UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Erik Martin Vetemaa

Simulating the Collective Movement of Fish Schools

Bachelor's Thesis (9 ECTS)

Supervisor: Raimond-Hendrik Tunnel, MSc

Tartu 2020

Simulating the Collective Movement of Fish Schools

Abstract:

Fish schools are an example of emergent behaviour, where the complexity of the group's behaviour arises from the simple interaction of the individuals in the group. The emergent behaviour of fish schools is typically simulated using the Boids algorithm. The algorithm is based on three rules that the fish follow: separation, alignment and cohesion. In this thesis, the common method of implementing these rules is discussed and some improvements are proposed. Furthermore, additional rules to the aforementioned three are added to the algorithm. These rules are wander, predator avoidance and obstacle avoidance. The focus of this thesis is developing the algorithm for a visually interesting and lifelike fish schooling simulation. In addition to the algorithm, a demo application is built. The demo application visualizes the created algorithm on a fish school. At the end of the thesis, the visual results are analyzed and further improvements are proposed.

Keywords:

Computer graphics, Boids algorithm, behavioural animation, fish schools, distance field, gradient vector, obstacle avoidance, JavaScript, Three.js

CERCS:

P170: Computer science, numerical analysis, systems, control

P175: Informatics, systems theory

Kalaparvede liikumise simulatsioon

Lühikokkuvõte:

Kalaparved on näide emergentsest käitumisest, kus rühma käitumine põhineb rühma liikmete lihtsal koostoimel. Kalaparvede käitumist simuleeritakse tavaliselt kasutades Boidsi algoritmi. Algoritm põhineb kolmel reeglil, mida rühma liikmed järgivad: lahknemine, joondumine ja kogunemine. Käesolevas bakalaureusetöös antakse ülevaade tüüpilisest Boidsi algoritmi rakendamisest ja pakutakse sellele parandusi. Algoritmile lisatakse ka kolm uut reeglit. Need reeglid on uitamine, kiskjate vältimine ja takistuste vältimine. Töö eesmärk on luua algoritm visuaalselt huvitava ja loomutruu kalaparvede simulatsiooni jaoks. Algoritmile lisaks luuakse ka näidisrakendus, mille abil saab vaadelda

kalaparvede liikumist loodud algoritmi põhjal. Töö lõpus analüüsitakse visuaalseid tulemusi ja pakutakse edasiarenduse võimalusi.

Võtmesõnad:

Arvutigraafika, Boidsi algoritm, käitumuslik animatsioon, kalade parvestumine, kaugusväli, gradient vektor, takistuste vältimine, JavaScript, Three.js

CERCS:

P170: Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

P175: Informaatika, süsteemiteooria

Table of Contents

1 Introduction

The motion of a school of fish is a fascinating sight in nature. It is delightful to observe and an interesting topic of study. This kind of collective movement is also widespread among many other animal species and is described by the term *swarming*. Swarming behaviour has many known advantages over solitary behaviour, like avoiding predators, enhanced foraging and finding a mate [1]. The well-known V-formation of migrating flocks of birds is an example of aerodynamic gains of swarming [2]. There is increasing evidence of a similar, hydrodynamic benefit, to fish swimming in a school [3].

Due to the frequency of swarming behaviour in nature and its visual delight, it is commonly used in films and video games. Examples of this are the movement of a large number of rats or fast-moving robots in the widely popular Pixar movies Ratatouille (2007) and WALL-E (2008) [4]. Schools of fish have been an important part of recent video games like ABZÛ[1] (2016) and Subnautica[2] (2018) (Figure 1). Fish schooling is an integral part of these games and likely will be in many more video games in the future. However, the research in swarming behaviour is usually focused on bird flocks instead. Thus in this thesis, the focus of study is the movement of fish schools.

Figure 1. Left: ABZÛ. Right: Subnautica[3].

Pitcher and Parrish [1] describe the term *school* as a coordinated group of swimmers. A group of fish that stay together is described using the term *shoal*. Therefore, schooling is a type of behaviour that can occur in shoals. Schooling is common for thousands of fish

[1] https://abzugame.com

[2] https://unknownworlds.com/subnautica

[3] Image source: https://unknownworlds.com/subnautica/subnautica-below-zero/

species that spend at least a part of their lives in a school. While some species, like tuna and mackerel, spend most of their life in a school, other species do it only occasionally (e.g. for reproductive purposes) [5].

Different species of fish have quite different schooling behaviours. However, by making some generalizations the movement of most fish schools can simply be described by the fish keeping close to each other and moving in the same direction in a coordinated manner. Defining different types of movement is an important part of computer graphics and there are various methods for doing it. In computer graphics, the process of generating images of moving objects is called animation and one of the most common methods of animation is keyframe animation [6]. It is the process of setting values (e.g. position, rotation, scale) for specific points in time. Values with their corresponding points in time are interpolated between to create a smooth animation. In this thesis, the focus is on animating only the position of the fish. Keyframing would be suitable for animating the position of a single fish, but animating the positions of hundreds of schooling fish would require an overwhelming amount of manual work. Additionally, manually setting the position of all the fish so that the animation looks natural is very difficult. For this reason, it is much more appropriate to use an algorithm for animating the movement of fish schools.

The standard algorithm used for simulating swarming behaviour in computer graphics is the Boids algorithm created by Greg Reynolds in 1986 [7]. It was first used in an animated film in 1987's Breaking the Ice and in a feature film in 1992's Batman Returns [8]. The algorithm is based on three rules that each individual in the swarm obeys. The rules are as follows: avoiding collisions with nearby flockmates, matching the velocity of nearby flockmates and moving towards nearby flockmates [9]. A lot of research has been put into developing this algorithm further and adding custom rules to the aforementioned three. Petzold, Halle, and Thielecke [10] in 2004 added rules for avoiding obstacles and following a pre-set path. Sun and Tokunaga [7] in 2014 created a method for adding different external forces which could be steering towards food, staying away from obstacles or the flow of water in a certain direction.

In this thesis, the common implementation of the three rules in the Boids algorithm is improved upon and several additional rules are proposed. The focus of this thesis is developing the algorithm for a visually interesting and lifelike fish schooling simulation. Optimizing the algorithm for computer performance is mentioned in some parts of the thesis, but it was not the goal when developing the algorithm.

The programming language used while developing the algorithm and creating a demo application was JavaScript. It was chosen for the possibility of running the simulation on any device with a modern web browser and the ease of distributing the created demo application on the web. The tool used for rendering the simulation is WebGL[4], which is a JavaScript API for rendering 3D graphics within web browsers. It was chosen for its cross-platform royalty-free nature and because it is supported by all major browsers. Due to the low-level nature of WebGL, a high-level library Three.js[5] was used to speed up the development process. The code examples in this thesis are written in JavaScript and Three.js. Figure 2 illustrates a fish school simulation produced in this thesis.

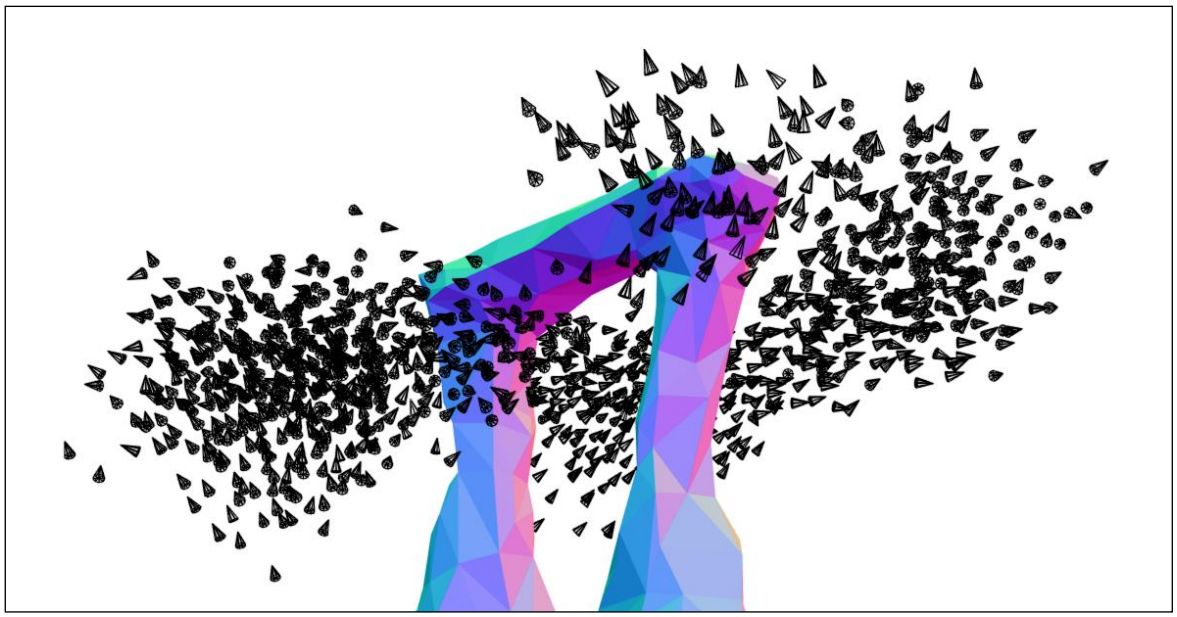

Figure 2. Screenshot of 1000 fish schooling around an obstacle.

This thesis is organized as follows. In Chapter 2, the Boids algorithm is described. In Chapter 3, the common implementation of the Boids algorithm is described and improved. In Chapter 4, additional rules are added to the algorithm. In Chapter 5, the results of the thesis and potential improvements are discussed. In Appendix I, the accompanying files are described. The files include the source code of the demo application and some videos of the simulation. In Appendix II, the structure of the source code is described. A live version of the demo application is accessible via the URL: http://boids.tint.digital/[6].

[4] https://www.khronos.org/webgl

[5] https://threejs.org

[6] Alternative link: https://vetemaa.github.io/fish-simulation/

2 Boids Model

The approach devised for simulating fish schooling in this thesis is based on the Reynolds' Boids algorithm. Boids is an artificial life algorithm and its main purpose is to simulate the flocking behaviour of birds. In the algorithm, Reynolds refers to each individual of the flock as a *boid* (bird-oid object) and therefore the algorithm is called Boids. In this thesis, the individual of the school is also referred to as a boid. A group of boids is in most research referred to as a flock. A flock, by definition, is strictly a group of birds, but to align with other research and Reynolds' original paper, a group of boids will be often referred to as a *flock* in this thesis as well.

The Boids algorithm is a great example of emergent behaviour. Emergence occurs where the complex behaviour of a system arises from the simple interactions of its parts [11]. The boids follow basic rules, but from their interaction, behaviour arises that is more complex than the behaviour of an individual boid. The rules that the individuals in the algorithm follow are expressed by vectors that steer their movement. In this thesis, these vectors are referred to as *steering vectors*. In the following sections, a brief overview of the rules is given based on the paper Reynolds published in 1987 [9].

2.1 Collision Avoidance

The first and most precedent rule of the Boids algorithm is collision avoidance, also called *separation* (Figure 3: left). Collision avoidance is the boid's desire to steer away from an impact and it is the most self-evident rule of the algorithm. The collision avoidance in the Boids algorithm only uses the present position of flockmates and does not predict their future movements. Therefore, separation is a non-predictive method of avoiding collisions.

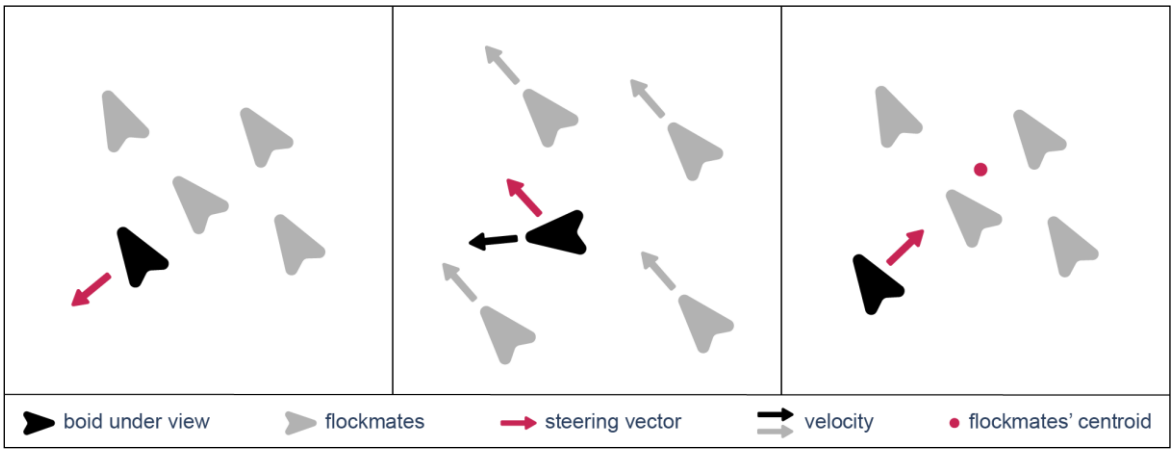

Figure 3. Left: separation. Centre: alignment. Right: cohesion.

For the flocking behaviour to look realistic, it is important to ensure that the members of the flock do not get too close to each other and that they do not collide. Completely avoiding collisions by using the separation rule also removes the need to implement an explicit collision detection algorithm. It would otherwise be necessary to stop the boids in the simulation from moving into and intersecting with each other.

2.2 Velocity Matching

The second rule of the Boids algorithm is velocity matching, which refers to each boid matching its speed and heading with its flockmates (Figure 3: centre). The velocity matching rule is also called *alignment* as it aligns the boid's direction with its flockmates.

Alignment ensures the characteristic synchronized movement that fish schools have. It also has a complementary effect on collision avoidance. If a boid matches the velocity of its flockmates, it is unlikely to collide with them. Alignment is, therefore, a predictive method of collision avoidance.

2.3 Flock Centring

The third and least precedent rule of the Boids algorithm is flock centring, which refers to each boid attempting to move towards the centroid[7] of its flockmates. Flock centring is also called *cohesion* (Figure 3: right). Cohesion is required to imitate the grouping nature of flocks.

While these rules are quite clear and widely accepted, the actual method of implementing them for a simulation is not as evident. In the following chapter, the most common method of implementation is described.

[7] https://www.merriam-webster.com/dictionary/centroid

3 The Boids Implementation

The implementation of the Boids algorithm used as the foundation of this thesis is the most common implementation which is described in computer science books like *Discovering Computer Science* [12], *Python Playground* [13] and *Think Complexity* [14]. There are some minor differences in them, but the fundamental method of computing the steering vectors is the same. In this chapter, the main structure of the algorithm, the implementation of the three Reynolds' rules and a commonly added boundary rule are explained. At the end of this chapter, some problems with the common implementation are identified and a method of improving it suggested.

3.1 General Structure

The purpose of the algorithm is to change the position of the boids over time. Specifically, the position of the boids should be changed every time the simulation is rendered[8] and therefore the algorithm is to run once per frame. During each run of the simulation and for every boid, the steering vectors based on the three Reynolds' rules are computed. These three steering vectors are added together to create an *acceleration* vector for a boid for the current frame. The acceleration vector is then added to the boid's *velocity* vector. In the aforementioned books, delta timing[9] is not discussed, but it certainly should be used. Delta timing diminishes the effect of frame rate on the outcome of the simulation and makes real-time animation look visually smoother. Therefore, the acceleration vector is multiplied with delta time before it is added to the velocity vector. Moving a boid to a new location in the scene is achieved by adding the boid's velocity vector to its *position* vector.

As the velocity vector is not reset for each frame, there is the danger of it growing infinitely with each run of the simulation. To avoid this, the length of the velocity vector is limited by an arbitrary constant `maxSpeed`. To adjust the influence (length) of each steering vector, arbitrary scalars `separationScalar`, `alignmentScalar` and `cohesionScalar` are used. The steering vectors are multiplied by the corresponding scalar before being added to the velocity vector. Code 1 illustrates the main structure of the implementation.

[8] http://digitalarchaeology.org.uk/the-science-of-3d-rendering

[9] https://medium.com/@dr3wc/understanding-delta-time-b53bf4781a03

```
boids.forEach((boid) => {
  separation = getSeparation(boid);
  alignment = getAlignment(boid);
  cohesion = getCohesion(boid);
  // additional rules

  acceleration = new THREE.Vector3();
  acceleration.add(separation.multiplyScalar(separationScalar));
  acceleration.add(alignment.multiplyScalar(alignmentScalar));
  acceleration.add(cohesion.multiplyScalar(cohesionScalar));
  // additional rules added to acceleration

  boid.velocity.add(acceleration.multiplyScalar(deltaTime));
  boid.velocity.capLength(0, maxSpeed);
  velocity = boid.velocity.clone();
  boid.position.add(velocity.multiplyScalar(deltaTime));
 });
```

Code 1. The main structure of the implementation.

Because the boid's velocity is altered using the `maxSpeed` constant in every frame, the velocity vector is also multiplied by delta time to maintain frame rate independence. In the following sections, the implementation of each of the steering vectors is described.

3.2 Separation

The separation vector steers the boid from collisions with its flockmates. The common method of creating the separation vector is finding a vector from the flockmates' centroid to the boid. If added to the acceleration vector, it will generally steer the boid away from any collisions with its flockmates. The centroid is found by adding together the position vectors of all flockmates and dividing the summed vector by the number of flockmates. To only include neighbouring flockmates, an arbitrary constant `separationRadius` (red circle in Figure 4) is used to ignore flockmates that are further from the boid than the value of `separationRadius`. This type of radius will be referred to as a *neighbourhood radius*. For comparing with the radius, the distances to flockmates have to be found. The distance to a flockmate is the length of the vector found by subtracting the boid's position vector from the flockmate's. When the centroid of neighbouring boids is found, it is subtracted from the boid's position to create the separation vector for the boid. The separation vector is illustrated in Figure 4 and the implementation of it can be seen in Code 2.

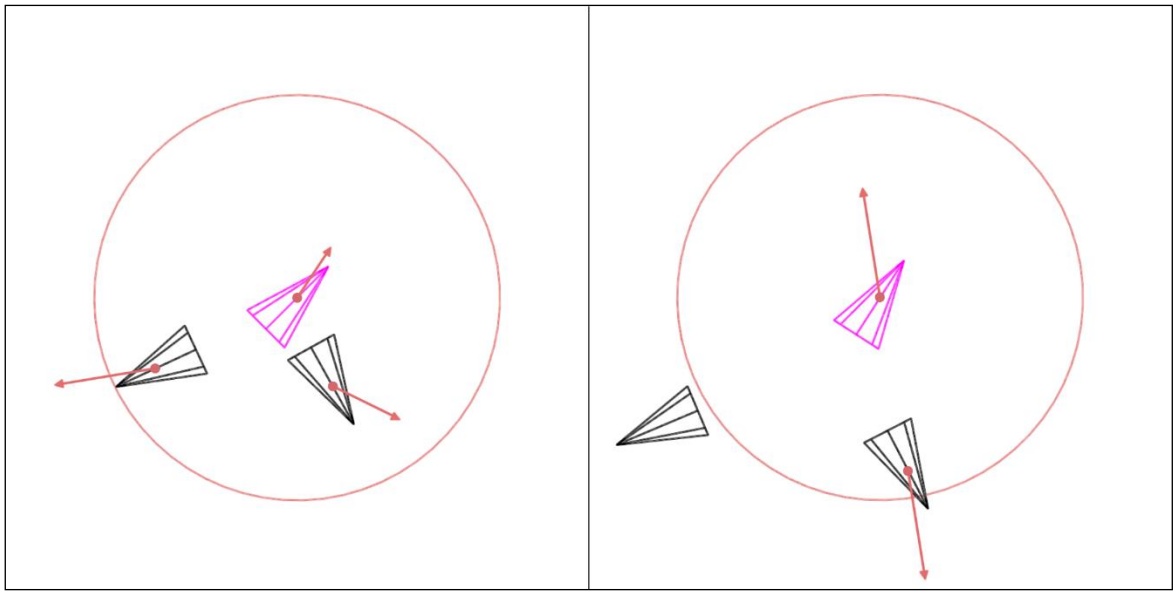

Figure 4. Screenshots illustrating the separation vectors with two flockmates (left) and one flockmate (right) in the neighbourhood radius of the purple boid. The lengths of vectors are scaled by an arbitrary value.

```
centroid = new THREE.Vector3();
neighbourCount = 0;
flockmates.forEach((flockmate) => {
  distance = boid.position.distanceTo(flockmate.position);
  if (distance < vars.separationRadius) {
    centroid.add(flockmate.position);
    neighbourCount++;
}});
if (neighbourCount > 0) {
  centroid.divideScalar(neighbourCount);
  separation = boid.position.clone().sub(centroid);
}
```

Code 2. Implementation of the separation vector.

The `separationRadius` is usually the smallest compared to the neighbourhood radiuses of alignment and cohesion. This allows the boids to get relatively close to each other, which is appropriate for swarming simulations. The weight (`separationScalar`) of the separation vector is usually the highest of the three Reynolds' rules to ensure that the other steering vectors do not cause boids to collide. This also results in the gap between neighbours usually being the same length as the value of `separationRadius.`

3.3 Alignment

The alignment vector steers the boid to match the velocity of its flockmates. The common method of creating it is finding the average of flockmates' velocity vectors. To find the average velocity vector, the velocity vectors of flockmates are summed and divided by the number of flockmates. To include only neighbouring flockmates, an arbitrary constant `aligmentRadius` is used as the neighbourhood radius. The alignment vector is illustrated in Figure 5 and the implementation of it can be seen in Code 3.

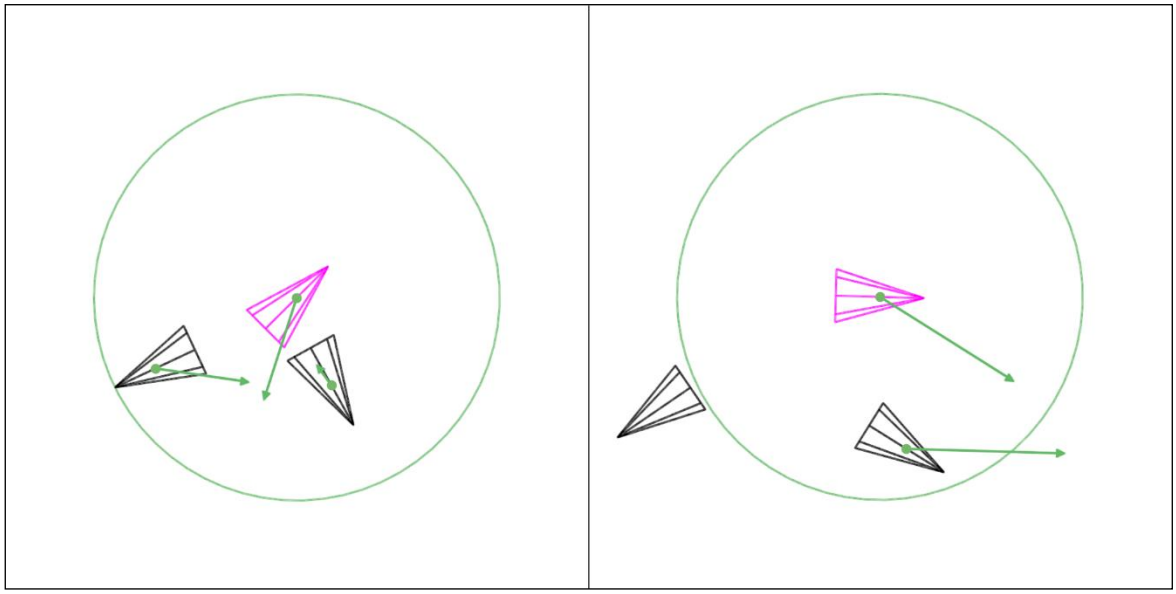

Figure 5. Screenshots illustrating the alignment vectors with two flockmates (left) and one flockmate (right) in the neighbourhood radius of the purple boid. The lengths of vectors are scaled by an arbitrary value.

```
alignment = new THREE.Vector3();
neighbourCount = 0;
flockmates.forEach((flockmate) => {
  distance = boid.position.distanceTo(flockmate.position);
  if (distance < vars.alignmentRadius) {
    alignment.add(flockmate.velocity);
    neighbourCount++;
}});
if (neighbourCount > 0) {
  alignment.divideScalar(neighbourCount);
}
```

Code 3. Implementation of the alignment vector.

Choosing the correct value for `alignmentRadius` is not as straightforward as for `separationRadius`. The effect of alignment spreads from boid to boid even if they are not in the neighbourhood radiuses of each other if they have a common neighbour. Their velocities will equalize through the common neighbour's velocity, but the effect is slower than if they were neighbours themselves. Therefore, the `alignmentRadius` can be adjusted to alter the alignment behaviour of the flock. Separation and alignment together ensure that the boids will not collide, but to gather the boids into a group, the cohesion rule is necessary.

3.4 Cohesion

The cohesion vector steers the boid towards its flockmates. The common method of creating it is finding a vector from the boid to the centroid of its flockmates. The centroid is found as described in Section 3.1 and the cohesion vector is the vector found by subtracting the boid's position from the centroid. To include only the neighbouring flockmates, an arbitrary constant `cohesionRadius` is used as the neighbourhood radius. The cohesion vector is illustrated in Figure 6 and the implementation of it can be seen in Code 4.

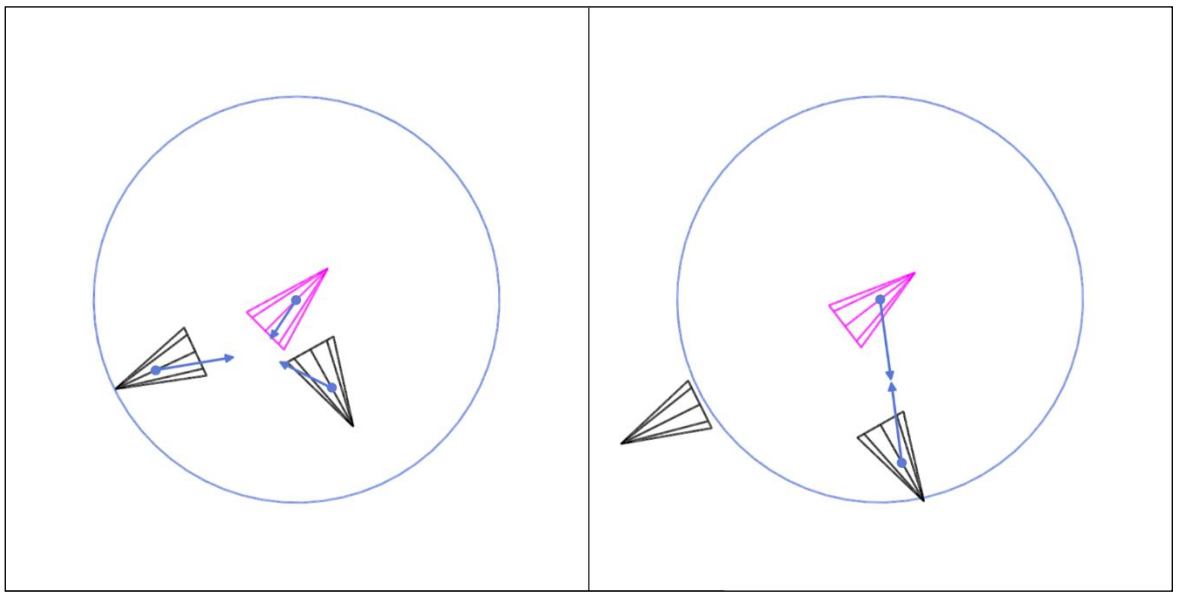

Figure 6. Screenshots illustrating the cohesion vectors with two flockmates (left) and one flockmate (right) in the neighbourhood radius of the purple boid. The lengths of vectors are scaled by an arbitrary value.

```
centroid = new THREE.Vector3();
neighbourCount = 0;
flockmates.forEach((flockmate) => {
  distance = boid.position.distanceTo(flockmate.position);
  if (distance < cohesionRadius) {
    centroid.add(flockmate.position);
    neighbourCount++;
}});
if (neighbourCount > 0) {
  centroid.divideScalar(neighbourCount);
  cohesion = centroid.sub(boid.position);
}
```

Code 4. Implementation of the cohesion vector.

Choosing the correct value for `cohesionRadius` is similar to that of `alignmentRadius`. To steer a boid towards a group of boids, the whole group does not have to be inside the neighbourhood radius. Steering towards some boids on the edge of the group is sufficient to make a boid part of the group. The radius can be used to alter the speed and behaviour in which the boids gather.

Using this implementation of the three Reynolds' rules with appropriate radiuses and scalars is enough to witness the emergence of schooling behaviour. In the following section, a rule is added to make observing the simulation of schooling behaviour easier.

3.5 Bounds

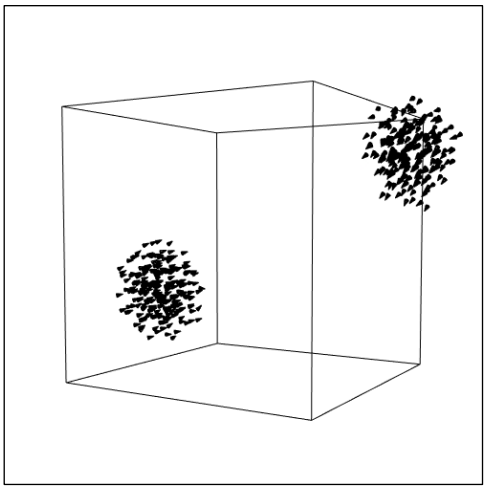

Figure 7. Cubic bounds.

In two of the three books mentioned at the beginning of Chapter 3, a rule was added to keep the boids from moving away from the visible area of the simulation. This is not a formal part of the Boids algorithm but is often added to demonstrate the algorithm better. In the book *Discovering Computer Science*, the method used for a 2D simulation was tiling the view in a way that when a boid crosses the right border of the visible area, it is immediately positioned to the left border of the visible area and so on. A method proposed for 3D in the book *Python Playground* was constantly steering the boids towards the centre of

the scene. However, constantly steering the boids in a direction might change the behaviour of the school in a significant manner. Thus, a different method is used in this thesis.

To keep the boids in a fixed area in 3D, a *bounds* rule is added. This area is defined by a cube (Figure 7) and boids that leave its boundaries are steered back towards it. The bounds vector is created by finding a vector from the boid to the cube.

The bounds vector towards the cube is found by calculating each axis of the vector separately. The x-coordinate of the bounds vector is found using the x-coordinates of the two axis-aligned planes e_{x1} and e_{x2} of the cube perpendicular to the x-axis using the formula

$$b_x = \begin{cases} e_{x1} - p_x, & p_x < e_{x1} \\ e_{x2} - p_x, & p_x > e_{x2} \end{cases},$$

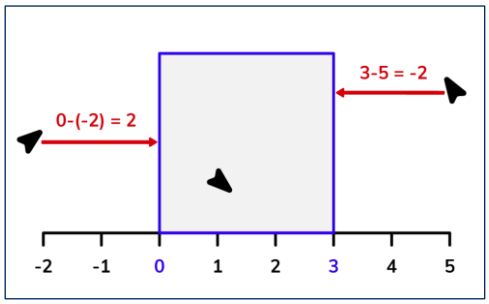

Figure 8. Calculating the bounds vector for a single axis.

where b_x is the x-coordinate of the bounds vector and p_x is the x-coordinate of the boid's position (Figure 8). The same process is done for each axis to create the bounds vector.

These four basic rules with custom scalars and neighbourhood radiuses are sufficient to create a simulation of fish schooling illustrated in Figure 7. However, the simulation in its current form looks quite unnatural. The reason for this is described in the following section and an improved implementation of the Reynolds' rules is given.

3.6 Improved Boids Implementation

Using the centroid of flockmates or the mean of their velocities when finding the steering vectors causes multiple issues and results in an unnatural simulation. In this section, separation is used as the primary example for presenting the current issues and a better implementation is proposed.

A problem when using the centroid to find the separation vector is that the distance between an individual flockmate and the boid does not have a significant effect on the steering vector. The left image of Figure 9 illustrates a situation where the steering vector from the centroid actually steers the boid towards the closest flockmate.

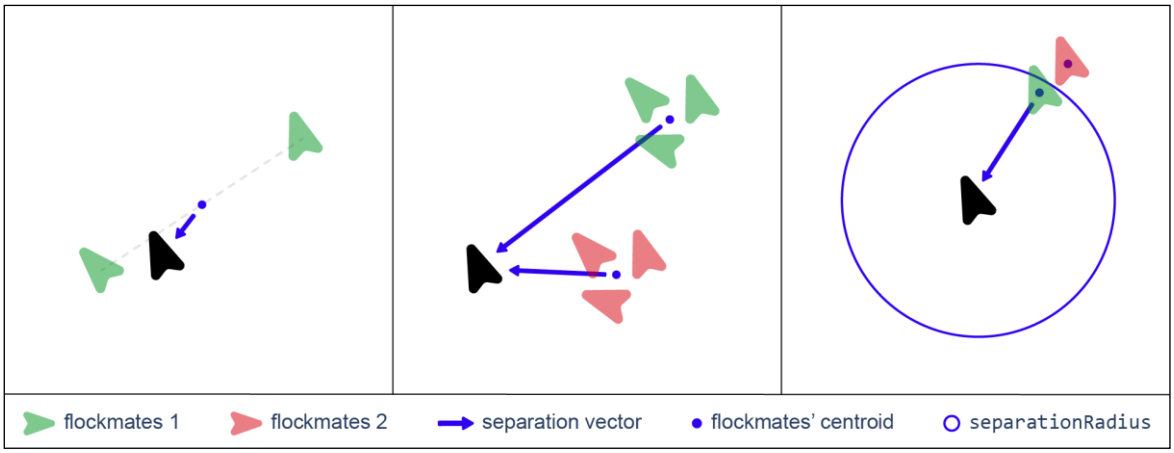

Figure 9. Issues that appear when calculating the separation vector. Left: steering towards a close flockmate. Centre: the lengths of steering vectors for flockmates in different positions. Right: separation vector when a flockmate is near the neighbourhood radius.

Additionally, the length and therefore the strength of the current separation vector is proportional to the distance between the boid and the centroid (Figure 8: centre). Thus, the simulated fish steers away from flockmates faster if they are far, compared to when they are close. This is not the desired behaviour for avoiding collisions.

The third main problem that arises is that when boids enter or leave the neighbourhood radius, the steering vector changes very suddenly. This can cause rapid changes in the velocity of the boid and results in stiff and unnatural movement for the fish. The sudden change in the steering vector is very evident when a single flockmate is inside `separationRadius` in one frame of the simulation and has left the radius in the next frame. This is illustrated by the green and red boids in the right image of Figure 9. The steering vector is very long when a flockmate is in the position of the green boid, but when that flockmate moves to the position of the red boid, the length of the steering vector is 0. The length of the steering vector in a situation like this is also illustrated by the sudden change in the orange line in Figure 10.

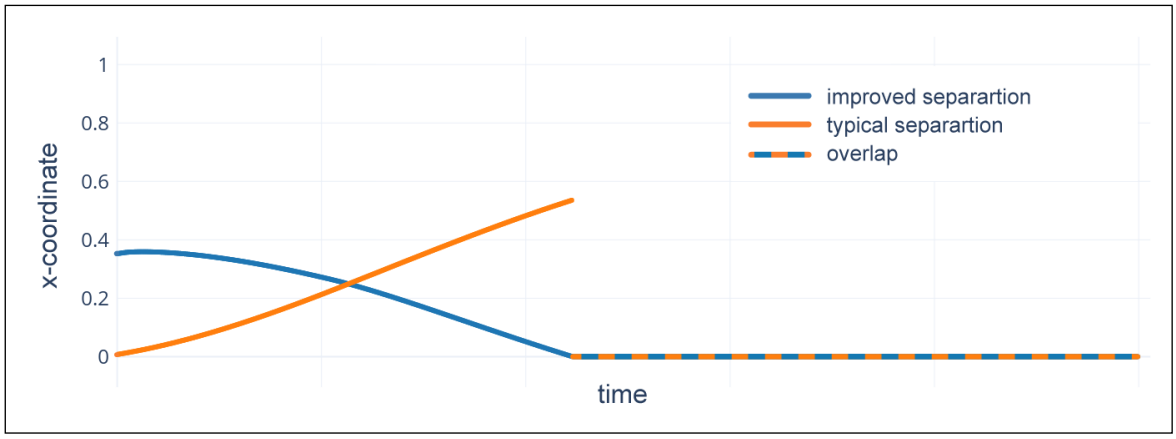

Figure 10. Examples of the x-coordinate of the original and improved separation vectors of a single boid over time using two boids in the simulation.

The undesired, sudden changes in the boid's velocity can be described with the mathematical term *smoothness*[10] used to measure the smoothness of functions. A function is smooth when it has continuous derivatives up to some order. The number of continuous derivatives determines the differentiability class[11] of the function. A function is of differentiability class C^k if it has continuous derivatives $f', f'' \dots f^k$. A function that belongs in the class C^k is more simply called C^k smooth.

The sudden changes in velocity are caused by the velocity function only being C^0 smooth. This means that the derivative of velocity, which is acceleration, is not continuous (has abrupt changes like the orange line in Figure 10). As acceleration is the sum of the steering vectors, a steering vector not being continuous directly causes the acceleration vector to be not continuous. To make velocity C^1 smooth, the steering vectors used to find acceleration have to be continuous (C^0 smooth). The orange line in Figure 10 illustrates the separation vector over time and it is clear that the function used to find separation is not C^0 smooth.

To avoid the described issues and to create a C^0 smooth separation vector, a different way to compute separation is required. Instead of finding the vector from the centroid of flockmates, vectors from each flockmate to the boid are used. A vector from a flockmate's position to the boid's position is referred to as the *difference* vector (red vectors in Figure 11: left). The sum of the difference vectors to each flockmate would yield a separation vector with an identical direction to the centroid method. To make the effect of close flockmates

[10] https://mathworld.wolfram.com/SmoothFunction.html

[11] https://www.encyclopediaofmath.org/index.php/Class_of_differentiability

stronger than the effect of further ones, the lengths of the difference vectors are set using linear decay by the formula

$$l_1 = \frac{r - l_0}{r} = 1 - \frac{l_0}{r} \ ,$$

where l_1 is the new length of the difference vector, r is the neighbourhood radius and l_0 is the prior length of the difference vector. By using the formula, the difference vectors length and the distance to the flockmate are no longer proportional ($l_0 \propto distance$), but inversely proportional ($l_1 \propto 1/distance$). An important aspect to note when using this formula is that the slope of decay is dependent on the neighbourhood radius.

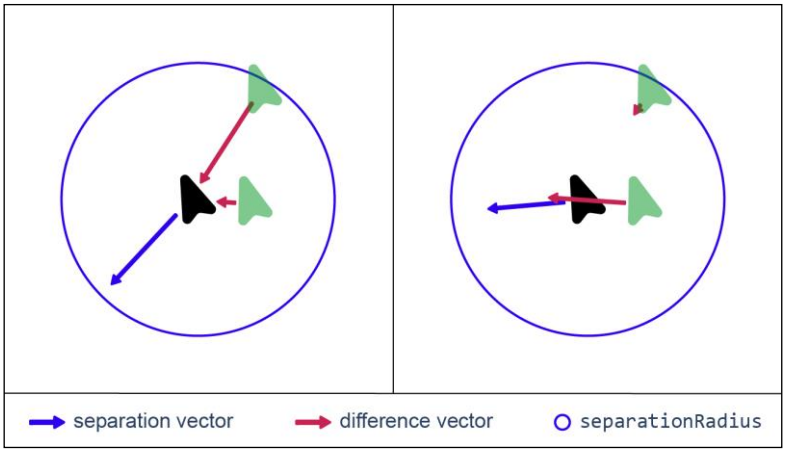

Figure 11. Finding the separation vector. Left: using the regular difference vector. Right: using difference vectors with linear decay.

By using the formula above to set the length of the difference vector, the effect of the difference vector corresponds to how close the flockmate is. Figure 11 illustrates the difference vectors before and after using linear decay. Figure 10 illustrates the smoothness of the separation function before and after.

When computing alignment, the effect of a flockmate's velocity vector should also decrease with distance. To achieve this, the flockmate's velocity vector's length is set to be inversely proportional to the difference vector's length using the linear decay formula mentioned above. It should be mentioned that the flockmates velocity vector, the length of which is being changed, is not the flockmate's velocity itself, but an instance of it used for finding alignment. Finding the improved cohesion vector is the same as separation, but instead of the difference vector being from the flockmate to the boid, it is from the boid to the flockmate.

To avoid the lengths of the steering vectors from being able to grow unconditionally with the number of flockmates in the neighbourhood radius, the lengths of the steering vectors are limited to an arbitrary value (1). This ensures that the length of a steering vector does not get too long, but still keeps steering C^0 smooth. The improvements made to the steering vectors can be seen in Code 5.

```
// (...)
flockmates.forEach((flockmate) => {
  distance = boid.position.distanceTo(flockmate.position);
  if (distance < separationRadius) {
    difference = boid.position.clone().sub(flockmate.position);
    difference.setLength(1 - distance / vars.separationRadius);
    separation.add(difference);
}});
separation.clampLength(0, 1);

// (...)
flockmates.forEach((flockmate) => {
  distance = boid.position.distanceTo(flockmate.position);
  if (distance < alignmentRadius) {
    velocity = flockmate.velocity.clone();
    velocity.setLength(1 - distance / vars.alignmentRadius);
    alignment.add(velocity);
}});
alignment.clampLength(0, 1);

// (...)
flockmates.forEach((flockmate) => {
  if (distance < cohesionRadius) {
    difference = flockmate.position.clone().sub(boid.position);
    difference.setLength(1 - distance / vars.cohesionRadius);
    cohesion.add(position);
}});
cohesion.clampLength(0, 1);
```

Code 5. Improved implementation of the steering vectors. Variable declarations are omitted from this code and are indicated by (...). The variable definitions can be seen in Code 2-4.

The algorithm implemented in this chapter yields a highly adjustable and lifelike fish schooling simulation. However, the algorithm can be developed even further by adding additional rules. These rules are discussed in the following chapter.

4 Additional rules

Due to the simple structure of the algorithm, additional rules and their corresponding steering vectors are easy to add. A new steering vector can be added to the acceleration vector without making any changes to the previous steering vectors. In this chapter, rules for randomness, predator avoidance and obstacle avoidance are proposed.

4.1 Random direction

A very impactful rule for adding realism to the Boids algorithm is the random direction rule, also referred to as *wander*. The goal of this rule is to add randomness to the movement of boids. Without it, schools that do not face any obstacles tend to move quite linearly forever. This might be the desired behaviour in some cases but tends to be visually uninteresting. A lone boid, without flockmates or predators, in an open space would never change its rotation or speed, which is very unrealistic (Figure 12: left). A large open space can be a common occurrence in simulations where fish swim in a large body of water. Thus, a wander rule should be added to introduce randomness to the boids movement.

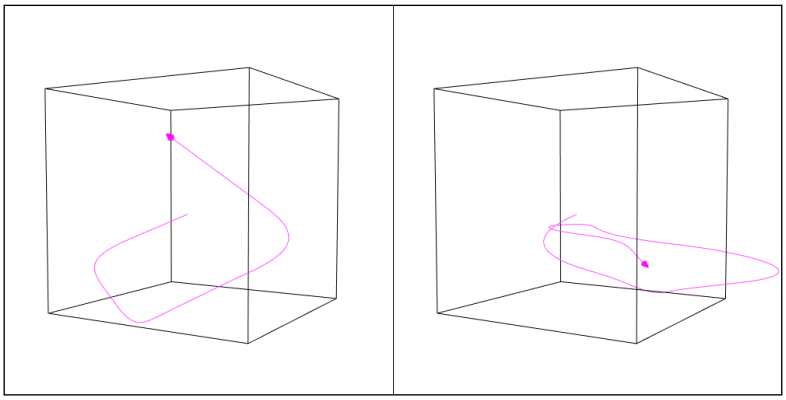

Figure 12. The movement of a single boid. Left: without wander (boid only changes its rotation because of the bounds). Right: with wander.

In order to continuously steer a boid in random directions, a random vector with a changed direction is required each frame. A simple approach to creating the vector would be generating random numbers for each axis of the steering vector and repeating the process for every frame. However, due to the vector being in an entirely new random direction each frame, the steering vector would not be C^0 smooth and the steering function would be dependent on the frame rate of the simulation. Therefore, a continuous and frame rate independent approach is used instead.

Stephens, Pham and Wardhan [15] implemented wander by using a single steering vector that is incremented by a short random vector at each time step. However, this steering vector is not C^0 smooth. If the increments were small enough, the steering vector would appear continuous, but then the sum of the increments would quickly converge into zero due to the uniformity of random numbers and the steering vector would not change enough over time to have a meaningful effect. Thus, a different method for finding wander, using a noise function, is proposed in this thesis.

The usage of noise functions is widespread in computer graphics ever since the algorithm for Perlin noise was presented by Ken Perlin in 1983 (Figure 13) [16]. There are multiple other well-established noise functions as well, but they are all mostly used for 2D texture generation, while only 1D noise will be used for wander. 2-dimensional noise means that two input values are used to generate the noise value. For the random vector, time is the only input so just 1D noise is required. Three 1D noise functions will be used to generate each coordinate of the steering vector. Most of the well-established methods could be used for generating the noise, but a much simpler method is sufficient for wander.

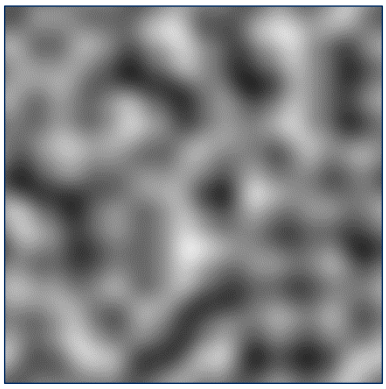

Figure 13. Perlin noise in 2D.

To create a basic noise function, a random number generator and an interpolation function are required. For this purpose, most pseudo-random number generators could be used, because common issues with random number generators like the lack of uniformity or a short period length will not be noticeable in the simulation. The random number generator is used to find values between -1 and 1 (Figure 14: left). To create a continuous function out of the random values, an interpolation function is used to create line segments between the random values. The simplest option is using linear interpolation (Figure 14: centre). The interpolated function is C^0 smooth and when used to find wander it would yield a C^0 steering vector.

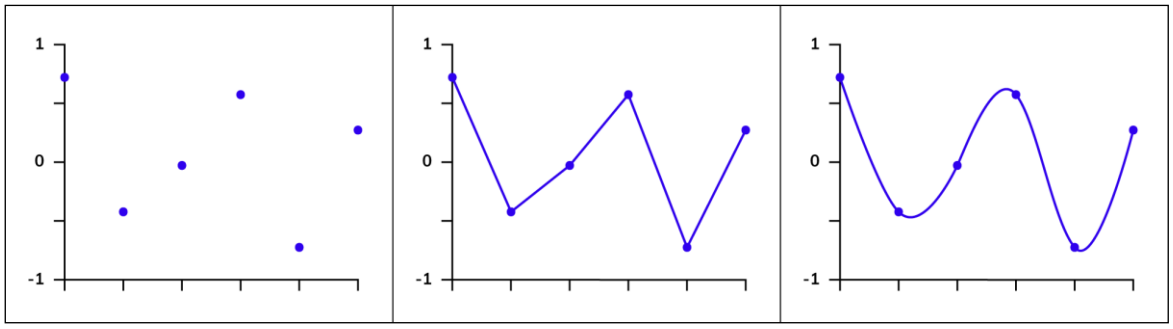

Figure 14. Left: random values between -1 and 1. Centre: linear interpolation. Right: cubic interpolation.

However, because wander is the only steering vector for a lone boid inside the bounds, the sudden changes near the random values of the noise function are still noticeable in the boid's velocity, even though the noise function is C^0 smooth. Therefore, the noise function is smoothed further using cubic interpolation, which is the simplest interpolation method that offers true continuity between the line segments[12]. Because cubic interpolation is very common in computer graphics, it is not described in this thesis but can be read about in the article by Paul Breeuwsma[13]. The noise function with cubic interpolation is illustrated in the right image of Figure 14.

Using time as the single input, a frame-rate independent random value can be found for each frame. A random vector is created by using a separate noise function for each of the three axes of the steering vector. Thus, a random steering vector that changes smoothly over time is created. Because fish tend to move less vertically than horizontally, the vertical steering is reduced by multiplying the vertical axis coordinate of the wander vector by an arbitrary value less than 1. Figure 12 illustrates the movement of a single boid before and after wander is added.

[12] http://paulbourke.net/miscellaneous/interpolation/

[13] https://www.paulinternet.nl/?page=bicubic

4.2 Predator avoidance

For simulations that have both prey and predator fish, a predator avoidance rule is necessary so that the prey fish flee from the predators. Figure 15 illustrates predator avoidance in use.

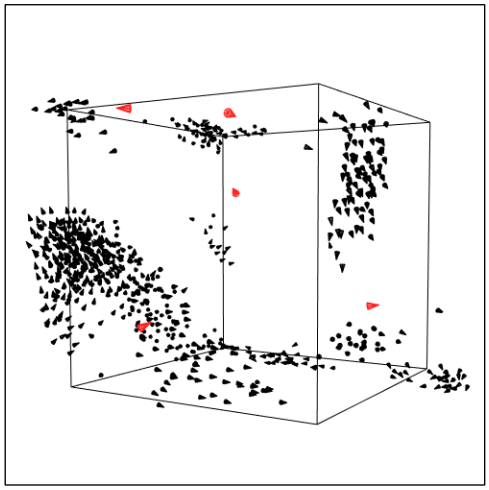

Figure 15. Boids avoiding predators.

Creating the steering vector for predator avoidance is very similar to the creation of the separation vector described in Section 3.6. The main disparity is that the difference vectors are found using the positions of predators, instead of the positions of flockmates. Additionally, a larger neighbourhood radius than `separationRadius` is used (`fleeRadius`). The implementation of the predators themselves is not discussed in this thesis but can be seen in the source code of the demo application (Appendix II).

4.3 Obstacle avoidance

In animations of fish schools, it is likely that an underwater landscape with rocks, corals or other objects is included in the scene. Therefore, the simulated fish need a method of avoiding those obstacles so that they would not swim through them. For this reason, an *obstacle avoidance* rule has to be implemented. The steering vector created using this rule should steer the boid away from any close obstacles while still allowing the boids to move freely around them.

Some work has already been put into avoiding obstacles in swarming simulations. Sun and Tokunaga [7] used a method of avoiding collisions by placing force objects (spherical areas) around the obstacle that steer the boid in some direction. Petzold, Halle, and Thielecke [10] approximated the obstacles with spheres and used those to find an avoidance vector. Neither of these methods would work well for usage in films and video games that often have hundreds of obstacles with complex shapes in the scene. Placing force objects around all those would be a lot of work and could lead to too many force objects that need to be iterated through, which would cause performance issues. Approximating complex objects with spheres is not a good option due to its imprecision. Therefore, a different method from these is proposed in this section.

The common method of defining an object in computer graphics is by using a polygon mesh. A polygon mesh consists of vertices, edges and faces that collectively represent the shape of an object in 3D [17]. The simplest type of polygon mesh is a triangle mesh. All the faces of a triangle mesh are triangles that consist of three vertices. A Polygon mesh with more than three vertices in a face can be triangulated to a triangle mesh and triangulation algorithms are widely used for that purpose [18]. Because of the simplicity of doing computations on a triangle compared to higher-order polygons, triangle meshes are taken as the basis for obstacle avoidance.

The trivial approach to steering boids away from an obstacle would be to find the closest point on the obstacle's mesh, creating a vector from that point to the boid and using that as the steering vector. Finding the closest point on a mesh to the boid is a complex task and is described in the following subsection.

4.3.1 Closest Point on a Mesh to a Boid

The closest point on a mesh can be found by comparing the closest point on each of the faces of the mesh and choosing the closest point among them. Finding the closest point on a face to a point in space is a common problem in computer graphics and the method for finding the closest point on a face used in this thesis is the algorithm described in the book *Real-Time Collision Detection* [19].

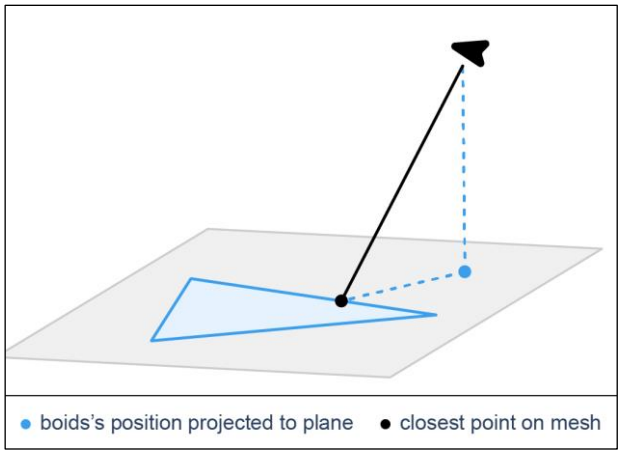

Figure 16. Closest point on a mesh to a boid.

First, the boid's position is projected onto the plane defined by the three vertices of the face (Figure 16). Thus, the face and the boid's position are on the same plane and the problem is reduced to finding the closest point on a triangle to a point in 2D. The method for finding the closest point on a triangle uses Voronoi regions and barycentric coordinates. As the method is quite complex it is not described in this thesis but is thoroughly discussed in the book *Real-Time Collision Detection* [19]. Using this method the closest point on each face can be found. By comparing the lengths of the vectors from the boid to the points on the faces and choosing the shortest, the closest point on the mesh to the boid is found. A vector

from some point p to the closest point on a mesh to that point p is referred to as the *closest point vector* of point p. The usage of this vector as a steering vector is described in the following sub-section.

On strictly convex meshes, computationally faster methods like the GJK-algorithm [19] can be implemented instead, but are not used in this thesis to allow concave meshes as well. A different approach to avoid performance issues is described in Subsection 4.3.3.

4.3.2 Basic Steering Vector

A steering vector for obstacle avoidance can be created by finding the boid's closest point vector and using that as the steering vector. With multiple meshes in the scene, the closest point vector for each mesh would be found and the shortest vector used for steering. To stop boids from avoiding meshes that are too far, a variable `avoidanceRadius` is used to ignore meshes with the closest point vector longer than the specified radius.

To make the steering vector's length and therefore its effect smaller as the boid gets further from close meshes, linear decay is used to set the steering vector's length. The formula for linear decay was described in Section 3.6.

This method for finding a steering vector already works quite well to avoid obstacles but is very slow. Figure 17 illustrates the time it takes to find a single closest point vector relative to the number of faces in the scene. In video games, a frame rate of 60 frames per second is considered as a good standard. This means that a single frame should not take longer than 16 milliseconds (ms) to render. However, from the graph in Figure 17, it can be deduced that finding the closest point vector for a single boid in a scene with more than 20 thousand faces already takes more than 16ms.

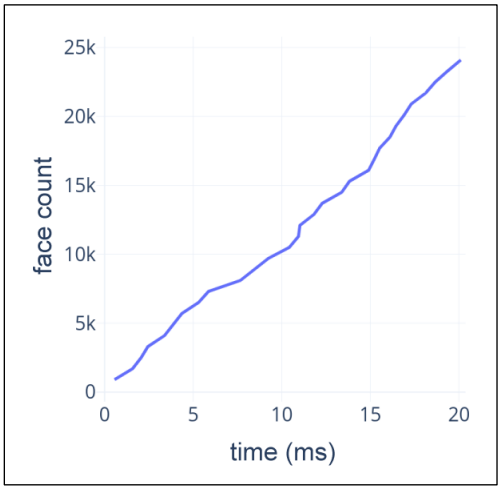

Figure 17. Closest point vector calculation time.

The computation time for the closest point vector was measured using an AMD Ryzen 5 1500X processor. It should be noted that better results can be achieved when using a GPU for computing the closest point vector, but as scenes in modern films and video games often have millions of faces, this method for obstacle avoidance is still too slow in practice. Thus,

in the next subsection a faster method, that uses a vector field with precomputed steering vectors, is proposed.

4.3.3 Vector Field

A vector field[14] is a function that assigns a vector to each position in a subset of space. The vector fields created in this thesis are based on a 3D array that holds vectors from which the output of the field is created. The vectors inside the array correspond to specific positions in space (Figure 18, Figure 19: left). The value of the vector field at any of those positions is the corresponding vector. To get the value of some position inside the vector field that does not correspond to a specific vector, the vectors that surround that position are interpolated via trilinear interpolation[15] (Figure 18). Thus, a vector field is created that outputs a steering vector for any input coordinates that the field covers.

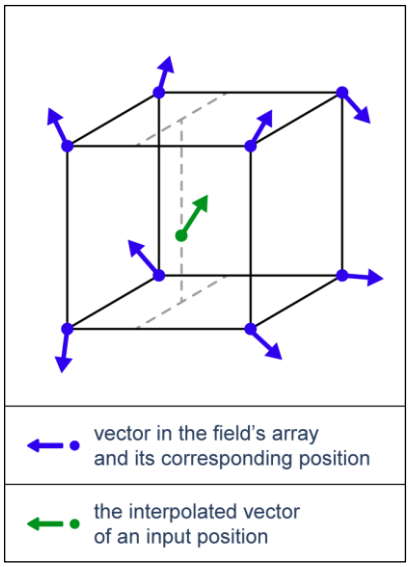

Figure 18. Four neighbouring vectors in the field's array and their interpolation.

A vector in the field's array associated with a position in space will be referred to as a *point* in the field. The fields in this thesis will cover a cubical area of space and the *length* of the field will refer to the length of the area's edge and the *resolution* of the field to the number of vectors in the field array for each axis. Therefore, a 3D vector field with a resolution of 3 and a length of 30 would consist of $3x3x3 = 27$ vectors and the gap between points in the field would be $30/3 = 10$ units.

For creating the vector field, vectors corresponding to each of the field's points are required. The vectors are found using the method of finding a steering vector (using the closest point vector) which was described in the previous subsection. But instead of using the boid's position, the field point's position is used for creating the vector. The field points' positions are computed using the pre-set resolution and length of the field. The left image of Figure 19 illustrates the vectors in the field's array in their corresponding positions in space. The

[14] https://www.whitman.edu/mathematics/calculus_online/section16.01.html

[15] https://en.wikipedia.org/wiki/Trilinear_interpolation

right image of Figure 19 illustrates the lengths of the field's output vectors for input positions on a plane.

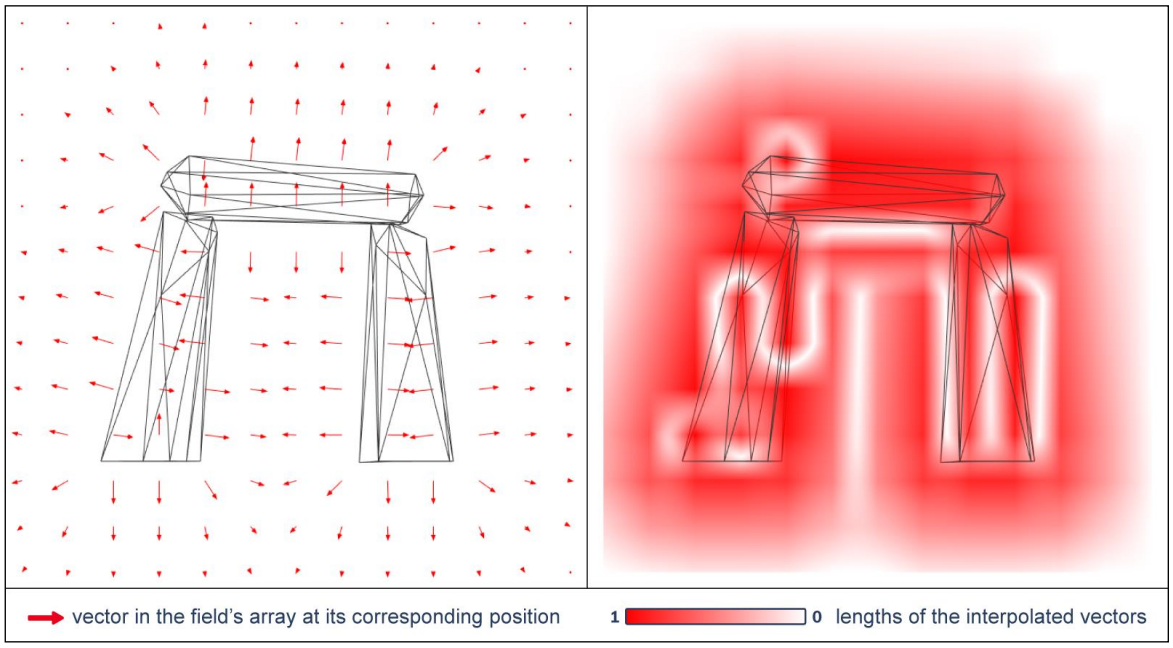

Figure 19. 2D slices of a vector field with a mesh inside it. Left: vectors in the field's array. Right: the length of the interpolated vectors for input positions on a plane.

In the right image of Figure 19, an odd pattern emerges. This pattern is caused by the vectors in the field's array that are positioned inside the mesh being in the opposite direction to adjacent vectors outside the mesh. When interpolating these vectors, they cancel each other out and the interpolated vector's length is very short, which causes the white areas in Figure 20. The interpolated vector in a situation like this is also illustrated by the red dot in the left image of Figure 20. The interpolated red steering vector's length is 0 and the boid is not steered away from the mesh.

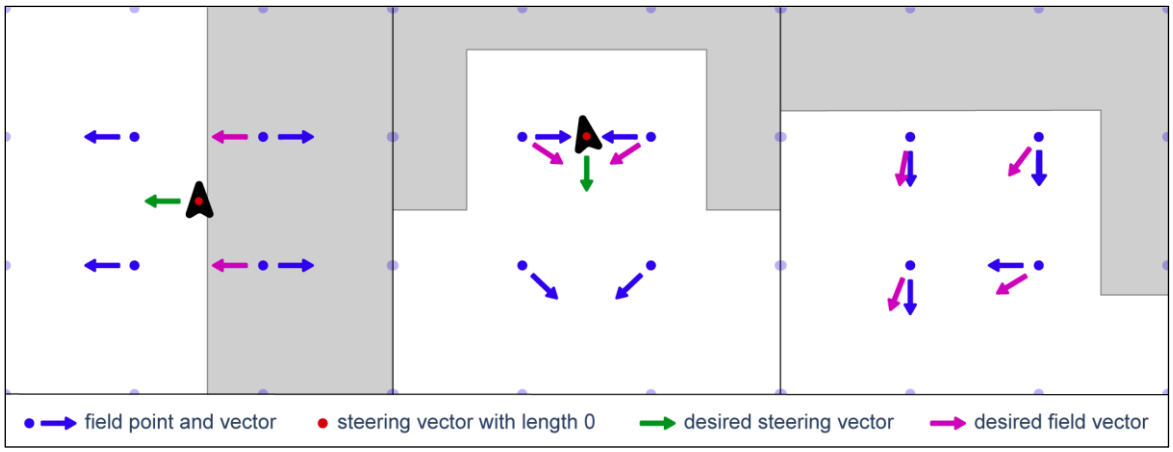

Figure 20. Problems with the direction of vectors in the vector field.

Furthermore, other issues arise when using the vector field to steer the boid. The centre image of Figure 20 illustrates a situation in which the field's vectors found from the closest point vector do not steer the boid away from the mesh. The purple vectors in Figure 20 show what the vectors in the field should look like to steer the boid away as opposed to the blue vectors that are found by the current method. Because of these problems, a different method for creating the vectors in the field is proposed in the following subsection.

4.3.4 Avoidance Vector

With the current method, a simulated fish essentially finds the closest point on the closest obstacle and steers exactly in the opposite direction of that. This behaviour seems unnatural for a fish. Steering in the direction that leads the fish to swim away from close obstacles the fastest seems much better. In other words, a fish should swim in the direction in which distance to the closest mesh increases the fastest. To find the direction in which the distance increases the fastest, a field will be used that outputs the distance to the closest mesh of some input coordinates.

A field that represents the distance to meshes or some other object is widely used in computer graphics and is referred to as the *distance field*[16]. The distance field is very similar to a vector field, but instead of a vector, it outputs a scalar value (a number). Therefore, the distance field is a type of *scalar field*[17].

The distance field is created as the vector field described in Subsection 4.3.3, but instead of filling the field's array with vectors, the array is filled with the distances to the closest mesh. These distances are the lengths of the closest point vectors for each of the points in the distance field. This distance field outputs the interpolated distance to the closest mesh of some input coordinates (Figure 21). Next, the direction in which the distance field values for some input coordinates increase the fastest has to be found.

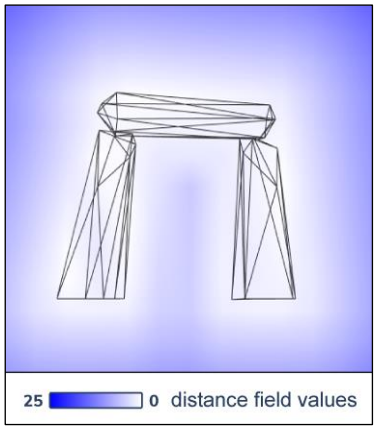

Figure 21. 2D slice of the distance field. The length of the field is 40 units.

[16] https://prideout.net/blog/distance_fields/

[17] https://en.wikipedia.org/wiki/Scalar_field

In calculus, a vector that points in the direction of greatest increase of a multivariable function f at some point p is called the *gradient*[18] and is notated as $\nabla f(p)$. The gradient vector at a point p is a vector whose coordinates are the partial derivatives of the function f at p and is found by the formula $\nabla f(p) = \left[\frac{\partial f}{\partial x_1}(p) \quad \cdots \quad \frac{\partial f}{\partial x_n}(p)\right]$.

As the gradient vector of a function shows the direction and rate of increase of a function, it can be used for obstacle avoidance by finding the gradient of the distance field. Because the distance field is not an analytical function that the partial derivatives can be found for, the derivatives are approximated using a finite difference method[19]. Finite difference methods are a numerical way of solving differential equations and are widely used in computer science.

A commonly used finite difference method states that the difference between the values of two neighbouring points of point p for some function f can be used as the approximate derivative of f at point p. These neighbouring points are illustrated by $p + \Delta p$ and $p - \Delta p$ in Figure 22. This method for finding the derivative of $f(p)$ is called the central difference approximation[20] and is found by the formula $'(p) \approx \frac{f(p+\Delta p) - f(p-\Delta p)}{2\Delta p}$.

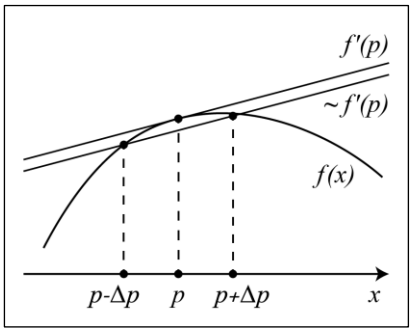

Figure 22. Apprximating the gradient of a function.

To find an approximate gradient of the distance field, the central difference method is used to find the partial derivative in all three axes. Because the distance field is a regular grid and Δp between neighbouring discrete points is always 1, the difference between points in the field can be replaced by the constant 1. Thus, the approximate gradient for the distance field f for the coordinates x, y, z is found by the formula

$\nabla f(x, y, z) \approx \left[\frac{f(x+1) - f(x-1)}{2} \quad \frac{f(y+1) - f(y-1)}{2} \quad \frac{f(z+1) - f(z-1)}{2}\right]$.

On the boundary of the distance field, some of the neighbouring points used for finding the gradient vector will be outside of the field. Thus, a *boundary condition* is required for those situations. In this thesis, the boundary condition will be that the gradient vector at the

[18] https://en.wikipedia.org/wiki/Gradient

[19] https://en.wikipedia.org/wiki/Finite_difference_method

[20] http://www.dam.brown.edu/people/alcyew/handouts/numdiff.pdf

boundary of the field is a zero vector, which means that the boids will not be steered in any direction. This will ensure that the obstacle avoidance is C^0 smooth, when the gradient vectors are used to create the steering vector that will be described in Subsection 4.3.5.

The group of six neighbouring points used to approximate the derivative at a point is referred to as the 6-point stencil (circles in Figure 23: left). Using the stencil, finding the gradient vector can also be described with vectors.

First, vectors from the centre point to each of the neighbouring points in the stencil are found (vectors in Figure 23: left and Figure 23: centre). These vectors to surrounding points will be referred to as *stencil vectors*. Then, the lengths of the stencil vectors are set to the corresponding values in the distance field to form the vectors notated by d in the right image of Figure 23. Next, these vectors are summed to find the gradient vector notated by s. However, the gradient vector s is twice as long as the vector found using the formula $\frac{f(p+\Delta p)-f(p-\Delta p)}{2\Delta p}$. To correct this, the vector s could be divided by 2, but because the exact length of the gradient vector is not important for steering boids, this discrepancy is ignored. Instead, the gradient vector's length is set using the distance value in the underlying distance field by the formula $length = 1 - distance$. This makes the vector inversely proportional with the distance to the closest mesh. The result of this is that the fish swims away faster the closer it is to the mesh. As the vector does not fit the formal definition of gradient anymore, it is from now referred to as the *avoidance vector* and is notated by a in Figure 23.

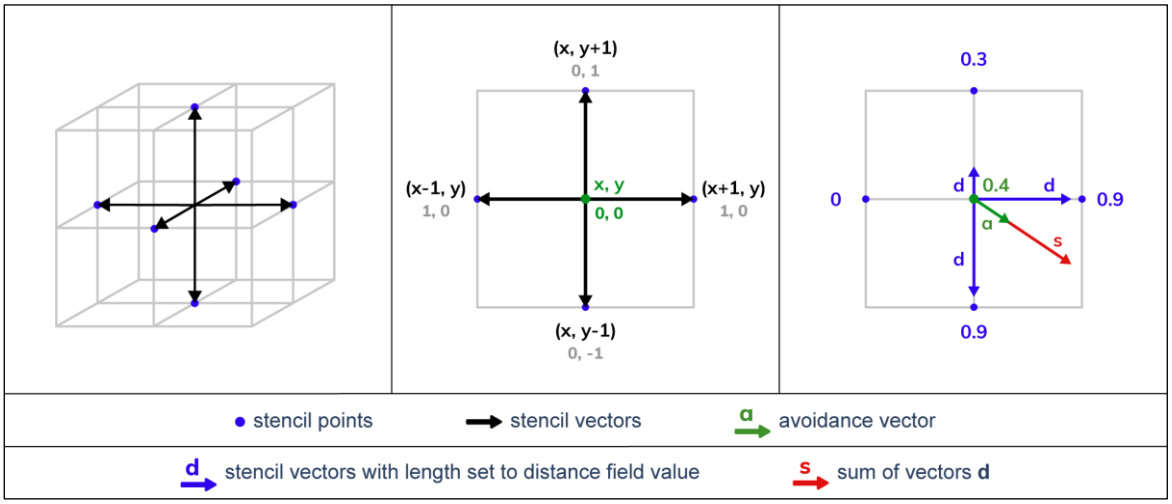

Figure 23. Creating the voidance vector using a 6-point stencil. Left: 6-point stencil. Centre: stencil vectors in 2D and the coordinates to the values in the distance field's array. Right: creating the avoidance vector in 2D.

The avoidance vector steers the boid in the direction where the distance from obstacles increases the fastest. However, obstacles further than `avoidanceRadius` should not be steered away from. Because of this, the values in the underlying distance field larger than the radius are set to the value of `avoidanceRadius`. Thus the values in the field are in the range $[0,\ avoidanceRadius]$.

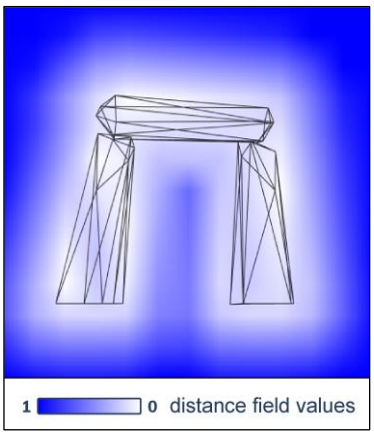

Figure 24. 2D slice of the distance field using avoidRadius.

To make further calculations on the field easier, the field is converted to the range $[0,1]$ by dividing the values by `avoidanceRadius`. The resulting distance field is illustrated in Figure 24. In the following subsection, the avoidance vectors derived from this distance field will be used to create a vector field.

4.3.5 Avoidance Field

Using the described method, the avoidance vector for any discrete point in the distance field can be found (Figure 25: left). Using these discrete points and their corresponding avoidance vectors, a vector field based on the avoidance vectors is created. This field will be referred to as the *avoidance field*. The interpolated values of the avoidance field can be seen in the right image of Figure 25. Using the output of the avoidance field for steering the boids results in very lifelike obstacle avoidance behaviour. In this subsection, some additional improvements to the avoidance field are described.

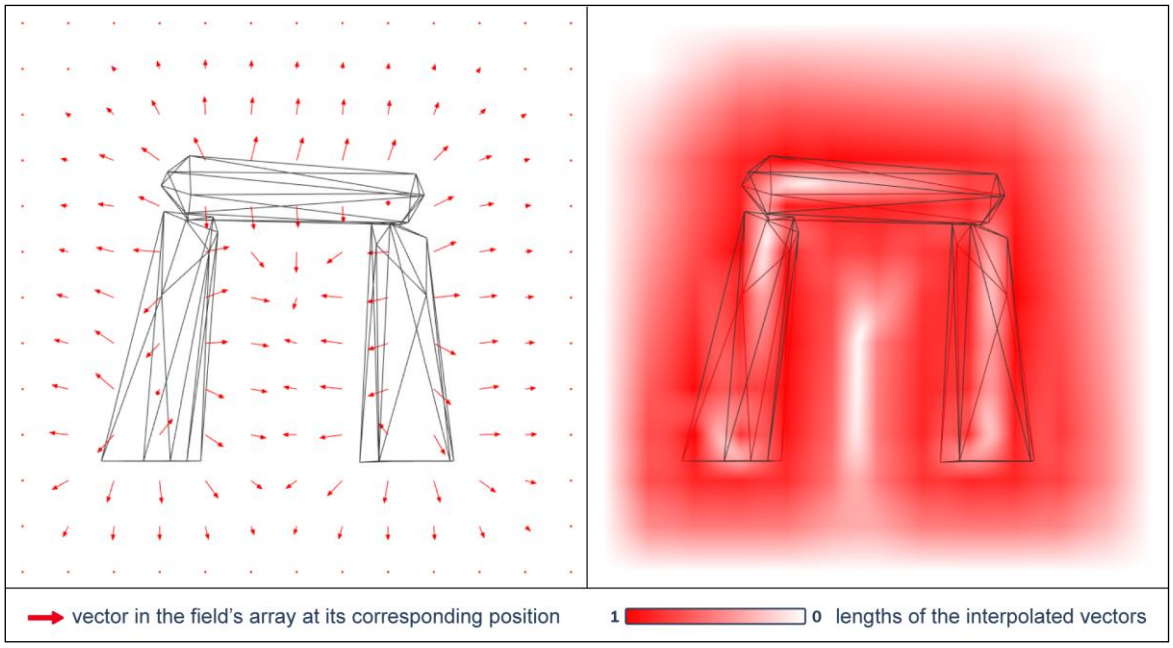

Figure 25. 2D slices of the avoidance field with a mesh inside it. Left: vectors of the field. Right: the length of the interpolated steering vectors for positions on a plane.

The avoidance field can be improved by using a larger stencil than the 6-point stencil used before. In this thesis, a 26-point stencil will be used to find avoidance vectors. Finding the avoidance vector using the 26-point stencil is illustrated in Figure 26.

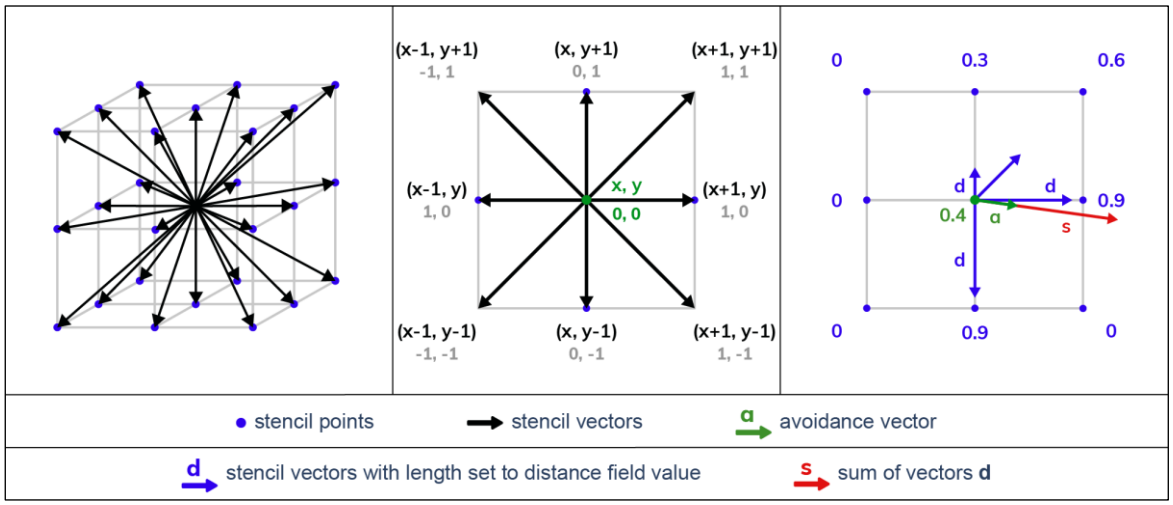

Figure 26. Creating the voidance vector using a 26-point stencil. Left: 26-point stencil. Centre: stencil vectors in 2D the coordinates to the values in the distance field's array. Right: creating the avoidance vector in 2D.

The avoidance field created with the 26-point stencil is illustrated in Figure 27. The avoidance vectors that the field is based on are very similar to the desired vectors that were illustrated in Figure 20. The described 26-point method for finding the avoidance vector is very different from the formal gradient vector and its accuracy for approximating the

derivative is not tested. However, because the empirical evidence shows that it is more accurate than the 6-point method, it is used in this thesis.

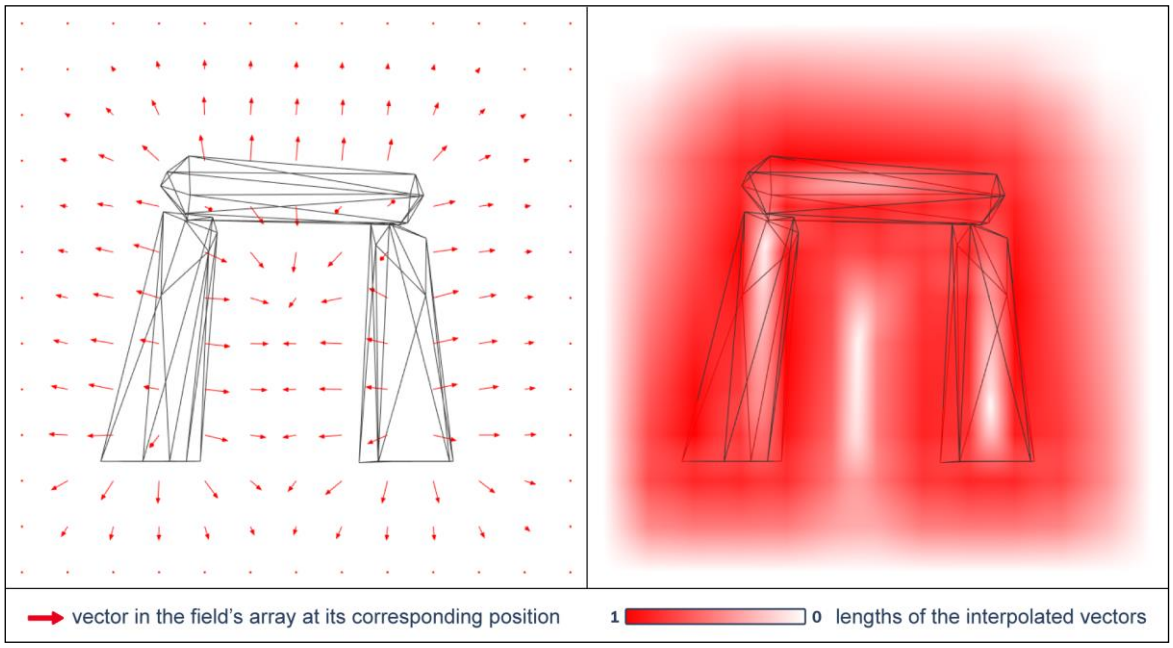

Figure 27. 2D slices of the avoidance field with a mesh inside it. Left: vectors of the field. Right: the length of the interpolated avoidance vectors for positions on a plane.

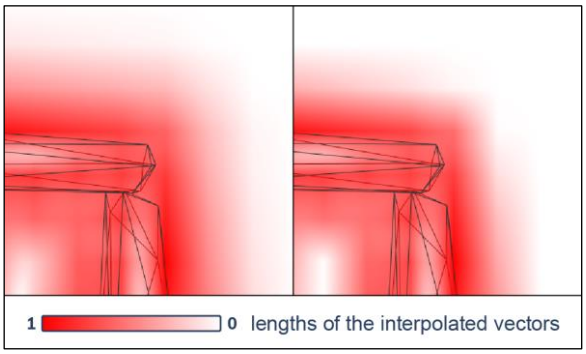

Figure 28. 2D slices of the avoidance field. Left: length of avoidance vectors raised to the power of 2. Right:avoidanceRadius divided by 2.

Another option for improving the field and thus the obstacle avoidance behaviour is making the steering vectors exponentially smaller as the distance to the closest obstacle increases. Making the values decrease exponentially allows the boids to get closer to the mesh while still steering them away fast when too close. This is achieved by raising the length of the avoidance vectors to some power larger than 1. Just making `avoidanceRadius` smaller allows the boids to get closer to the obstacles as well, but it does not produce the exponential effect. Decreasing the `avoidanceRadius` also makes for a less precise avoidance field, due to fewer points in the field having a length other than 1. The difference between raising the avoidance field to the power of 2 and dividing the `avoidanceRadius` by 2 can be seen in Figure 28. The field illustrated in Figure 27 was taken as the base for these changes.

Another way to improve obstacle avoidance is by setting the values of the distance field that are positioned inside the mesh to 0. Without this, values that are inside the mesh, but not

right by the edge of a mesh, can be rather big. Therefore, these points in the field have a weak effect on the direction of the avoidance vector. Thus, the values in the distance field that are positioned inside a mesh should be set to 0.

To determine if a point in the field is inside a mesh, the closest point vector of that field point and the normal of the face that the closest point is on have to be compared. If they are in the opposite direction of each other, then the field point is inside a mesh (Figure 29). Two vectors are opposite to each other if they are set to unit length and the dot product of the vectors is -1. However, this method only applies to convex solid meshes and can cause problems otherwise. Therefore, it is recommended only to be used with the appropriate meshes.

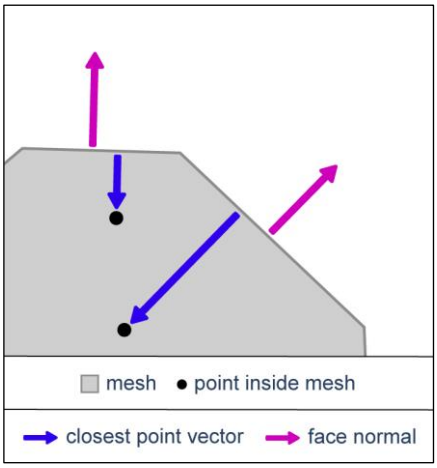

Figure 29. Inside detection.

An avoidance vector found by the method described in this section produces quite lifelike avoidance behaviour. Changing values like the resolution, `avoidanceRadius` and the power the avoidance vectors raised to, can be used to conveniently modify the obstacle avoidance behaviour. In the following chapter, the developed obstacle avoidance and other improvements made to the common Boids implementation are discussed.

5 Results

In this chapter, the results of the thesis are analysed. The goal of the thesis was to make the simulation more lifelike and visually interesting. Thus, mostly the visual aspects of the algorithm and simulation are discussed. The Reynolds' rules and their common implementation described in Section 3.1 to Section 3.5 are referred to as the *original algorithm* and the algorithm produced in this thesis is referred to as the *improved algorithm*.

5.1 Visual Results

The simulations produced by the improved algorithm are visually quite different from the original. As the produced simulations are highly dependent on the chosen scalars and variables, they have been picked in a way that the algorithms and their corresponding simulations are best comparable.

Figure 30 illustrates the difference in the school shapes of the original and improved algorithms' simulations. The improved algorithm produces a much more dynamic simulation with drastic variations in the size of the formed groups and their movement. The original algorithm will usually result in the fish forming a single big group. The dynamic nature of the improved algorithm can mostly be attributed to the addition of the random direction (wander) rule. If having one large group is the desired behaviour for the simulation, then the effect of wander can be reduced by decreasing the value of its scalar.

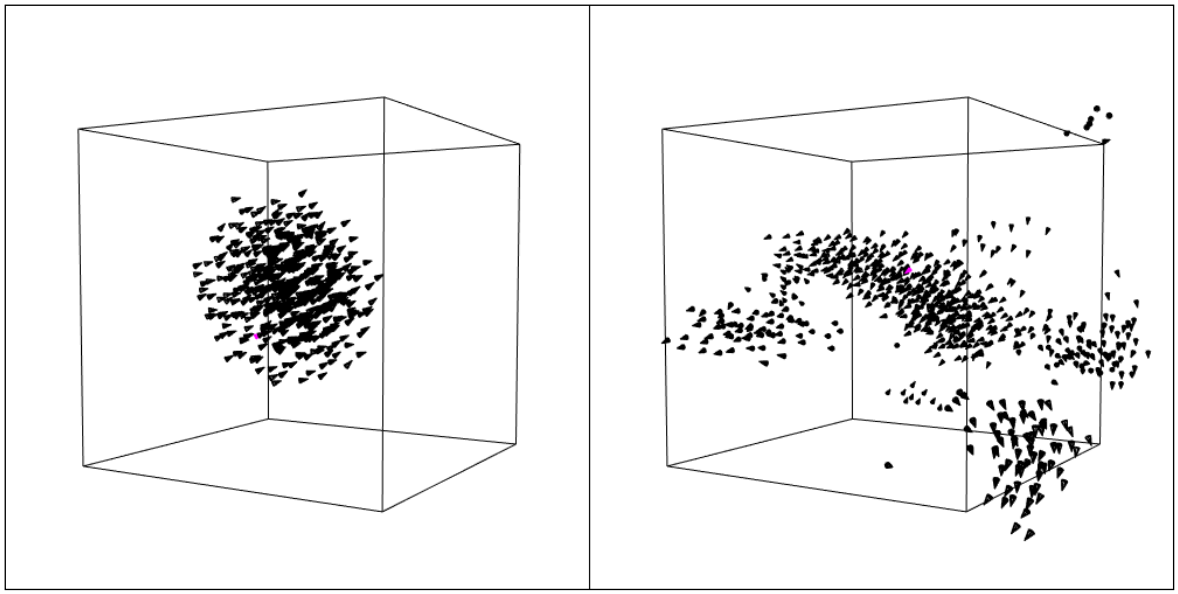

Figure 30. Fish schools. Left: original algorithm. Right: improved algorithm.

If only the three Reynolds' rules (alignment, separation, cohesion) are used, then the differences in the simulations are more subtle. The main distinction between the two is the smoothness of the movement of boids. This is difficult to capture in a static image but can be illustrated by the changes in the acceleration vector of a boid over time. Figure 31 illustrates the x-coordinates of the acceleration vector of a boid using the common and improved algorithms. A video of the two algorithms side-by-side can be found in the accompanying file *common-vs-improved.m4v* (Appendix I).

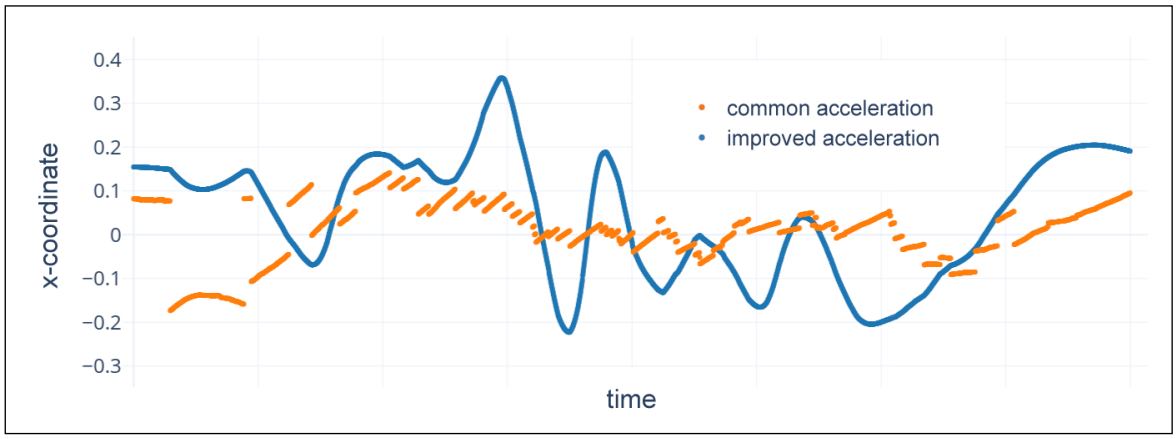

Figure 31. Examples of the x-coordinate of the original and improved algorithm's acceleration vector of a single boid over time using 300 flockmates in the simulation.

From Figure 31 it can be empirically deduced that the improved acceleration is C^0 smooth. Therefore, the velocity is C^1 smooth and the position of the boid is C^2 smooth which leads to the simulations produced using the improved algorithm looking more realistic than the original.

The most complex addition to the algorithm is the obstacle avoidance rule. The steering vector for obstacle avoidance is found using the avoidance field. Figure 32 illustrates the avoidance field with quite a complex mesh inside it. As it is not a convex mesh, inside detection was not used. Figure 33 illustrates boids successfully moving around and through the mesh without colliding with it.

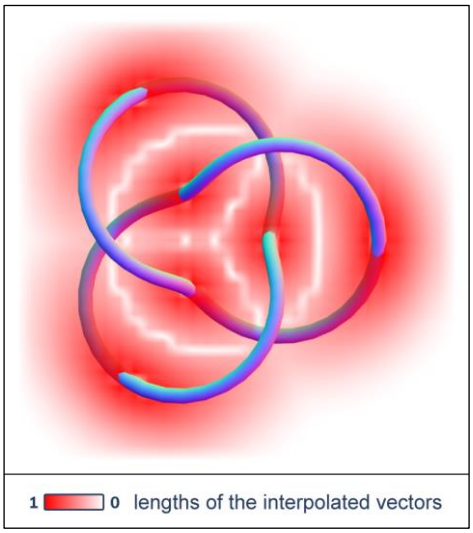

Figure 32. 2D slice of interpolated avoidance vector lengths with a complex mesh inside the field.

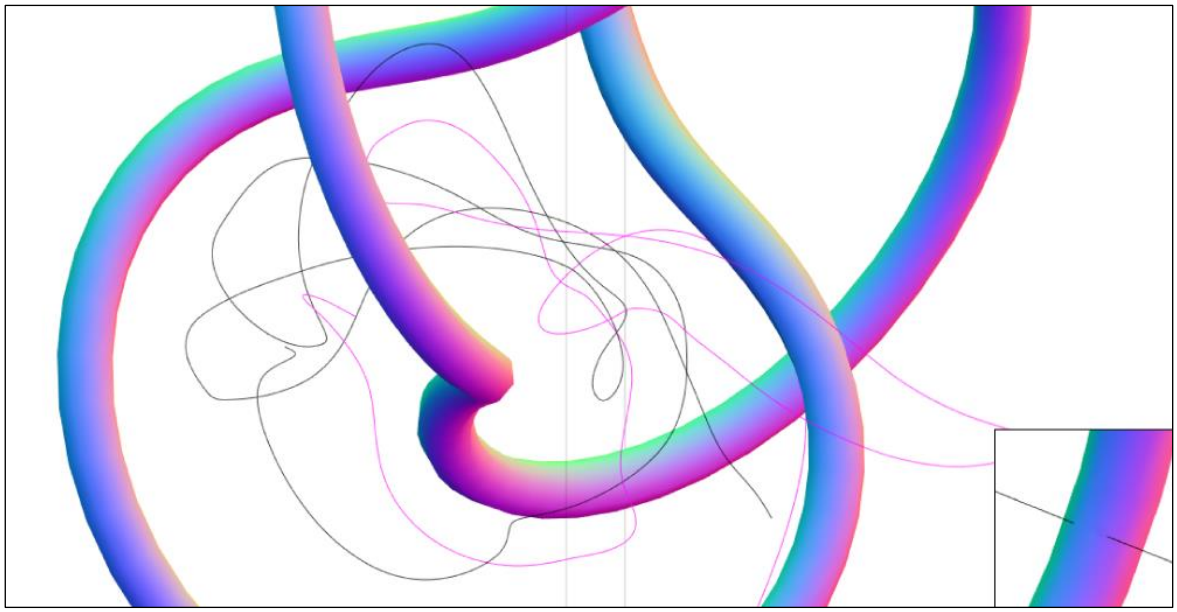

Figure 33. Paths of two boids moving around a complex obstacle without intersecting it. The image in the bottom left corner illustrates what an intersection would look like.

One of the biggest advantages of the developed obstacle avoidance is that the avoidance vectors that the avoidance field is based on can be precomputed. Precomputing a high-resolution avoidance field in a scene with many meshes is a computationally expensive task, but as it can be done before the simulation starts, it does not affect the frame rate of the simulation itself.

Finding the closest point vector on a mesh with 20 thousand faces takes over 16ms as was illustrated in Figure 17. Therefore, it is not practical to be found and used as a steering vector in real-time. Getting the steering vector from the created avoidance field takes under 0.001ms and increasing the face count in the scene does not affect that time. Additionally, the obstacle avoidance behaviour using the avoidance field is more realistic than the closest point vector method for the reasons described in Section 4.3.

The improvements made to the algorithm in this thesis add realism and new types of behaviour to the movement of the simulated fish schools. Additionally, the algorithm is highly adjustable to alter the behaviour of fish schools. These adjustments can be conveniently tested using the demo application built during the creation of this thesis.

5.2 Demo Application

The demo application (Figure 34) was created to test the movement of fish schools while developing the algorithm. However, it also performs as an educational tool for studying the algorithm and the effects of different rules on the behaviour of fish schools. In this section, some of the libraries used for building the demo application are described.

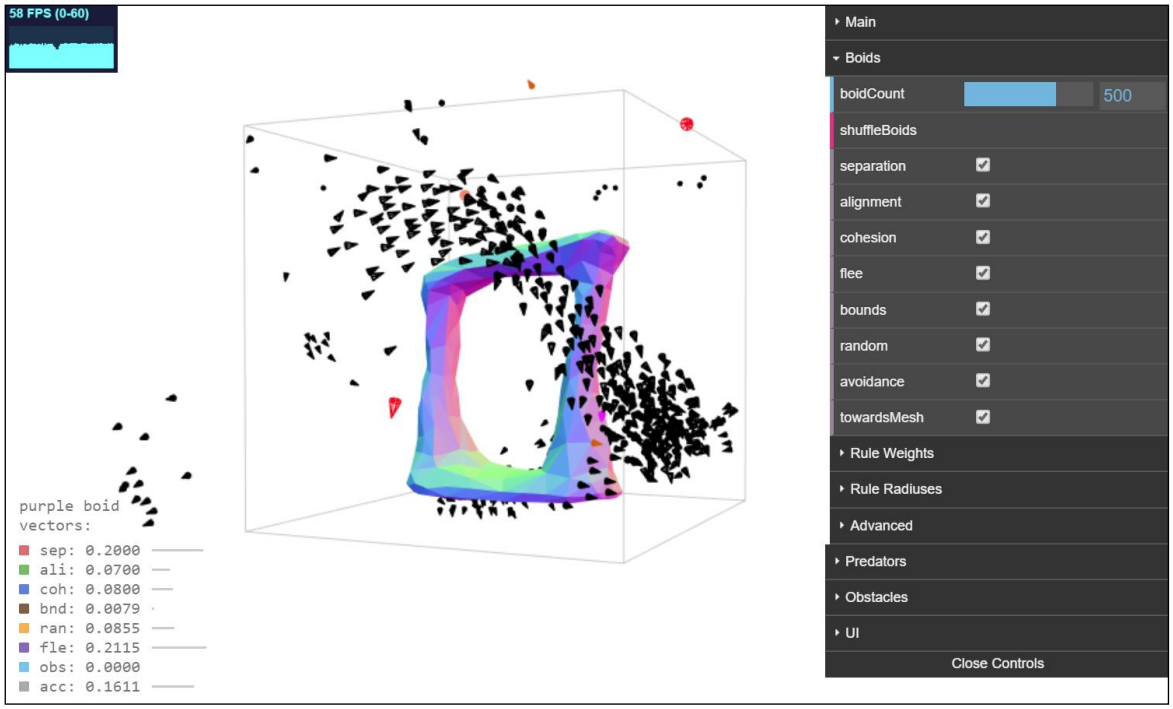

Figure 34. Demo application.

Most of the application was built using the JavaScript library Three.js. It allowed for the convenient setup of the simulation's scene and provided tools for managing the objects in the scene. Additionally, it provided all the functions required for vector calculations.

For the graphical user interface (right side of Figure 34), the JavaScript library dat.GUI[21] was used. The library provides a user-friendly interface to alter different scalars and variables in the application. The user can adjust variables like the speed of the simulation, the number of boids and the scalars of the steering vectors. The performance of the application can be observed using the monitor (top left of Figure 34) provided by the library stats.js[22]. The user can switch between viewing the frame rate, the milliseconds required to render frames and the megabytes of allocated memory.

[21] https://github.com/dataarts/dat.gui

[22] https://github.com/mrdoob/stats.js

In the bottom left corner of the application, the lengths of a single boid's steering vectors are illustrated by the corresponding values and bar lengths. This was manually built using HTML and JavaScript. The steering vectors of a boid in 3D can be seen by selecting `showVectors` in the UI.

Access to the source code of the application is provided in Appendix II. The application and the algorithm could be developed much further and some potential improvements are described in the following section.

5.3 Potential Improvements

The current algorithm has the potential to be further improved in numerous ways. In this section, some improvements regarding the performance, the implementation of the current rules and the inclusion of some additional rules proposed.

5.3.1 Performance

Because in this thesis the performance of the algorithm was not much considered, there are many potential ways of improving performance. The main performance problem with the Boids algorithm is that it runs in $O(n^2)$[23] time when finding steering vectors for the Reynolds' rules. This is because each boid has to be compared against every other boid. Delvin [20] proposes using an octree or a Delaunay graph to reduce the number of necessary comparisons.

For increasing the performance of the demo application, the algorithm could be run on the graphical processing unit (GPU) instead of the currently used central processing unit (CPU). Husselmann and Hawick [21] used parallelism and GPUs to run the algorithm in real-time with over 15 000 boids with consumer hardware from 2011. The demo application in its current form has difficulty to run in real-time 60 frames per second with more than a 1000 boids.

5.3.2 Movement of Boids

The velocity of a boid is limited by a constant `maxSpeed` to avoid the velocity from growing unconditionally. However, this results in the boid always moving with the same speed. A boid should be able to move faster when the acceleration is high and vice versa. Thus,

[23] https://en.wikipedia.org/wiki/Big_O_notation

limiting the velocity by implementing drag could have a significant effect on making the simulation more realistic. Using drag also opens up the possibility to add the hydrodynamic benefit of fish schooling to the algorithm by reducing drag when the boids are positioned in the correct configuration [3] for drag reduction.

As fish do not see directly behind themselves, a perceptual range could be used for the neighbourhood radiuses of the Reynolds' rules and predator avoidance. This can be achieved by comparing the angle of the vector to a flockmate and the boid's rotation. If a flockmate is not in the perceptual range of the boid, then it is ignored. The addition of a perceptual range might quite drastically change the shape of the simulated fish schools.

In video games, it might be desired that in addition to avoiding other fish, the fish avoid the playable character (e.g. scuba diver in the game *ABZU*). To avoid the playable character, the boids separation vector with the character's position can be found. For more precise avoidance, additional points could be added on the character (e.g. on hands and feet) that are separated from.

In video games and films, it might also be desired that the fish schools follow some pre-set path. Petzold, Halle and Thielecke [10] have proposed two different methods for this. The first is based on the fish being steered towards successive waypoints. The second method is based on cylindrical corridors in which the fish have to stay in.

5.3.3 Coefficients

One of the challenges when using the developed algorithm is selecting the weights and radiuses (coefficients) for the rules. As more rules are added to the algorithm, the difficulty of the problem increases significantly. Larsson and Lundgren [22] used a survey method for choosing coefficients by gathering data from volunteers that were asked to choose the most realistic simulation from a set. Alaliyat, Yndestad and Sanfilippo [23] used a genetic algorithm (GA) to pick the coefficients. They defined a desired shape of the flock by using a cost function that included things like the boid's alignment with the flock and distance to the centroid. Chen et al. [24] proposed an interactive GA where a user's subjective evaluation is used as the cost function. Potentially, the boids' actual survival against predators could be used in a GA to find the coefficients that produce the best chance of survival.

To allow multiple different species of fish with varying schooling behaviours, different coefficients can be used. As some species of fish do not school with other species, only specific groups of fish could be used when finding alignment and cohesion.

5.3.4 Appearance of the Fish

In this thesis, the appearance of the fish was not considered at all. Thus, there is a lot of potential to improve the realism of the simulation by making the boids look like actual fish. When animating the fish, only their position was considered. However, the mesh of the fish can also be animated to move the body and fins of the fish. Satoi et al. [25] created a unified motion planner for fishes with various swimming styles. The motion planner decides which parts of the body of the fish should be animated based on its speed.

In addition to animation, realistic meshes and textures can be used to make the fish look more lifelike. By adding meshes with textures to the background of the simulation produced in this thesis as well, a virtual aquarium could be designed.

5.3.5 Avoidance field

The avoidance field created for obstacle avoidance has potential for improvement as well. The accuracy of the 26-point stencil, used for approximating the gradient vector, should be analysed. Using different weights for the different points in the stencil might result in more realistic avoidance behaviour. Interpolating the avoidance vectors with tricubic interpolation instead of trilinear might provide a more accurate result as well.

Although the avoidance field can be precomputed, performance improvements should still be made to decrease the amount of time it takes to compute the field. The main method for improving performance is reducing the number of faces that have to be compared to find the closest point vector. In scenes with multiple meshes, bounding boxes could be used to find the closest meshes before starting to compare distances to their faces. For convex mesh, algorithms exist for finding the closest point on a mesh without comparing each face separately [19]. The data structures mentioned in Subsection 5.3.1 could also be used to reduce the number of distance comparisons needed for finding the closest meshes.

6 Conclusion

In this thesis, an algorithm for simulating the movement of fish schools was developed. The base for the thesis was the Boids algorithm and its common method of implementation. The common implementation was developed further by improving the method for finding steering vectors for the three Reynolds' rules and by adding additional rules to the algorithm. These rules were wander, predator avoidance and obstacle avoidance.

The implementation of the three Reynolds' rules (separation, alignment, cohesion) was improved by using linear decay to reduce the effect of flockmates that are further away from the boid. This resulted in a C^0 smooth acceleration function, which successfully made the movement of boids look more realistic.

The first additional rule, wander added randomness to the boids movement. The steering vector for wander was created by using a 1D noise function for each of the three axes of the vector. The noise function consisted of a random number generator and cubic interpolation to connect the random values. The inclusion of the wander rule to the simulation resulted in the movement of fish schools being more dynamic and it successfully created a lot of variation in the behaviour of the schools.

The second added rule, predator avoidance, was fundamentally very simple but worked well for adding the behaviour of avoiding predators. The third added rule, obstacle avoidance, was the most complex addition of the three. For the obstacle avoidance, first, a distance field was created. That distance field was used to create an avoidance field by using modified gradient vectors of the distance field. This method was used because of its accuracy and the possibility of precomputing the vector field before the simulation starts. The output of the avoidance field used as a steering vector produced good obstacle avoidance behaviour for the fish schools while having almost no impact on performance.

The developed algorithm can be a useful tool for creating animations of fish schools in films and video games. While the focus of this thesis was the simulation of fish schools, the made improvements to the Boids algorithm and the proposed additional rules can likely be adapted to birds and other animal swarms.

Along with the development of the algorithm, a demo application was produced. The demo application is a good showcase of the algorithm in use and it provides a UI for the user to experiment with different coefficients for the rules. The demo application could be made more realistic by improving the appearance of the fish and the landscape of the scene.

Additionally, the schooling algorithm itself could be further improved to increase the realism of the simulation.

There are still numerous opportunities for improving the algorithm. Many improvements that greatly increase the usability of this algorithm in video games have to do with performance optimizations. A proposed option for optimizing performance was reducing the number of distance comparisons that have to be made with flockmates by an octree or a Delaunay graph. Another option is to utilize the GPU for computing the steering vectors.

The movement of fish could be improved by implementing drag and perceptual range. To back up the claims made in this thesis about increased realism, user testing could be conducted to see whether the made assumptions are true. User testing could also provide new ideas on how the algorithm could be improved.

The author of this thesis finds the possibility of using genetic algorithms based on survival for coefficient selection as the most compelling direction of further study. It could be an interesting experiment to see whether the schooling behaviour that emerges with the best survival rate is alike to the schooling behaviour that occurs in nature.

For the assistance and the dedicated involvement in the creation of this thesis, sincere thanks go to the supervisor Raimond-Hendrik Tunnel. Gratitude goes to the entire faculty of the Institute of Computer Science of Tartu University for providing the author with the math and programming knowledge required for the development of the algorithm produced in this thesis.

References

[1] T. J. Pitcher and J. K. Parish, "Functions of Shoaling Behaviour in Teleosts," in *Behaviour of Teleost Fishes*.: Springer Science & Business Media, 1992, pp. 363-365.

[2] C. Cutts and J. Speakman, "Energy Savings in Formation Flight of Pink-Footed Geese," *Journal of Experimental Biology*, vol. 189, pp. 251-261, 1994.

[3] C. K. Hemelrijk, D. A. P. Reid, H. Hildenbrandt, and J. T. Padding, "The increased Efficiency of Fish Swimming in a School," *Fish and Fisheries*, vol. 16, no. 3, pp. 511-521, 2015.

[4] N. Pelechano, J. M. Allbeck, M. Kapadia, and N. I. Badler, *Simulating Heterogeneous Crowds with Interactive Behaviors*. CRC Press, 2016.

[5] T. J. Pitcher, "Shoaling and Schooling Behaviour in Fishes," in *Comparative Psychology: a Handbook*.: Routledge, 1998, pp. 748-760.

[6] C. Liang, "Optimization of Animation Curve Generation Based on Hermite Spline Interpolation," *International Journal of Multimedia and Ubiquitous Engineering*, vol. 11, no. 5, pp. 337-344, 2016.

[7] N. Sun and Y. Tokunaga, "An Alternative Flocking Algorithm with Additional Dynamic Conditions," in *Proceedings of the Ninth International Conference on Broadband and Wireless Computing, Communication and Applications*, Guangdong, 2014, pp. 491-496.

[8] I. L. Bajec and F. H. Heppner, "Organized flight in birds," *Animal Behaviour*, vol. 78, no. 4, pp. 777-789, 2009.

[9] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," in *Proceedings of the 14th annual conference on Computer graphics and interactive techniques (SIGGRAPH)*, 1987, pp. 25-34.

[10] T. Petzold, M. Halle, and F. Thielecke, "Simulation of Flocking Behaviour for a Group of Autonomous Flight Systems," in *Proceedings of the 3rd Workshop on Self-Organization of Adaptive Behavior*, 2004, pp. 178-186.

[11] J. S. Osmundson, T. V. Hunyh, and G. O. Langford, "Emergent Behavior in Systems of Systems," 2008.

[12] T. H. Jessen, *Discovering Computer Science: Interdisciplinary Problems, Principles, and Python Programming*. CRC Press, 2015.

[13] M. Venkitachalam, *Python Playground: Geeky Projects for the Curious Programmer*. No Starch Press, 2015.

[14] A. Downey, *Think Complexity*, 2nd ed. O'Reilly Media, 2018.

[15] K. Stephens, B. Pham, and A. Wardhani, "Modelling fish behaviour," in *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, 2003, pp. 71-78.

[16] A. Lagae et al., "A Survey of Procedural Noise Functions," *Computer Graphics Forum*, vol. 29, no. 8, pp. 2579-2600, 2010.

[17] P. Singh, *OpenGL ES 3.0 Cookbook*. Packt Publishing, 2015.

[18] D. G. Kirkpatrick, M. M. Klawe, and Tarjan R. E., "Polygon triangulation in O(n log log n) time with simple data structures," *Discrete & Computational Geometry*, vol. 7, no. 4, pp. 329-346, 1992.

[19] C. Ericson, *Real-Time Collision Detection*. CRC Press, 2004.

[20] C. Devlin, "An Investigation into an Assortment of Flocking," MSc thesis, Bournemouth University, 2016.

[21] A. V. Husselmann and K. A. Hawick, "Simulating Species Interactions and Complex Emergence in Multiple Flocks-of Boids with GPUs," in *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, 2011.

[22] M. Larsson and S. Lundgren, "Perception of Realistic Flocking Behavior in the Boid Algorithm," BSc thesis, Blekinge Institute of Technology, 2017.

[23] S. Alaliyat, H. Yndestad, and F. Sanfilippo, "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)*, 2014, pp. 643-650.

[24] Y. Chen et al., "Application of Interactive Genetic Algorithms to Boid Model Based Artificial Fish Schools," in *Proceedings of Knowledge-Based Intelligent Information and Engineering Systems*, 2008, pp. 141-148.

[25] D. Satoi, M. Hagiwara, A. Uemoto, H. Nakadai, and J. Hoshino, "Unified motion planner for fishes with various swimming styles," *ACM Transactions on Graphics*, vol. 35, no. 4, 2016.

Appendix

I. Accompanying Files

The ZIP-file containing the accompanying files to this thesis is structured as follows:

- *Source* – the folder containing the source code (see Appendix II) of the demo application.
- *Common-vs-improved.m4v* – video illustrating the differences between the common and improved implementations of the Reynolds' rules (1000 fish in the simulation).
- *Rules-one-by-one.m4v* – video illustrating the effect of all the rules being added to the simulation in succession (1000 fish in the simulation).
- *6000-fish.m4v* – video illustrating the simulation of a school with 6000 fish.

II. Source Code

The source code of the project can be found in the *Source* folder of the ZIP-file included with this thesis.

Contents of the *Source* folder:

- *Js* – the folder containing the JavaScript files
    - *External* – the folder containing all the used libraries
    - *Algorithm.js* – finding the steering vectors
    - *Avoidance-field.js* – creating of the avoidance field
    - *Boids.js* – creating and moving the boids
    - *Controls.js* – the user interface for coefficients
    - *Main.js* – scene setup
    - *Mesh-distance* – finding the closest point on a mesh
    - *Other.js* – various additional functions
- *Index.html* – the file that runs the application when opened with a browser
- *Rocks.glb* – rock model

The application can be run by opening the *index.html* file with any of the following web browsers: *Google Chrome*, *Mozilla Firefox*, *Safari*, *Microsoft Edge*. To import the rock model, the application must be run on an HTTP or HTTPS web server due to browsers' same-origin policy[24] restrictions.

The source code can also be found in a GitHub repository with the following URL: https://github.com/vetemaa/fish-simulation. The source code in the state of the publication of this thesis is marked by the tag *v1.0*[25].

[24] https://en.wikipedia.org/wiki/Same-origin_policy

[25] https://github.com/vetemaa/fish-simulation/releases/tag/v1.0

III. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Erik Martin Vetemaa**

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

Simulating the Collective Movement of Fish Schools,

supervised by Raimond-Hendrik Tunnel,

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Erik Martin Vetemaa

01.05.2020