, which is the direction the model is facing when imported into the scene, and the vector for the current direction $\boldsymbol{d}_c$, that the model should be facing after the rotation. The axis for the rotation can be computed by taking the normalized cross product of the two vectors to find a vector perpendicular on both. The angle θ between the initial and current vectors can then be computed by extracting the arccosine of the dot product of the vectors: $\theta = \arccos\frac{\boldsymbol{d}_i \cdot \boldsymbol{d}_c}{||\boldsymbol{d}_i||\,||\boldsymbol{d}_c||}$.

A visual problem that can appear from this type of rotation is when the angle θ is greater than $90°$, in which case the model will appear flipped on its back, as in **Figure 2.1** (left). A solution is to mirror the rotation if $\theta > 90°$, i.e., to start off from the model rotated towards the opposite vector of the initial direction, and to use the negative complementary angle to θ as the rotation angle.

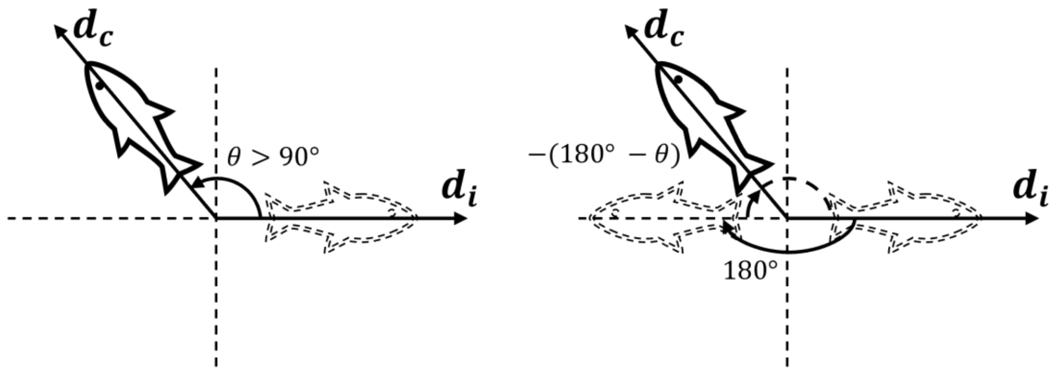


Figure 2.1 (left) Boid is flipped when rotated at an angle > 90°. (right) Corrected rotation by mirroring initial orientation and rotating in the opposite direction with $\mathbf{180°} - \boldsymbol{\theta}$.

2.1.1.2 Boid Translation

The current position of a boid is represented by a 3D vector containing its xyz coordinates. In order to translate the boid to this position, a translation matrix is used, with the position values. The current position vector is the one that gets incremented for every frame using the current direction $\boldsymbol{d}_c$ multiplied by a scalar speed modifier.

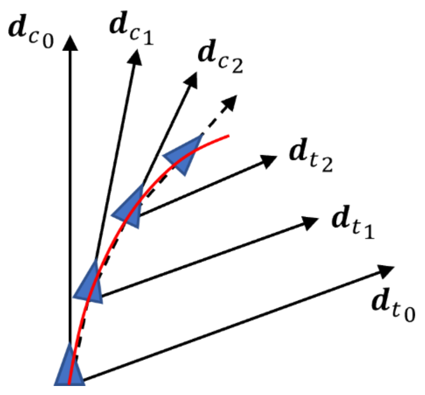


Figure 2.2 Successive interpolations of direction vector create movement tangent to 3D curve.

When first created, Boids start off with a random current position and direction in the simulation space, and with the target direction initialized as the current direction. For every frame, the new current direction $\boldsymbol{d}_{c_i}$ is computed from the previous current direction $\boldsymbol{d}_{c_{i-1}}$, and previous target direction $\boldsymbol{d}_{t_{i-1}}$. By interpolating the current direction vector with different methods and incrementing the position with it, this creates a continuous movement along a 3D curve, as depicted in **Figure 2.2**.

Interpolation is done by assigning complementary weights to the two vectors, so that the result is more influenced by one or the other and summing them up. In our case, that weight represents how sharp the boid will turn towards a direction.

The main type of interpolation used is normalized linear interpolation (nlerp) when the angle between $\boldsymbol{d}_c$ and $\boldsymbol{d}_t$ is less than $90°$. This way, the output vector will have the same unit length as the current one, resulting in a smooth constant movement. When the object needs to turn more than $90°$, some cases appear where nlerp no longer works properly, and spherical linear interpolation (slerp) is the better choice, as it creates a more natural turn while always maintaining constant velocity. Lastly, one edge case that needs to be handled is when the current and target vectors are completely opposite i.e., the angle is exactly $180°$, in which neither interpolation works since it needs a second axis for a plane on which to start the process. A solution to this problem is to increment the current vector very slightly on an arbitrary axis.

2.1.2 Algorithm Implementation

The implementation of the algorithm relies in changing the target direction vector depending on the outside factors. These include not only the three rules described in Reynolds' paper (1987), *cohesion*, *alignment,* and *separation*, but also other rules for avoiding the obstacles and simulation boundaries, as well as user input that influences the boids' behaviour. In practice, all these rules are implemented as functions that return vectors, which are all added to the current vector and normalized. This way, the output is a direction vector with influences from each rule. These rules can even have different influence strengths, which can be implemented as scalars that change the vectors' magnitudes.

Cohesion, Alignment and Separation act based on neighbouring boids, which are defined as boids in a certain radius of vision from the current boid, as depicted in **Figure 3**. The same way a bird or fish cannot see behind them, the radius must have a maximum angle α. Moreover, this neighbourhood of boids must be constantly updated, and the most straightforward way to do this is to loop through all other boids and measure the distance between their positions to decide if they are in the current boid's vision range. Unfortunately, without optimisations this is not as scalable, since it becomes too computationally expensive for a larger number of boids, as will be outlined in ***Chapter 3: Results***.

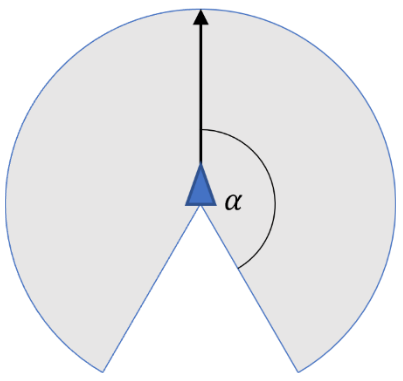


Figure 2.3 Vision radius at angle alpha.

The rules in the implementation are described as follows:

1. **Cohesion:** Each boid attempts to go towards the centre of its neighbourhood, i.e., get the average position of neighbouring boids and construct a vector towards that position.

$$\boldsymbol{Coh} = \frac{\sum_{i=1}^{N} \boldsymbol{p}_{b_i}}{N} - \boldsymbol{p}_{b_c},$$

Where $\boldsymbol{p}_{b_c}$ is the current position of the current boid, $\boldsymbol{p}_{b_i}$ is the current position of neighbouring boid i, and N is the number of neighbours.

2. **Alignment:** Each boid attempts to align its direction with its neighbours, i.e., get the average current direction of neighbouring boids.

$$\boldsymbol{Ali} = \frac{\sum_{i=1}^{N} \boldsymbol{d}_{b_i}}{N},$$

Where $\boldsymbol{d}_{b_i}$ is the current direction of neighbouring boid i, and N is the number of neighbours.

3. **Separation:** Each boid attempts to avoid collisions with nearby boids, i.e., get the average of the vectors opposite to the positions of neighbouring boids. This rule can be improved by decreasing the vision radius to half, and increasing the strength of the rule, so that the boids stay closer to each other while only avoiding immediate collisions.

$$\boldsymbol{Sep} = \frac{\sum_{i=1}^{n} (\boldsymbol{p}_{b_c} - \boldsymbol{p}_{b_i})}{n},$$

Where $\boldsymbol{p}_{b_c}$ is the current position of the current boid, $\boldsymbol{p}_{b_i}$ is the current position of neighbouring boid i, and n is the number of *close* neighbours, i.e., neighbours that are in half the normal vision radius.

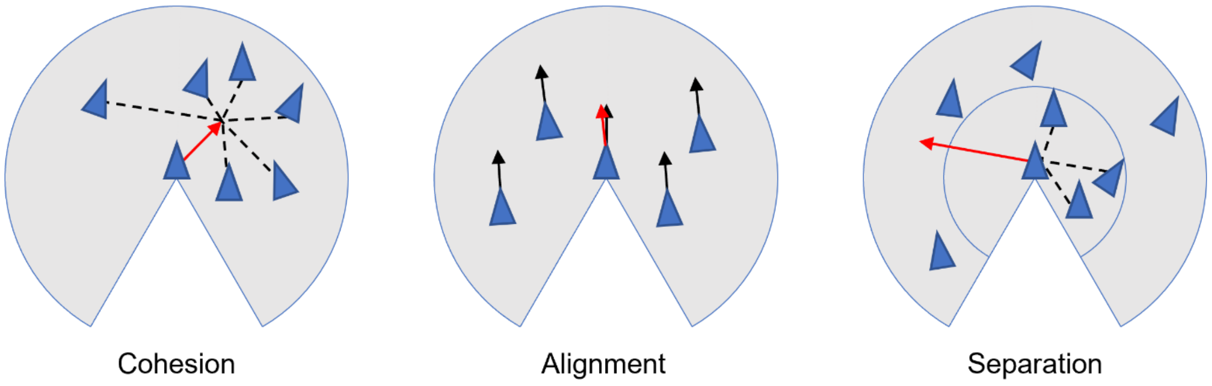


Figure 2.4 The main three Boids rules visualized. Red arrow represents the resulting direction vector.

4. **Simulation space boundary avoidance:** Each boid attempts to avoid travelling past the simulation bounds. This can be achieved by adding a vector perpendicular on the edge's plane, oriented inside the simulation. For example, if the object is close to the maximum x value of the simulation bounds, $\boldsymbol{avoidEdges} = \{-\mathbf{1}, \mathbf{0}, \mathbf{0}\}$ to influence its direction away from the boundary. It is important to choose a collision boundary that it is met before the visual boundaries of the simulation, so that the boids have a buffer zone in which they can turn back, still within the simulation space.

5. **Obstacle avoidance:** Each boid attempts to avoid colliding with obstacles in the simulation space. Obstacles can be conceptually represented as Bounding-Volumes

(Sulaiman et al., 2015), i.e., AABBs and spheres, which are able to encapsulate most objects that could exist in the simulation. Every boid should loop through every obstacle to check for collisions. In the case of spheres, the boid detects a collision if the distance between its position and the centre of the sphere is less than the sphere's radius. For axis-aligned boxes, a collision can be detected if the boid's xyz coordinates are between the coordinates of the two opposite corners of the box. If there is a collision, this rule should output a vector opposite to the centre of the obstacle. Similar to the simulation boundary avoidance, it is important to choose an appropriate collision boundary. The Bounding-Volume method was chosen as it is the least computationally expensive one out of the available options explored in **1.2.4 Related work**, while also being very suitable since high accuracy is unnecessary for this application.

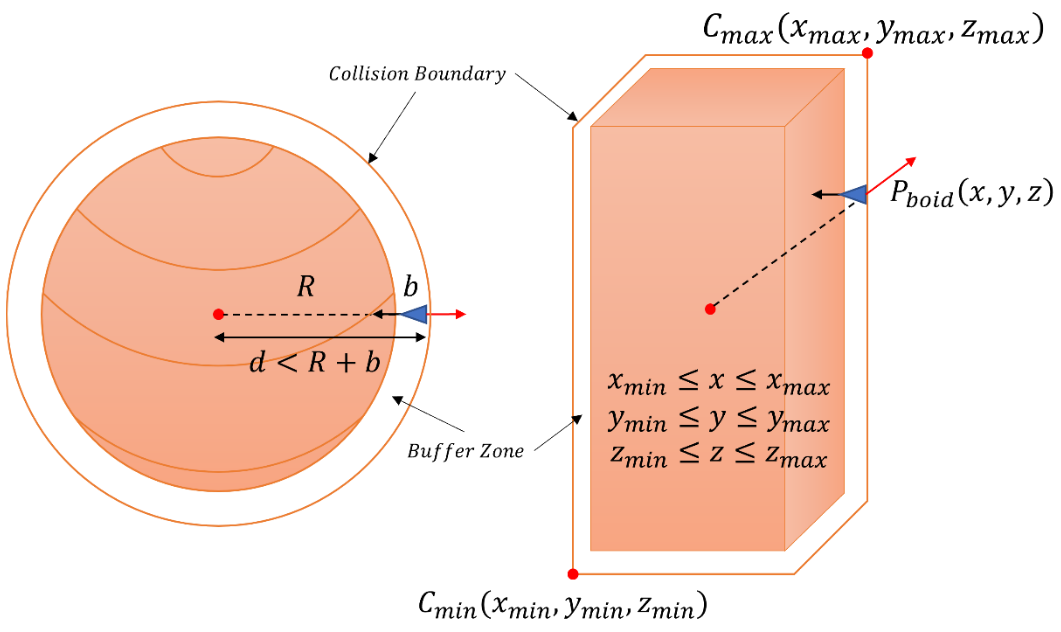

$C_{max}(x_{max}, y_{max}, z_{max})$
Collision Boundary
$P_{boid}(x, y, z)$
R
b
$d < R + b$
$x_{min} \leq x \leq x_{max}$
$y_{min} \leq y \leq y_{max}$
$z_{min} \leq z \leq z_{max}$
Buffer Zone
$C_{min}(x_{min}, y_{min}, z_{min})$

Figure 2.5 Collision detection for spheres and parallelepipeds

6. **User input:** The user can input either a direction or a position in the simulation space where they want the boids to go. This is simply an arbitrary normalized direction vector $\boldsymbol{userInput}$ that gets added to the current direction to influence the boids' movement. Since for this application, the fish do not need to act as coherent groups (it does not matter if they split up when avoiding an obstacle), guiding them can be implemented with this simple method that maintains a low computational overhead.

The vectors from the first 3 rules get normalized and multiplied with weight scalars w to set the strength for each rule. Afterwards, they get summed up with the current direction and user input vectors, as follows:

$$\boldsymbol{d}_{temp} = \boldsymbol{d}_c + w_c \cdot \frac{\boldsymbol{Coh}}{|\boldsymbol{Coh}|} + w_a \cdot \frac{\boldsymbol{Ali}}{|\boldsymbol{Ali}|} + w_s \cdot \frac{\boldsymbol{Sep}}{|\boldsymbol{Sep}|} + \boldsymbol{userInput}$$

After including the edge and obstacle avoidance vectors, the final formula for each boid's target direction vector is:

$$\boldsymbol{d}_t = \frac{\boldsymbol{d}_{temp}}{|\boldsymbol{d}_{temp}|} + w_{ae} \cdot \boldsymbol{avoidEdges} + w_{ao} \cdot \boldsymbol{avoidObstacles}$$

Since avoiding collisions must be a priority for boids, it is best that the obstacle and edge avoidance vectors are left out of the normalization. This way, if any avoidance is needed it will influence the boid's movement much more than the others and will result in immediate efforts to avoid such collisions, only barely influenced by other factors. This emulates the natural survival instinct of fish.

2.2 Environment Design

Since the simulation's purpose is to observe the emerging behaviour of fish, there should be enough opportunities for them to exhibit said behaviour. Therefore, there are a number of considerations to be addressed. First, the size of the simulation space should be appropriate for the number of boids. Next, there should be enough obstacles throughout the environment so that it diversifies their movement and provides opportunities for separating large groups, so that new ones can emerge. Finally, it is important to maintain a balance between empty spaces and obstacles to ensure that the boids have sufficient time to form new groups, while also preventing them from persisting too long, which could lead to monotonous behaviour.

2.3 User Interface Design

A crucial aspect of any interactive simulation is the UI/UX design. It should provide an intuitive, easy to understand interface that allows the user to customize the output in some capacity. In the case of this simulation, the intention is to give the user some control over the parameters of the algorithm, and to provide ways to view and interact with the 3D scene.

2.3.1 Camera controls

There are many standard ways to control the camera in a 3D environment, and it is up to the developer to decide on the most appropriate ones. In the case of this simulation, the user should have a variety of options that provide a pleasant experience. The cameras that have been chosen for the implementation are:

1. **Locked Angle Arc-ball Camera:** It is placed at a distance to the world origin, with a fixed $30°$ rotation on the X-axis and an arbitrary rotation on the Y-axis, controlled by the user. This allows rotating the scene left and right around the origin so it can be viewed from different angles. It also allows zooming in/out by decreasing/increasing the distance to the origin.

2. **Top-Down View Camera:** This camera follows the same concept as the locked angle arc-ball, except it has a fixed $90°$ rotation on the X-axis, creating a top-down view of the simulation, which can be useful in visualizing the whole scene at once.
3. **Fly-through Camera:** The camera uses the principles of a LookAt matrix to create a camera view that can move and rotate freely within the simulation space, thus allowing the user to view any detail from any angle.
4. **Third-person Camera:** This uses the principles of an arc-ball camera, but centred on an arbitrary fish in the simulation, so the user can follow its behaviour more closely. It allows rotation on both X and Y axes and zoom in/out, while activating the camera again changes the currently viewed fish.

For ease of use, all cameras also allow moving faster/slower to enable users to cover large distances quickly or to manoeuvre around objects to observe finer details.

A link has been provided with a demo video, where the different camera setups are showcased.

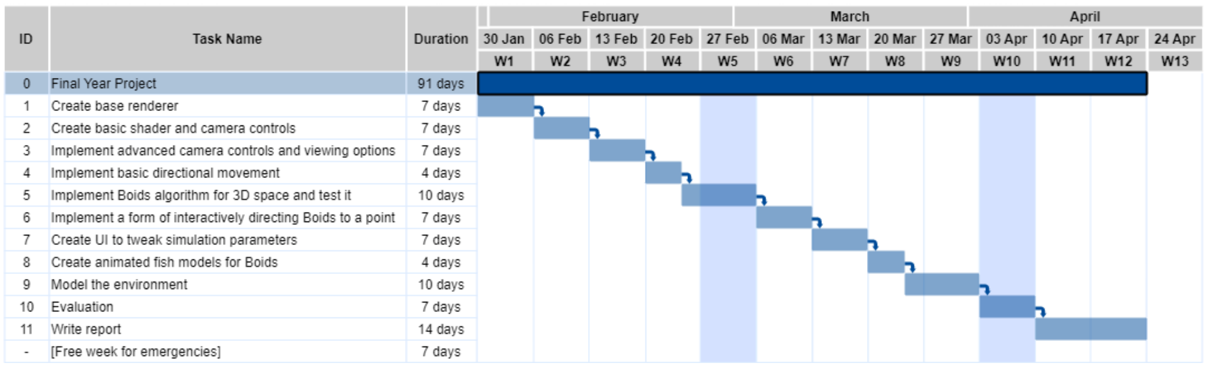
2.3.2 GUI Design

The interactive element of the simulation relies in the GUI provided to the user, that they can use to modify simulation parameters or input a location or direction for the boids. The interface follows design principles generally used in Interaction Design, such as consistency, simplicity, feedback, responsiveness, flexibility, and accessibility, as will be seen in **Chapter 3 Results**.

The GUI includes sliders and buttons for adjusting the strengths of the *three main rules* and other parameters, such as boid count, speed, vision angle and vision range, or options to pause/play the simulation. Moreover, it includes comprehensive instructions for camera usage, and an interface for controlling the boids through the $\boldsymbol{userInput}$ element. This interface includes sliders for the three axes, and a mouse drag-and-drop option for modifying the location of a ball that signifies the point in space that attracts the boids. Most buttons also have a keyboard shortcut for ease of use and accessibility.

2.4 Project Management

Due to the nature of the project, a waterfall methodology was used for its management. This was the most appropriate choice since the development of the application involved a linear path of building each element on top of previous work. The algorithm cannot be implemented without the base of geometric flight, which needs a rendered model, which in turn requires the rendering setup. Even though there is always a higher risk associated with a waterfall methodology compared to agile methodologies, having a risk mitigation plan addresses this issue. Therefore, with a good and precise initial plan (**Figure 2.6**), this led to a successful outcome.



ID	Task Name	Duration	February				March				April				
			30 Jan	06 Feb	13 Feb	20 Feb	27 Feb	06 Mar	13 Mar	20 Mar	27 Mar	03 Apr	10 Apr	17 Apr	24 Apr
			W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13
0	Final Year Project	91 days	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	
1	Create base renderer	7 days	▨												
2	Create basic shader and camera controls	7 days		▨											
3	Implement advanced camera controls and viewing options	7 days			▨										
4	Implement basic directional movement	4 days				▨									
5	Implement Boids algorithm for 3D space and test it	10 days				▨	▨								
6	Implement a form of interactively directing Boids to a point	7 days						▨							
7	Create UI to tweak simulation parameters	7 days							▨						
8	Create animated fish models for Boids	4 days								▨					
9	Model the environment	10 days								▨	▨				
10	Evaluation	7 days										▨			
11	Write report	14 days											▨	▨	
-	[Free week for emergencies]	7 days													

Figure 2.6 Gantt Chart for Final Year Project plan.

2.4.1 Version Control

Throughout the entire project, a version control system (GitHub) was consistently used. The work was done on a single branch since it didn't require developing multiple features at once. Still, it was managed through the use of Issues, created for each task in the plan, and Milestones, used to group related issues and set goals for the project. Evidence of version control can be found in the Commits section of the GitHub repository.

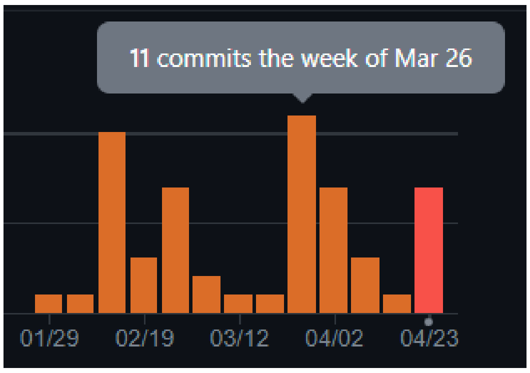


Figure 2.7 Commit distribution over planned timeframe (57 total commits).

Chapter 3
Results

3.1 Implementation Description

The software was implemented in C++ with OpenGL for handling graphics, GLFW for handling user inputs and application output, rapidobj for loading objects from .obj files and ImGui for the GUI setup. The implementation of the simulation involved programming not only the logic for Reynold's algorithm (1987) and other techniques used, but also the basis of a 3D computer graphics application, i.e., a working renderer with shaders and definitions of mesh objects. Additionally, it involved designing and modelling in Blender the meshes used in the implementation or procedurally generating them in code.

3.1.1 Software Architecture

The implementation follows an Object-Oriented Programming (OOP) approach, as it uses classes to define objects and their behaviour, and principles like polymorphism and inheritance for reusing code and creating a hierarchy of related objects.

The Shader class can set up and initialize any shader from a given pair of *.vert* and *.frag* files, written in GLSL, that define vertex and fragment shaders. The main shaders are set up to handle a corrected Blinn-Phong lighting model with one or more materials, while a different shader is used to implement a skybox, using a Cube map texture.

The Model class is the backbone of the application since it handles the creation and rendering of all objects in the scene. The mesh data (vertex positions, vertex normals and materials) can either be imported from an *.obj* file or generated procedurally with some defined functions. This is then used to create vertex buffer objects (VBOs) that store the data in memory managed by OpenGL and vertex array objects (VAOs) that describe the input data for the shaders. The meshes can then be transformed within the simulation space by initializing their *model2world* matrix, and rendered on screen using the specified shader, light values, camera position, and mesh's materials. A model can either have one material, in which case it sent to the Single Material Shader as a uniform variable, or multiple materials sent to the Multiple Material Shader as an array of uniforms along with a material index VAO.

The Boid class is another crucial part of the application since it contains the logic for the implementation of the geometric flight and the algorithm's rules. A Boid object is defined as a position in space, a *model2world* matrix that defines its transforms (rotation, translation, and shear for animation), and a current and target direction. It is an abstract object that needs a model to be rendered, and therefore, can be applied to a model of the developer's choice.

The Boid objects are initialized to a random position inside the simulation space, but outside all the obstacles, and they keep a reference of the obstacles to use for the collision avoidance methods. The class also contains methods that compute the neighbours in a boid's vision range, and based on those, the cohesion, alignment and separation vectors.

Obstacle is an abstract class that represents the Bounding-Volumes around the actual models used for the obstacles. It contains a virtual method for collision detection, that gets inherited and defined differently by the SphereObstacle and BoxObstacle classes, and also a pointer to a model that is used visually represent the bounding volumes in the technical view of the application.

Lastly, the Cubemap class handles the texture setup and rendering of a skybox, using a cube map texture from 6 images (up, down, left, right, front, back). It is a variant of the Model class, as it sets up the VBOs and VAOs required and has a render method, but it is structured differently since it only needs the vertex positions of a cube, and a different *view* matrix to be rendered.

The architecture of the application is represented in the UML Class Diagram in **Figure 3.1**.

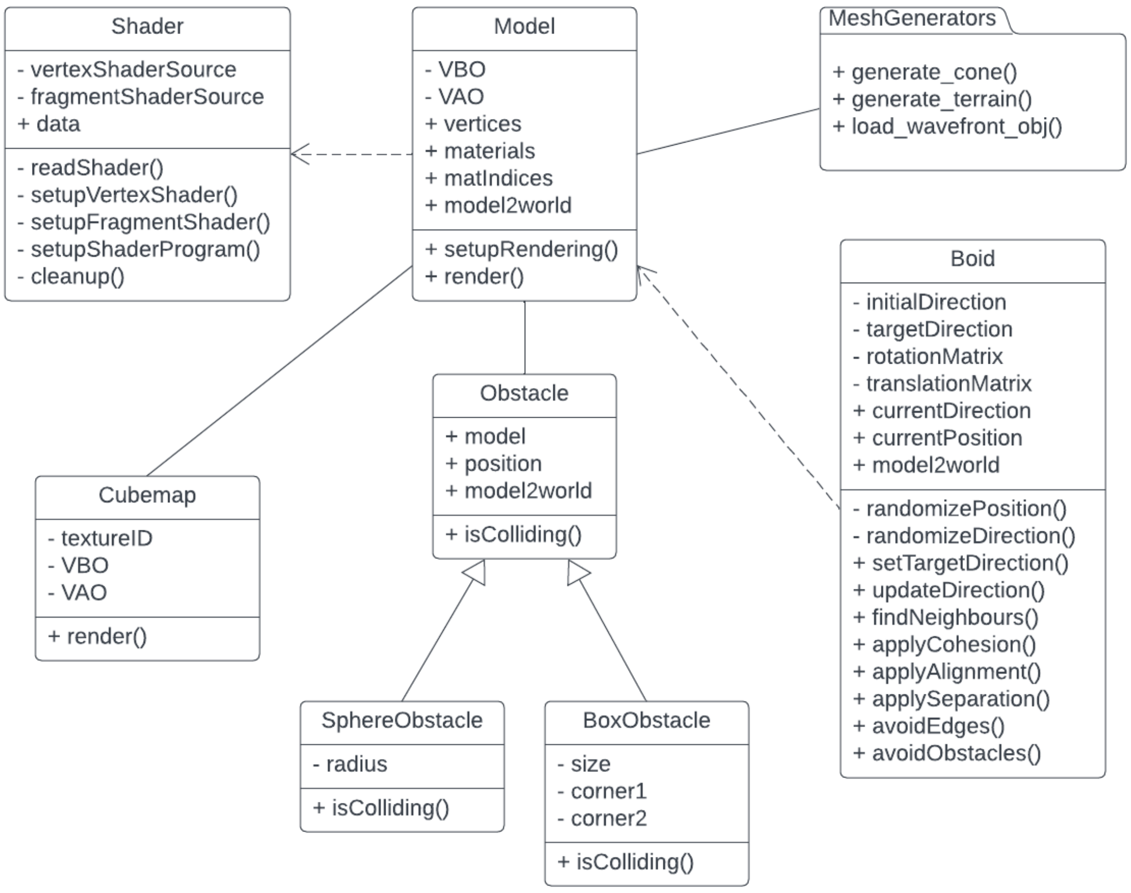


Figure 3.1 UML Class Diagram of the application

3.1.2 Model Design

Models are a crucial part of computer graphics simulations that directly impact the quality and realism of the final output. 3D assets can usually be found online for free or even bought, but for the purpose of this project, I decided to create my own. Therefore, the models necessary to run this simulation, i.e., fish, obstacles and terrain were either designed in Blender or procedurally generated. The modelled fish, columns and rocks can be seen in **Figure 3.2** along with reference images used. A cyclical oscillating shear transformation was also applied to the fish while the simulation is running, creating a swimming animation for the tail fin.

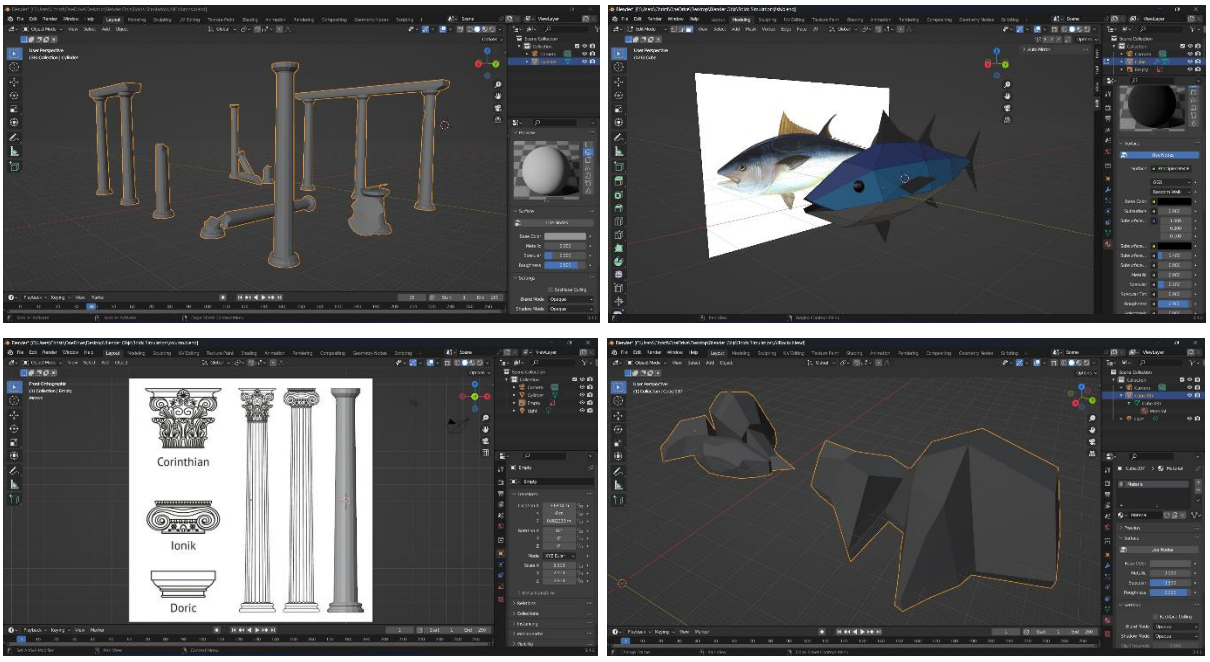


Figure 3.2 Blender screenshots of modelled meshes.

The cone and terrain models were generated in code with procedural generation techniques. For the cones, the vertex vector was created by adding vertex positions and normals along the perimeter of a circle and connecting them with the centre of the circle to form a disc of triangles. The same vertices then get connected with another point at a unit distance perpendicular to the disc, to form the cone's body.

The terrain is generated using a heightmap, by getting the pixel values from the black and white image and using those values to form height fields ($h = f(x, y)$), as explained by Ong et al., 2005. These are then used to create the vertex positions with x and z coordinates corresponding to the pixel coordinates, and the y coordinate based on the height value. Using these values, a triangle mesh is created from strips of triangles, as depicted in **Figure 3.3**. The normal values required for shading are then computed by defining two vectors from the edges of each triangle and calculating the cross product of those vectors, to get a vector perpendicular to the triangle's face. This normal value is then assigned to each of the three vertices in the triangle. The resulting terrain mesh can then be scaled to the preferred size.

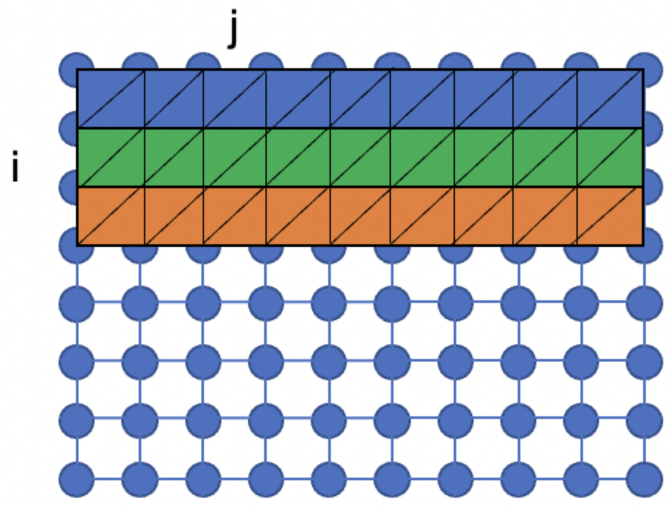


Figure 3.3 Terrain mesh generation from triangle strips (Paone, 2021).

3.1.3 Lighting Model

The lighting model used for the main shaders of the application is a corrected Blinn-Phong model, following the formula (Billeter, 2022):

$$I_p = k_a I_a + k_d \frac{I_l}{\pi} \left(\hat{N} \cdot \hat{L}\right)_+ + \frac{\alpha' + 2}{8} k_s I_l \left(\hat{H} \cdot \hat{N}\right)_+^{\alpha'} + I_e, \text{ where:}$$

- I_p represents the output colour of a fragment.
- $k_a I_a$ is the ambient term, a mix between the material's ambient attribute and the ambient light.
- $k_d \frac{I_l}{\pi} \left(\hat{N} \cdot \hat{L}\right)_+$ is the diffuse term, a mix between the diffuse material attribute, the light's colour I_l (common value for diffuse and specular), and the clamped dot product of the normalized surface normal and light direction.
- $\frac{\alpha' + 2}{8} k_s I_l \left(\hat{H} \cdot \hat{N}\right)_+^{\alpha'}$ is the specular term, a mix between the specular attribute of the material, the light's colour, a shininess parameter (α') and the clamped dot product of the normalized half vector (oriented halfway between the light and view vectors) and surface normal.
- I_e represents the emissive light.

Blinn-Phong was chosen since it is a commonly used lightning model that adds sufficient detail to the scene without focusing too much computation power on the shading aspect of the application. It is an effective model for rendering smooth, reflective surfaces, such as those of the fish, columns or rocks in this simulation, and results in a balance between performance and visual quality.

3.1.4 Software validation/testing

Validation and testing for graphics applications is not as straightforward as testing other applications. Since the purpose of most methods defined is to either create models and send their data to the shaders, or to create some sort of movement or rotation, a visual analysis is more suitable. Therefore, the main approaches used throughout the project were visual and performance testing, i.e., testing whether the result fits our expectations of natural behaviour and running benchmarks to inspect the rendering time of a frame (ms/frame) or the frames per second (FPS). The application was extensively tested for visual bugs related to the smoothness of the movement or correct orientation of the fish models, and many adjustments have been made to fix said bugs. In particular, the implementation of geometric flight has been rigorously tested in isolation, with a single model having to change direction and analysing how it performs, along with many edge cases.

3.2 Implementation Evaluation

3.2.1 Results

The resulting simulation meets the expectations of realism, as the fish exhibit emerging behaviour while attempting to avoid collisions with each other or with obstacles and terrain. Very rare exceptions have been observed when there is a large number of boids in a small area near obstacles and against their best efforts, a few of the ones at the flock's boundaries are forced to collide since they cannot escape otherwise. But in the usual case of a moderate number of fish appropriate for the size of the environment, they maintain their formations while successfully avoiding collisions.

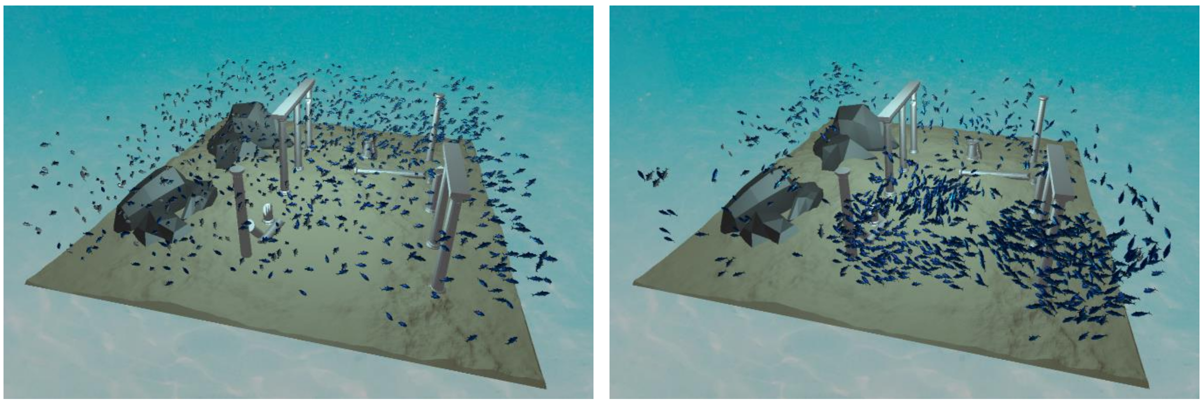


Figure 3.4 (left) Fish initialized in random positions. (right) Formed fish schools after a few seconds of running the simulation.

Adjusting the simulation's parameters so that only one of the three main rules is turned on at a time can help in visualizing the effect of each one. Additionally, the bounding volumes and the terrain's skeleton can be seen when turning on the technical mode of the application. Examples of these can be found in **Figure 3.5**.

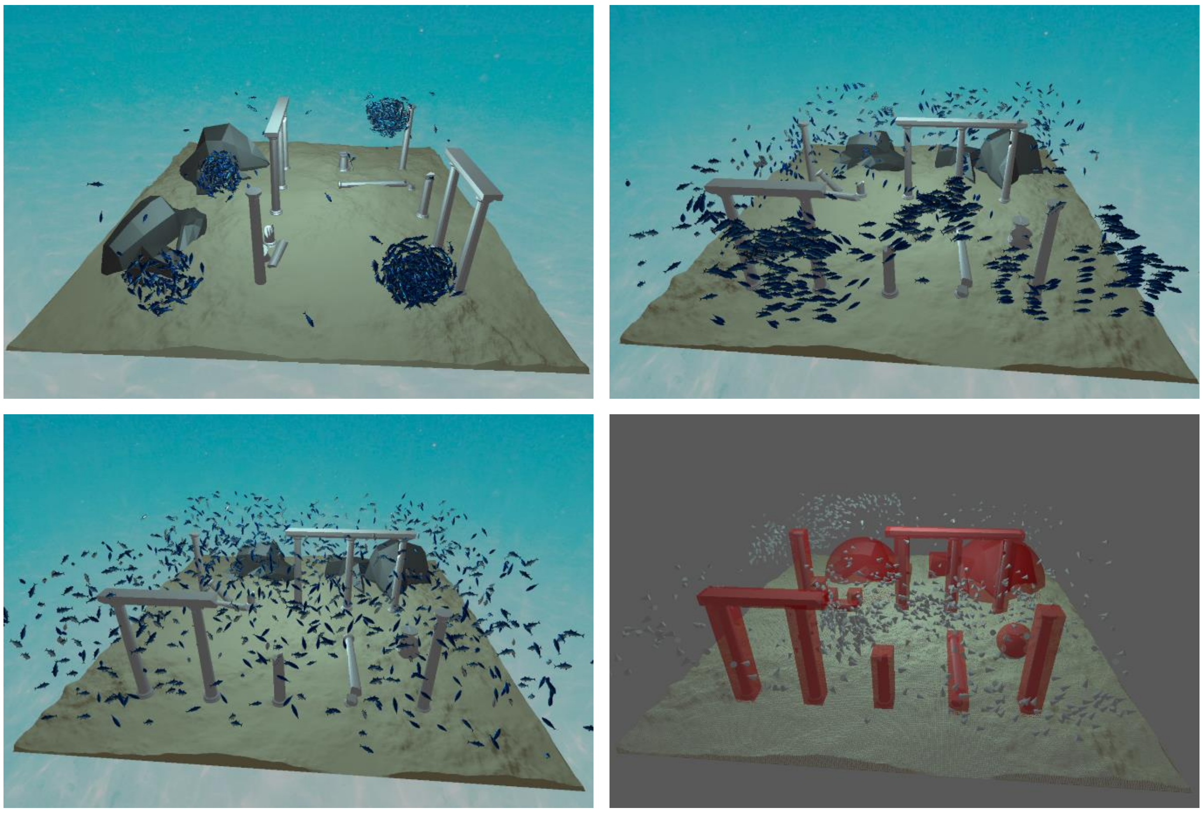


Figure 3.5 (top left) Cohesion rule visualized. (top right) Alignment rule visualized. (bottom left) Separation rule visualized. (bottom right) Technical view of the simulation.

An interesting result appeared when directing boids to a point in space. Since they have to continuously move and they do not stop at the target point, they exhibit a swarming behaviour of circling around the point over and over, as if they have come across a food source and want to determine whether it is safe to eat. Additionally, when directed towards an arbitrary direction, they exhibit a migration behaviour, that would be even more apparent in a larger simulation space. A visualization of both behaviours can be seen in **Figure 3.6**.

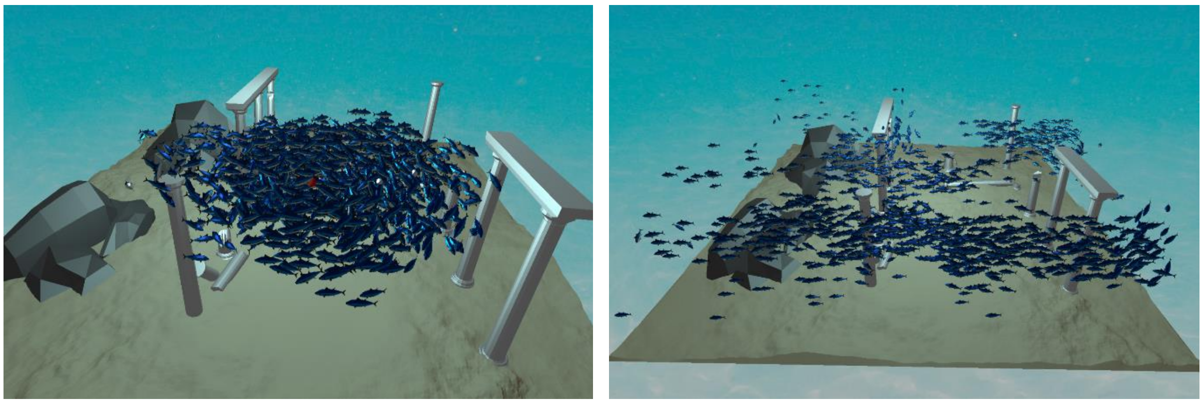


Figure 3.6 (left) Fish circling target point. (right) Fish migrating towards a direction.

3.2.2 Performance Analysis

3.2.2.1 Experiment Setup

The tests for performance analysis were run on a Windows 11 Home x64 with an 11th Gen Intel® Core™ i7-11800H @ 2.30GHz processor, with 32GB RAM, and an NVIDIA GeForce RTX 3080 Laptop GPU. The exact specifications of the CPU and GPU can be seen in **Table 3.1**. Since these can be regarded as high-end hardware, the results of the performance analysis may not accurately reflect the software's behaviour on typical configurations, but they are still useful in providing insights of any issues.

CPU		**GPU**	
Specification	**Value**	**Specification**	**Value**
Total Cores	8	NVIDIA CUDA Cores	8960 / 8704
Total Threads	16	Boost Clock	1.71 GHz
Max Turbo Frequency	4.60 GHz	Memory Size	12 GB / 10 GB
Cache	24 MB	Memory Type	GDDR6X
Bus Speed	8 GT/s		

Table 3.1 Specifications of CPU and GPU used for performance testing and analysis.

3.2.2.2 Experiment 1: Memory Utilization Benchmarking

The goal of the first experiment is to get an overview of the memory and processing usage. After performing a diagnostic with Visual Studio 2022's built-in tool, the following results have been obtained. The average CPU usage of the application was at an average 5% of all processors, while the GPU utilization measured around 10-20% when there are no fish, and around 50-60% when there are up to 3000 fish. The application also had an average process memory usage of 200MB. All of these results are in line with other similar computer graphics applications. A timeline of these measurements over the 43 second diagnostic runtime can be seen in **Figure 3.7**.

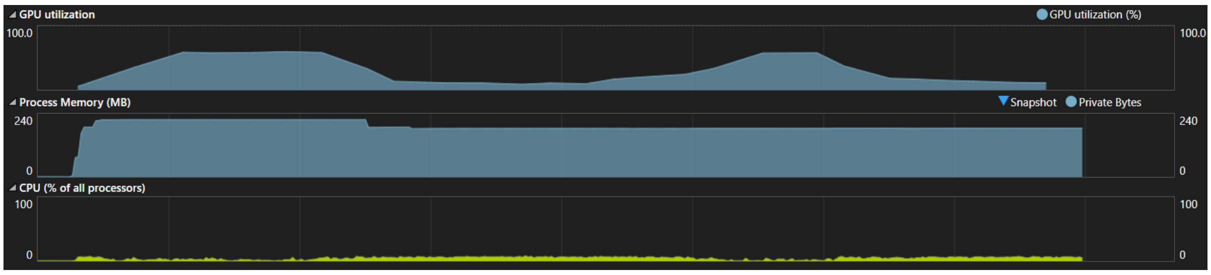


Figure 3.7 Diagnostic results timeline over 43 seconds.

To avoid memory leaks, dynamic allocation was implemented through the use of C++'s *new* and *delete* keywords. This decision is reflected in the resulting process memory usage.

Out of the total CPU usage time, three functions measured a substantial usage.

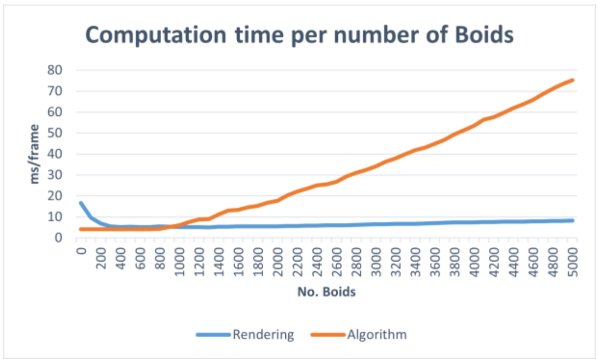
- void ImGui_ImplOpenGL3_RenderDrawData(ImDrawData* draw_data) – ImGUI's function for rendering the UI component measured a **30.01% usage**.
- std::vector<Boid*> Boid::findNeighbours(std::vector<Boid*> totalBoids, float radius, float visionAngle) – the function in the algorithm that loops through all boids to find neighbours measured a **20.16% usage**.
- void Model::render(Vec3f cameraPosition, Light light, Mat44f world2projection, Mat44f givenModel2world, GLuint shaderProgs[]) – which was used for rendering all models measured a **19.97% usage**.

Note: All other functions measured a usage lower than 10%.

These results were anticipated since they are the functions that perform the most computation and allocations, which, except for the first, are also called multiple times for every single frame.

3.2.2.3 Experiment 2: Processing Time and Frame Rate

The goal of this experiment is to test the application's processing and rendering times. The key metrics to be analysed are the time to render a frame (ms/frame) and the frame rate (FPS). The FPS was measured using ImGUI's built-in function for measuring frame rate (ImGUI::GetIO().Framerate), and the time per frame is the result of dividing 1000 by the FPS. The performance tests have been executed while the application was running with the algorithm paused, and with it on. While paused, the fish are simply rendered in their initial position, and the functions that apply the algorithm's rules are not called. Therefore, the paused version acts as a base processing time for rendering the scene, that can be compared to the running time of the simulation. The results can be seen in the graphs in **Figure 3.8**.



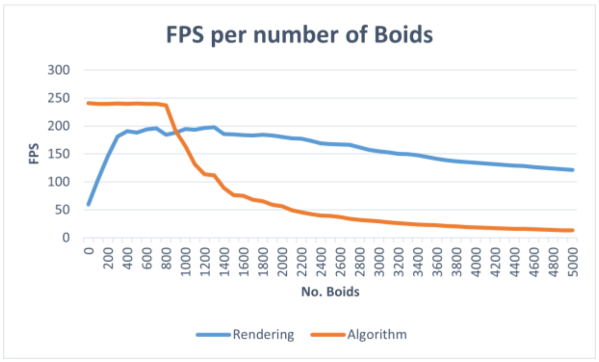


Figure 3.8 Comparison of rendering (blue) and algorithm (orange) processing time. (left) ms/frame. (right) FPS. Note: The second chart starts with a constant value of 240 FPS for the algorithm, since the tests were performed on a device with a 240 Hz monitor.

As expected, the results show a quadratic increase in the computation time, due to the problems discussed in **Chapter 1 Introduction and Background Research**, where any particle system in which every individual interacts with the others, will need to perform at least one loop through all other individuals to decide if they are nearby. Since there were no space-partitioning optimizations done to reduce the computation to an $O(N \log N)$, a complexity of $O(N^2)$ is the best that can be achieved.

3.2.3 Complexity Analysis

The application's execution has two different parts, the initialization of shaders, models, and everything that has to be loaded at runtime, and the main rendering loop in which the interactions are computed, and the preloaded assets are used. Therefore, the most important operations performed by the simulation are those of the algorithm, whose implementation can be seen in the pseudocode in **Figure 3.9**.

Boids Algorithm *(Note: N(b) refers to a boid b's neighbourhood, cN(b) to b's close neighbourhood, pos(b) to b's positon, dir(b) to b's direction)*

```
1    B ← vector of Boids
2    O ← vector of obstacles
3    for b in B do
4        for n ∈ B\{b} do                // Find neighbours
5            if n in b's vision range then
6                N(b) ← N(b) ∪ {n}
7        for n ∈ N(b) do                 // Cohesion
8            coh(b) ← coh(b) + pos(n)
9        coh(b) ← coh(b) / |N(b)| - pos(b)
10       for n ∈ N(b) do                 // Alignment
11           ali(b) ← ali(b) + dir(n)
12       ali (b) ← ali(b) / |N(b)|
13       for n ∈ N(b) do                 // Separation
14           if n in b's half vision range then
15               cN(b) ← cN(b) ∪ {n}
16       for n ∈ cN(b) do
17           sep(b) ← sep(b) + (pos(b) – pos(n))
18       sep(b) ← sep(b) / |cN(b)|
19       if b is at the edge of the simulation then initialize avoidEdge(b)
20       for o ∈ O do
21           if b is colliding with o then
```

```
        avoidObstacles(b) ← avoidObstacles(b) + pos(b) – pos(o)
targetDirection(b) ← normalize(dir(b) + coh(b) + ali(b) + sep(b) + userInput) +
```

Figure 3.9 Boids algorithm pseudocode

The most expensive loop is the one used to *find neighbours*, which for a number of N boids, performs $N \cdot (N-1)$ comparisons of the distance between the pair of boids and the vision range. Since in a moderately large simulation space, fish are likely to have a small fraction of the flock as neighbours, the loops used for the rules, that perform computations with every neighbour of the current boid, don't contribute much to the complexity of the algorithm. Additionally, the *for* that loops through the obstacles is also dependent on the number of obstacles, but that number is a constant M, therefore it can also be discarded. In the end, the complexity for this implementation can be computed as $O(N^2)$.

3.2.4 Interaction Design Analysis

As explained in **Section 2.3 User Interface Design**, an important aspect of an interactive simulation is the GUI, that should follow interaction design principles. Therefore, it should be evaluated accordingly to ensure that it meets the users' needs and expectations in terms of usability.

One option for evaluating the design is by performing an expert review by the application's developer. Adequate metrics for an Interaction Design evaluation can be found in Nielsen's Heuristics for ID (Nielsen, 1994), numbered from 1 to 10:

1. *Visibility of system status,*
2. *Match between system and the real world,*
3. *User control and freedom,*
4. *Consistency and standards,*
5. *Error prevention,*
6. *Recognition rather than recall,*
7. *Flexibility and efficiency of use,*
8. *Aesthetic and minimalist design,*
9. *Help users recognize, diagnose, and recover from errors,* and
10. *Help and documentation.*

These can be applied to inspect every feature of the UI and record any findings such as features that follow the heuristics, or issues measured with a severity of *low, medium,* or *high*.

The results of the evaluation show that the GUI is of high standard, since all features follow one or more heuristics, as outlined in **Table 3.2**, but it still has some low severity issues that could be addressed in the future.

#	**Findings**	**Heuristics followed**	**Heuristics violated (if any)**
#1	The base colour palette from ImGUI is consistent, with good contrast all throughout the GUI, making it accessible.	**4, 7, 8.**	**-**
#2	The buttons and sliders all have a tag with a suggestive and easy to understand tag.	**4, 5, 6, 10.**	**-**
#3	There are instructions provided for controls both inside and outside the GUI.	**3, 5, 6, 10.**	**-**
#4	All the buttons that have an alternative key binding, include it in the tag.	**1, 2, 4, 6, 7, 10.**	**-**
#5	Buttons and sliders respond to mouse hover or press, by changing colour.	**1, 3, 4, 5, 6, 7.**	**-**
#6	The interactive options are organized into "accordions" (term commonly used to refer to a section that can expand or collapse).	**3, 6, 7, 8.**	**-**
#7	The sections for parameter changes include *default* buttons for reverting to the original state.	**3, 4, 7, 9.**	**-**
#8	The slider for Separation has a different default value than for the other rules since it also differs in implementation.	**-**	**4 *(low)*.**
#9	The values for the sliders don't have units of measurement, which might lead to confusion about their effect.	**8.**	**5 *(low)*.**

Table 3.2 Heuristic Evaluation using Nielsen's Heuristics.

Chapter 4
Discussion

4.1 Conclusions

All the objectives of this project, set during the planning phase, have been successfully completed. This includes thorough research of the problem and potential solutions, a meticulous description of the methods used, a comprehensive analysis of the results that proves the effectiveness of this approach, and a tested implementation that works as intended and provides the user with great control over the application.

Therefore, the final product is a simulation that successfully showcases Reynolds' Boids algorithm (1987) by producing emergent behaviour, and is also presented in a good aesthetic, adding to the intended realism and user engagement. The fish behave naturally in their given environment, basing their movement on their perception of neighbouring fish and surroundings. Despite not including many optimizations, the simulation has a stable performance when running with up to 3000 fish, which is enough to create a believable scene within the size of the simulation space.

Still, even though the outcome was successful, there remains work to be done to improve the simulation.

4.2 Ideas for future work

First, the simulation could have been optimized in various ways with some more time. As outlined in the first chapter, space partitioning would have been especially helpful and could be a pathway for future work since it would greatly improve the performance and scalability of the application. Once this is implemented, parallelization would be another important improvement that would further optimize the performance, allowing for up to tens of thousands of boids to interact at the same time (Da Silva, 2010).

Second, to address the realism limitations of the algorithm, more factors could be taken into consideration. For example, implementing features like predator-prey relationships, multiple species of fish only grouping with their own, or other more advanced social behaviours would result in a much more realistic simulation.

Third, since the application was programmed with OOP principles in mind and has a modular base, it could easily be used to create scenes in animated movies, with little work on adjusting all the necessary parameters, such as replacing any imported models or moving the lighting source.

Finally, an interesting path for continuing the development would be to transform it into a game. In its current form, the simulation has the setup for a third person camera for fish, that could be instead attached to an interactive object, like a predatory fish or a submarine, that the user can control with a set of key bindings. This could be used, along with other features like a point system, pre-programmed quests with objectives, and a larger map, to create a fun and engaging game.

List of References

Akenine-Möller, T., Haines, E. and Hoffman, N. 2008. *Real-Time Rendering*. [Online]. 3rd ed. Natick, Massachusetts: A K Peters, Ltd. [Accessed 13 April 2023]. Available from: https://www.vlebooks.com/Product/Index/796345.

Alaliyat, S., Yndestad, H. and Sanfilippo, F., 2014. Optimisation of Boids Swarm Model Based on Genetic Algorithm and Particle Swarm Optimisation Algorithm (Comparative Study). In: *Proceedings of the 28th European Conference on Modelling and Simulation (ECMS), May 2014, Brescia, Italy*. [Online]. Brescia, Italy: ECMS, pp. 643-650. [Accessed 10 April 2023]. Available from: https://www.scs-europe.net/dlib/2014/ecms14papers/simo_ECMS2014_0062.pdf.

Barnes, J. and Hut, P. 1986. A hierarchical O(N log N) force-calculation algorithm. *Nature*. **324**, pp.446–449.

Batman Returns. 1992. [Film]. Tim Burton. dir. United States: Warner Bros., Polygram Pictures.

Billeter, M. 2022. Lighting (Blinn-Phong), Normals. COMP3811 Computer Graphics. 28 November, University of Leeds.

Da Silva, A. R., Lages, W. S., and Chaimowicz, L. 2010. Boids that see: Using self-occlusion for simulating large groups on GPUs. *Computers in Entertainment*. **7**(4), pp.1-20.

Dai, J. S. 2015. Euler-Rodrigues formula variations, quaternion conjugation and intrinsic connections. *Mechanism and Machine Theory*. **92**, pp.144-152.

Darve, E., Cecka, C. and Takahashi, T. 2011. The fast multipole method on parallel clusters, multicore processors, and graphics processing units. *Comptes Rendus Mécanique*. **339**(2-3), pp.185–193.

Half Life. 1998. Windows [Game]. Sierra Studios: Los Angeles, USA.

Hermann, E., Faure, F. and Raffin, B. 2008. Ray-traced Collision Detection for Deformable Bodies. In: *3rd International Conference on Computer Graphics Theory and Applications, GRAPP 2008, January 2008, Funchal, Madeira, Portugal*. [Online]. Portugal: Institute for Systems and Technologies of Information, Control and Communication, pp.293-299. [Accessed 10 April 2023]. Available from:

https://www.researchgate.net/publication/29623005_Ray-traced_collision_detection_for_deformable_bodies.

Hussleman, A. V. and Hawick, K. A. 2012. Spatial Data Structures, Sorting and GPU Parallelism for Situated-agent Simulation and Visualisation. In: *Proc. Int. Conf. on Modelling, Simulation and Visualization Methods (MSV'12), January 2012, Las Vegas.* [Online]. Las Vegas, NV, USA: The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), p.1. [Accessed 10 April 2023]. Available from: https://www.researchgate.net/publication/266261197_Spatial_Data_Structures_Sorting_and_GPU_Parallelism_for_Situated-agent_Simulation_and_Visualisation.

Jackins, C.L. and Tanimoto, S.L. 1980. Oct-trees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing.* **14**(3), pp.249–270.

Kamphuis, A. and Overmars, M. H. 2004. Finding paths for coherent groups using clearance. In: *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation, August 2004, Grenoble, France.* [Online]. Goslar, DEU: Eurographics Association, pp.19-28. [Accessed 10 April 2023]. Available from: https://dl.acm.org/doi/10.1145/1028523.1028526.

Klow. 2009. *Standard Boids on Xen in Half-Life*. [Online]. [Accessed 10 April 2023]. Available from: https://half-life.fandom.com/wiki/Boid.

Kravtsov, A.V., Klypin, A.A. and Khokhlov, A.M. 1997. Adaptive refinement tree: A new high-resolution n-body code for cosmological simulations. *The Astrophysical Journal - Supplement Series.* **111**, pp.73–94.

Macal, C. M. and North, M. J. 2005. Tutorial on agent-based modeling and simulation. In: *WSC '05: Proceedings of the 37th conference on Winter simulation, December 2005, Orlando, FL, USA.* [Online]. Orlando: Winter Simulation Conference, pp.2-15. [Accessed 10 April 2023]. Available from: https://dl.acm.org/doi/10.5555/1162708.1162712.

Miller, G. S. P. 1986. The definition and rendering of terrain maps. In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques (SIGGRAPH '86), August 1986, Dallas, TX, USA*. [Online]. New York, NY, USA: Association for Computing Machinery, pp.39–48. [Accessed 14 April 2023]. Available from: https://doi.org/10.1145/15922.15890.

Nielsen, J. 1994. *Usability Engineering*. [Online]. San Francisco, CA, United States: Morgan Kaufmann Publishers Inc. [Accessed 19 April 2023]. Available from: https://dl.acm.org/doi/book/10.5555/2821575.

Ong, T. J., Saunders, R., Keyser, J. and Leggett, J. J. 2005. Terrain generation using genetic algorithms. In: *Proceedings of the 7th annual conference on Genetic and evolutionary computation (GECCO '05), June 2005, Washington DC, USA.* [Online]. New York, NY, USA: Association for Computing Machinery, pp.1463-1470. [Accessed 11 April 2023]. Available from: https://dl.acm.org/doi/10.1145/1068009.1068241.

Paone, J. 2021. *Tessellation Chapter I: Rendering Terrain using Height Maps.* [Online]. [Accessed 14 April 2023]. Available from: https://learnopengl.com/Guest-Articles/2021/Tessellation/Height-map.

Reynolds, C. W. 1987. Flocks, Herds, and Schools: A Distributed Behavioral Model. In: *Proceedings of the 14th annual conference on Computer graphics and interactive techniques (SIGGRAPH '87), July 1987, New York, USA*. [Online]. New York: Association for Computing Machinery, pp.25-34. [Accessed 10 December 2022]. Available from: https://dl.acm.org/doi/10.1145/37401.37406.

Ros, A. L., Chow, K., Geckler, J., Joseph, N. M. and Nghiem, N. 2021. Populating the World of Kumandra: Animation at Scale for Disney's "Raya and the Last Dragon". In: *SIGGRAPH '21: ACM SIGGRAPH 2021 Talks, July 2021, Virtual Event, USA.* [Online]. New York: Association for Computing Machinery, pp.1-2. [Accessed 10 April 2023]. Available from: https://dl.acm.org/doi/10.1145/3450623.3464648.

Rose, T.J. and Bakaoukas, A.G. 2016. Algorithms and approaches for procedural terrain generation-a brief review of current techniques. In: *2016 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES), September 2016, Barcelona, Spain.* [Online]. New York, USA: Institute of Electrical and Electronics Engineers (IEEE), pp.1-2. [Accessed 13 April 2023]. Available from: https://ieeexplore.ieee.org/abstract/document/7590336.

Saska, M., Vonasek, V., Krajník, T. and Preucil, L. 2012. Coordination and navigation of heterogeneous UAVs-UGVs teams localized by a hawk-eye approach. In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, October 2012, Vilamoura, Algarve, Portugal*. [Online]. New York, USA: Institute of Electrical and Electronics Engineers (IEEE). [Accessed 13 April 2023]. Available from: https://www.researchgate.net/publication/261495650_Coordination_and_navigation_of_heterogeneous_UAVs-UGVs_teams_localized_by_a_hawk-eye_approach.

Stanley and Stella in: Breaking the Ice. 1987. [Film]. Larry Malone. dir. United States: Symbolics Whitney / Demos Productions.

Sulaiman, H.A., Othman, M.A., Aziz, M.Z. and Bade, A. 2015. Implementation of axis-aligned bounding box for opengl 3D virtual environment. *ARPN Journal of Engineering and Applied Sciences*. **10**(2), pp.701-708.

Van der Vaart, E. and Verbrugge, R. 2008. Agent-based models for animal cognition: a proposal and prototype. In: *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 2, May 2008, Estoril, Portugal.* [Online]. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, pp.1145-1152. [Accessed 13 April 2023]. Available from: https://dl.acm.org/doi/10.5555/1402298.1402380.

Vlasov, R., Friese, K. and Wolter, F. 2012. Ray casting for collision detection in haptic rendering of volume data. In: *ACM Symposium on Interactive 3D Graphics and Games, 9 March 2012, Costa Mesa, California.* [Online]. New York: ACM. [Date accessed]. Available from: https://www.semanticscholar.org/paper/Ray-casting-for-collision-detection-in-haptic-of-Vlasov-Friese/4395a6b51cebccd7a5e9b73b87719f294059ba3c.

Wikipedia. 2005. Diamond-square algorithm. [Online]. [Accessed 14 April 2023]. Available from: https://en.wikipedia.org/wiki/Diamond-square_algorithm#:~:text=The%20diamond%2Dsquare%20algorithm%20is,which%20produces%20two%2Ddimensional%20landscapes.

Yang, HR., Kang, KK. and Kim, D. 2011. Acceleration of Massive Particle Data Visualization Based on GPU. In: Shumaker, R.: *Virtual and Mixed Reality - Systems and Applications. VMR 2011. Lecture Notes in Computer Science, vol 6774.* [Online]. Berlin, Heidelberg: Springer, pp.425-431. [Accessed 10 April 2023]. Available from: https://doi.org/10.1007/978-3-642-22024-1_47.

Appendix A
Self-appraisal

A.1 Critical self-evaluation

When I first planned the project, I was not sure how difficult it would be to complete it, since I lacked experience in graphics, having only learnt the basics in the first semester of this year. Therefore, the plan I made was mostly based on intuition, on what I thought I could achieve within each week. As I progressed through the project, I realised at certain times that some of the tasks were easier than I thought, and some were harder and needed more time. Overall, I feel that the project was not as complex as I would have wanted it to be, since the boids algorithm's implementation proved to be fairly straightforward, but at least I have learnt a lot from this project, and I am pleased with the final result.

I also wanted to work some more on the project before handing it in, so that I could improve it with different optimizations discussed in **Chapter 1**, but after getting some advice from my supervisor, I realised that I might not finish the project if I start working on extra features. Therefore, I made the decision to just get the report done and only afterwards to try to improve it, which proved to be the best choice, since writing almost 30 pages took longer than I thought it would and I finished everything close to the deadline.

A.2 Personal reflection and lessons learned

I really enjoyed working on this project, and it provided a level of challenge, being the first graphics-based one developed on my own, with experience only from a pair project I worked on in the first semester on the COMP3811 Computer Graphics module. It was a great opportunity to not only apply many of the concepts learnt throughout my course, but also to go through a full project development. Compared to that first OpenGL project, this one was much more structured and well planned, and it helped me better understand all the concepts related to OpenGL and graphics applications. I acquired a much better understanding of how the CPU and GPU communicate through buffers, what shaders are and how they work, and how to manage input and output through event handling.

Additionally, I learnt many techniques used in game development, that might prove very useful if I decide to pursue a job in that industry. I also learnt them in a much better way than I would have if someone thought me the concepts. For example, when trying to implement movement on a curve for the geometric flight part of the project, I did the research and documentation to find that vector interpolation was the answer. I had a problem to solve, and found the solution, instead of being taught about interpolations and about their applications.

Another thing I learnt is to do much more thorough research from the beginning. When I was researching the idea of boids and trying to form my project's objectives, I looked through many papers on the subject to try to find *a* way to implement it, but not necessarily *the best* way. I mostly followed the original paper, instead of trying to implement some of the newer improvements to the algorithm. I also had to implement concepts that are adjacent to boids, in collision avoidance for example, which I did not research from the very beginning, but only later in the process, before trying to implement it. Even though I did manage to finish everything I have planned, there is still room for improvement in every part of the project, and next time, I will remember to research more thoroughly, so that my decisions are well informed.

A.3 Legal, social, ethical and professional issues

A.3.1 Legal issues

One of the legal issues that can appear in a computer graphics project is that of copyrighted materials and assets. All the models and assets found online were the result of someone's hard work, which is why they should only be used with the author's explicit permission. Some of them are free to use, but require a reference, some are completely free, and others are found in marketplaces to be bought. To avoid this issue altogether, I opted for creating all of the models myself.

A.3.2 Social issues

An important social issue that appears in many computer graphics applications, especially video games, is that of access. Many of the new games released nowadays require heavy computation to render the detailed environment and player models in real time, along with all the interactions between them. This means that only people with access to the latest hardware can experience these applications. This issue applies to my simulation as well, since it was built and tested on a high-end gaming laptop, capable of running it and many other compute-intensive applications. Therefore, it likely performs worse on older hardware and can lead to unpleasant experiences for other users.

A.3.3 Ethical issues

One possible ethical issue that can be present in simulations replicating real-life behaviours is that it might not be accurate and thus could spread misconceptions related to the actual behaviours. In the case of this application, as discussed throughout this paper, there are limitations to the realism of the algorithm, which make the simulated behaviours appear natural, even if not accurate. This could lead to users basing their perception of fish only on this example, which would be inaccurate, thus leading to confusion or other problems.

A.3.4 Professional issues

In terms of professional issues, there is no apparent relevant issue since the project was developed by a sole developer. The project follows BCS code of conduct, since it acknowledges and references Third Parties, does not face any major ethical, social, or legal concerns and does not discriminate or cause harm in any way to anyone. It has been completed with professionalism, and has been described with sincerity and accuracy in this report.

Appendix B
External Materials

External materials used for the project include:

- Some of the code in the /math library was adapted from base code given to us in COMP3811 Computer graphics. The exact sections, mainly struct definitions and operators, have been clarified at the beginning of the files in the code repository.
- The texture used for the skybox is a free resource by sirsnowy7 available at: https://opengameart.org/content/ocean-hdriskybox.
- The heightmap used in the generation of the terrain was “A heightmap created with Terragen” from Wikipedia, available at: https://en.wikipedia.org/wiki/Heightmap.

External libraries used in the project (also references in the code repository with licences):

- Premake – for build configuration.
- GLFW – for creating windows and handling events.
- GLAD – OpenGL loader-generator.
- STB libraries – single-header libraries for image loading and writing.
- Rapidobj – single-header .OBJ file loader.
- ImGui 1.89.3 – UI library.