BSc Computer Science FT

An Investigation into Drone Swarms & Swarm Intelligence

6511 words

Joseph Kidger – Dr Rajesh Chitnis
2191943

1 – Abstract 2
2 – Introduction 2
3 – Specification & Literature Review
3.1 – Challenges & Concepts 3
3.2 – Development Environment 4
4 – Implementation
4.1 – Swarm Intelligence 4
4.1.1 – Separation 4
4.1.2 – Alignment 7
4.1.3 – Cohesion 8
4.2 – Obstacle Avoidance 10
4.3 – Rule Recombination 11
4.4 - Final Implementation in 2D 13
4.5 - Swarm Intelligence Rules in 3D 13
4.6 – Obstacle Avoidance in 3D
4.6.1 - Generating Points about a Sphere 14
4.6.2 - 3D Ray Casting 17
4.7 - Final Implementation in 3D 17
5 – Testing & Parameter Tuning
5.1 - Testing Metrics 20
5.2 - Parameters 21
5.3 - Testing 22
5.4 - Testing Conclusion 24
6 – Project Evaluation & Conclusion 24
7 – Extensions to the Project 25
8 – References 25
9 – Appendix
9.1 – Unity Scene Layout (2D) 26
9.2 – Unity Scene Layout (3D) 26
9.3 – Bugs and Oversights 26

1 – Abstract

The management of drone swarms currently faces multiple logistical issues, particularly concerning collisions between members and with the environment. Groups of animals, such as flocks of birds and schools of fish, exhibit behaviours to stay as a cohesive flock, whilst avoiding injury from collisions. Such behaviours could be applied in the case of simulating drone swarms.

In this project, I suggest expansions to Crag Reynolds' Swarm Intelligence (SI) Model (Reynolds, 1987), [which originally codified behaviours in birds into so-called "oracles", or rules] with the overarching aim to simulate a swarm of drones free of collisions. I experimented with variations for each SI oracle, along with additional proposed oracles, combined with environmental obstacle avoidance, to investigate whether the SI oracles and suggested additions could indeed simulate a successful drone swarm. I first defined and experimented with these technologies in 2 dimensions, until an effective and realistic model was achieved, and then extended this model into complex 3-dimensional environments.

The simulations demonstrated a successful application of these oracles, confirming that Reynolds' model for flocks of birds can indeed be modified and expanded to realistically simulate and guide swarms of drones. Furthermore, with such a large domain of parameter tuning, one can adjust the simulation to simulate a range of swarm types: from natural swarms (e.g., flocks of birds), to synthetic swarms (e.g.: swarms of drones). This gives this simulation a wide range of use cases and adaptability to more complex environments.

2 – Introduction

Creating a Swarm Intelligence model to manage swarms of drones would be beneficial in the real world, allowing for complex drone displays as well as vastly improved logistical operations. For example, projects involving scanning / measuring / dispersal over large areas that are inaccessible or inhospitable would be made feasible with the use of drones.

(Chipade & Panagou, 2019) describe the implementation of drone swarms in the case of defending critical infrastructure from adversarial aerial attackers via coordinated herding.

(Deng, et al., 2022) describe the case of swarm control using a shepherding agent in the case of manipulating micro and nano-particle swarms.

Swarms of drones would be exceptionally suited for these tasks as they are dispensable, inexpensive to produce, and easily configurable. Therefore, investigation into the technologies that could maintain these swarms is beneficial and of interest to the scientific community.

To manage a swarm of drones, one must decide upon some form of intelligence to control the members.

One approach would be to create a separate entity with superior intelligence to control the entire swarm. This has disadvantages in terms of communication between the swarm, as well as computational complexity, requiring a central system to host some Artificial Intelligence.

Another approach would be to instead apply simpler intelligence to each member of the swarm, following some set of rules. (Reynolds, 1987) describes this as Swarm Intelligence.

This approach has been shown to effectively simulate flocks of birds, albeit precalculated, given the hardware of the time. The increased range of information gathering that drones possess (compared to birds), combined with modern hardware could make Swarm Intelligence a viable methodology to manage drone swarms.

3 – Specification & Literature Review

3.1 – Challenges & Concepts

(Reynolds, 1987) discusses the concept of Swarm Intelligence; Consisting of 3 primary rules, Separation, Alignment, and Cohesion, this system combines suggestions from smaller intelligences, using a Rule Recombination Scheme, into a final suggested movement. These oracles (rules) focus on different flocking behaviours: the urge to avoid collisions with other flock members, the urge to follow the same direction and speed as other flock members, and the urge to move towards the centre of mass of the flock. To create a swarm simulation, one must define each of these oracles, as well as a scheme to effectively combine their suggestions. Each of these oracles can be defined in both 2 and 3 dimensions, with the additional dimension adding additional complication.

This project proposes further oracles that aid in smooth flocking behaviour, namely: the urge to continue with one's previous movement, the urge towards a goal point, an acceleration urge (to minimise the time taken to 'reset' after sudden interruptions in movement), as well as a gravitational force.

Furthermore, multiple implementations for each rule and recombination scheme have been suggested and experimented with. (Parker, 2007) suggests a simple recombination scheme which combines the suggestions with simple addition. (Reynolds, 1987), however, suggests three schemes of increasing complexity: simple average, weighted average, and weighted accumulation by importance (up to a maximum acceleration value). These schemes can have drastically different effects on the simulation, and thus have been tested during this project. The rule of separation can also be implemented in a number of ways. In this project I have tested an implementation that I propose, wherein the turning angle of an entity attempting to avoid a collision is inversely related to the distance of the encroaching entity. (Parker, 2007) suggests a simpler separation implementation, wherein the suggested movement is the inverse vector from the origin entity to the encroaching entity.

Previous implementations of Swarm Intelligence have focused on natural swarms, e.g.: (Reynolds, 1987)'s flocks of birds. In this project I have explored the application of these rules to instead simulate swarms of drones in modern environments. (Chipade & Panagou, 2019) propose a similar idea in their system for aerial defence, utilising swarms of drones to 'herd' attackers away from critical infrastructure.

In order to prevent collisions with the environment, this project also describes an Obstacle Avoidance oracle that utilises Ray Casting to locate a safe heading. In 2 dimensions, (Lague, 2019) describes an obstacle avoidance algorithm wherein rays are cast at increasing angles from the direction of movement until a non-colliding ray is found. For 3-dimensional ray casting, one requires the even generation of points about a sphere. (Deserno, 2004) proposes an algorithm to generate equidistant points around a sphere. (Drost, 2017) also proposes an algorithm for this, instead utilising a 3D (even) distribution of the Fibonacci Spiral. (Drost, 2017)'s algorithm is superior for the use case of this project as the points are generated in an enlarging spiral around the sphere, which closely mirrors (Lague, 2019)'s method in 2-dimensions.

3.2 – Development Environment

This project has been developed within the Unity Game Engine, which has allowed for rapid development and rendering of these simulations in real time. Utilising this engine also gives access to the Unity Asset Store where one can gather assets to build realistic environments to test these simulations in.

The layout of the final scenes in 2 and 3 dimensions can be seen here: (see Appendix 9)

4 – Implementation

4.1 – Swarm Intelligence

Without any underlying rules, each member of the swarm (which will be referred to from now on as a Boid) will move freely, without any interaction with any other Boid, as can be seen below:

See "Videos/1 – Boids No Rules.mp4"

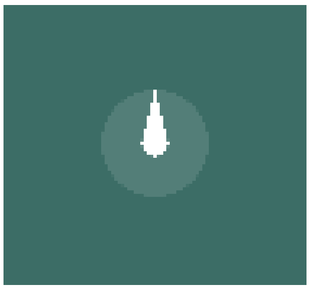
So, we will implement the rules defined in Swarm Intelligence (Reynolds, 1987).

4.1.1 – Separation

First, we will implement the rule of Separation.

(Reynolds, 1987) describes that the Separation rule enforces that a Boid must try to avoid collisions with any nearby Boids.

To start, we will first define an imaginary detection radius that we will refer to as the Separation Radius, as can be seen in the illustration below:

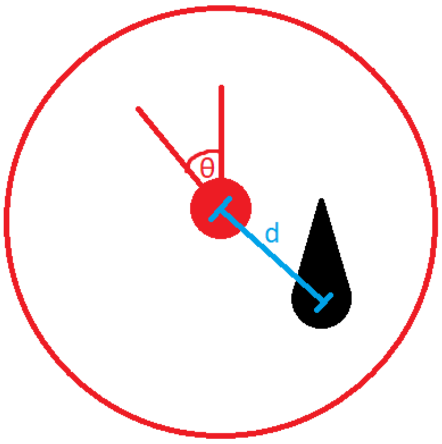


For each frame of the simulation, we will need to check if any other Boids are within this Separation Radius, to do this we will loop through each Boid in the swarm and test if their Euclidean distance is <= the Separation Radius.

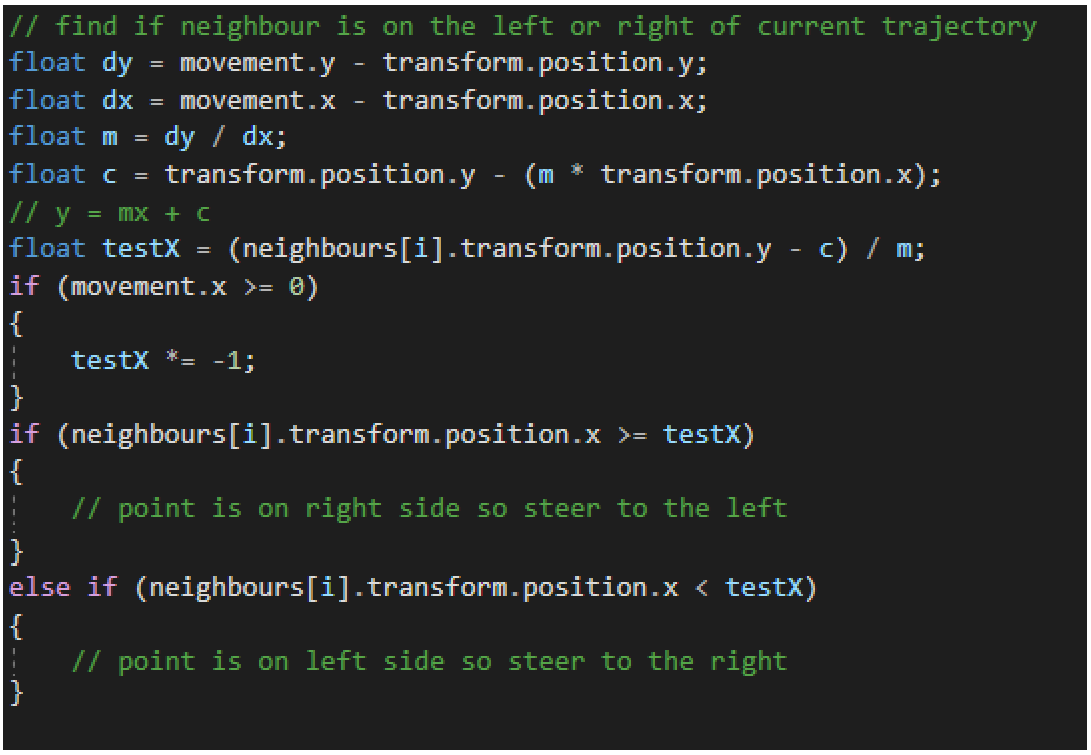
If a Boid is within the Separation Radius, we can propose a new heading, with the intent to prevent collisions with detected Boids.

Separation Method 1

I propose a separation method wherein the turning angle is directly related to the distance from the current Boid to the encroaching Boid.



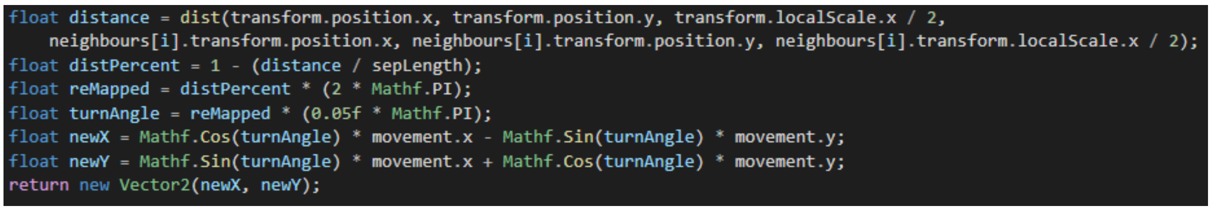
We will first find out which 'side' of our Boid the encroaching Boid in is on. We can do this with the following lines of code:



```
// find if neighbour is on the left or right of current trajectory
float dy = movement.y - transform.position.y;
float dx = movement.x - transform.position.x;
float m = dy / dx;
float c = transform.position.y - (m * transform.position.x);
// y = mx + c
float testX = (neighbours[i].transform.position.y - c) / m;
if (movement.x >= 0)
{
    testX *= -1;
}
if (neighbours[i].transform.position.x >= testX)
{
    // point is on right side so steer to the left
}
else if (neighbours[i].transform.position.x < testX)
{
    // point is on left side so steer to the right
}
```

If the Boid (B2) is on the left side of our Boid (B1)'s path then we must 'steer' to the right, and vice versa for the opposite side.

To define the 'steering' that must occur, we will take the distance that we calculated before and calculate the turning angle:



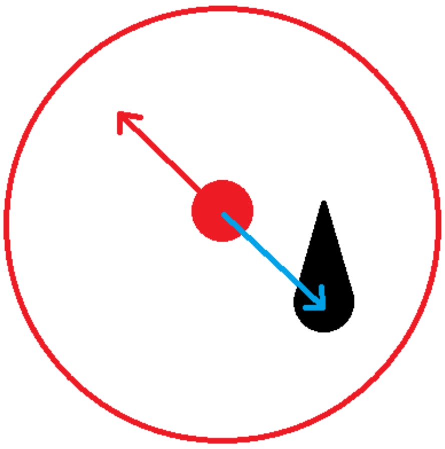
```
float distance = dist(transform.position.x, transform.position.y, transform.localScale.x / 2,
    neighbours[i].transform.position.x, neighbours[i].transform.position.y, neighbours[i].transform.localScale.x / 2);
float distPercent = 1 - (distance / sepLength);
float reMapped = distPercent * (2 * Mathf.PI);
float turnAngle = reMapped * (0.05f * Mathf.PI);
float newX = Mathf.Cos(turnAngle) * movement.x - Mathf.Sin(turnAngle) * movement.y;
float newY = Mathf.Sin(turnAngle) * movement.x + Mathf.Cos(turnAngle) * movement.y;
return new Vector2(newX, newY);
```

This will cause our Boid (B1) to steer more harshly away from the incoming Boid (B2) the closer that B2 is to B1, as can be seen in the video below:

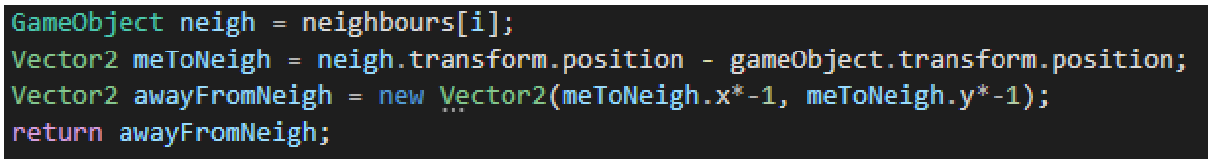
See "Videos/2 – Separation Method 1.mp4"

Separation Method 2

(Parker, 2007) proposes an alternate method, wherein the separation function first calculates the vector from B1 to B2 and returns the inverse of that vector.



The code for this method is shown below:



```
GameObject neigh = neighbours[i];
Vector2 meToNeigh = neigh.transform.position - gameObject.transform.position;
Vector2 awayFromNeigh = new Vector2(meToNeigh.x*-1, meToNeigh.y*-1);
return awayFromNeigh;
```

This forces both Boids to move in direct opposite directions, with the aim to avoid collisions, as can be seen in the example below:

See "Videos/3 – Separation Method 2.mp4"

After experimenting with both separation methods, I have found that although my proposed method has its merits in theory, it does not perform as effectively as (Parker, 2007)'s method. Through testing of the simulation in various scenarios, I have found that my separation method performs very effectively with a small number of detected Boids but does not scale well with a large number of Boids, instead inducing erratic behaviours within the swarm. The simplicity of (Parker, 2007)'s method most likely aids in this regard, also benefiting the simulation with (comparatively) less computation.

As (Parker, 2007)'s separation method is more effective for the context of my simulation, I will be using it from now on.

Below you can see the Boids interacting with one another with **only** separation enabled:

See "Videos/4 – Boids Separation Only.mp4"

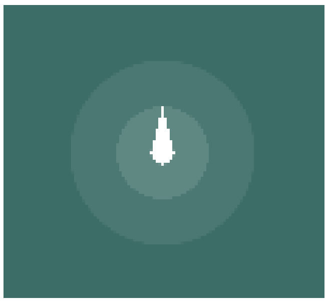
We can experiment with the Separation Radius and Weighting, resulting in a range of swarm layouts: from very sparse / cautious Boids to very close Boids.

4.1.2 – Alignment

We will then implement the rule of Alignment.

(Reynolds, 1987) describes that the Alignment rule enforces that a Boid must try to follow in the same direction as nearby Boids.

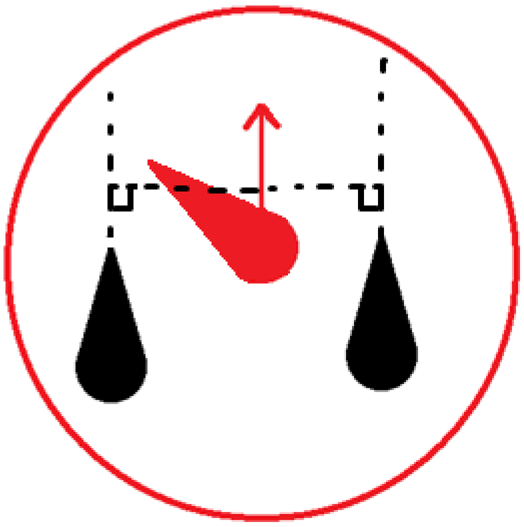
To begin this implementation, we will define another imaginary radius around our Boid, which we will refer to as the Alignment Radius, as can be seen as the larger radius in the illustration below:



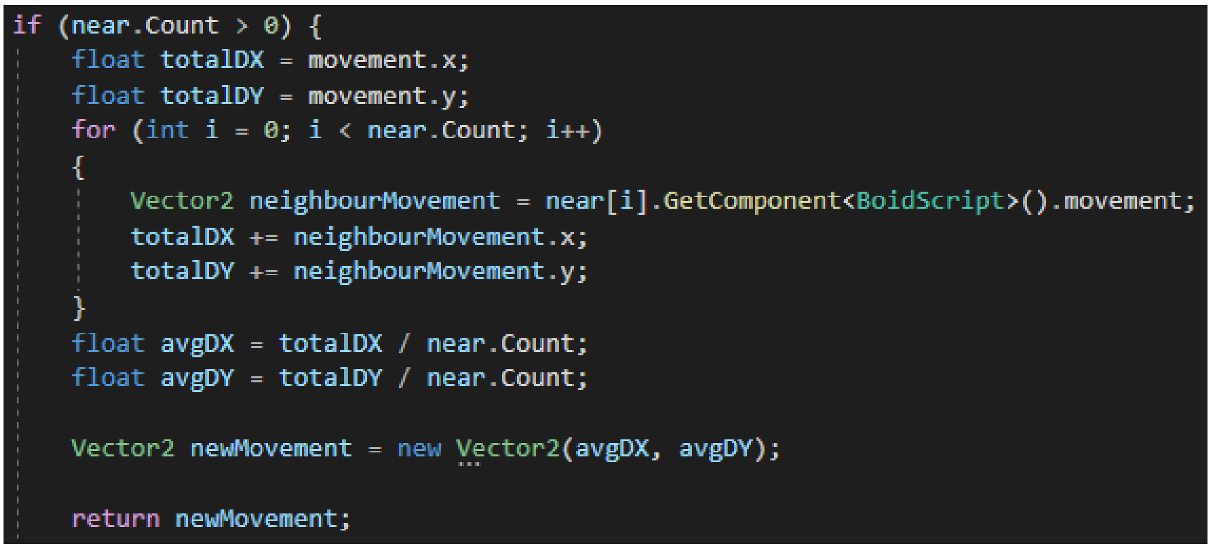
For each frame of the simulation, we will first detect Boids within the Alignment Radius.

We will take every detected Boid and find their movement vector. From this we can calculate the total average movement vector for every nearby Boid (including the primary Boid).

We can then suggest this average movement as the new movement vector (for the primary Boid).



The code for this method is shown below:



```
if (near.Count > 0) {
    float totalDX = movement.x;
    float totalDY = movement.y;
    for (int i = 0; i < near.Count; i++)
    {
        Vector2 neighbourMovement = near[i].GetComponent<BoidScript>().movement;
        totalDX += neighbourMovement.x;
        totalDY += neighbourMovement.y;
    }
    float avgDX = totalDX / near.Count;
    float avgDY = totalDY / near.Count;

    Vector2 newMovement = new Vector2(avgDX, avgDY);

    return newMovement;
```

This will align all nearby Boids into moving in the same direction (and speed), as can be seen in the example below:

See "Videos/5 – Alignment.mp4"

Below you can see the Boids interacting with one another with **only** alignment enabled:

See "Videos/6 – Boids Alignment Only.mp4"

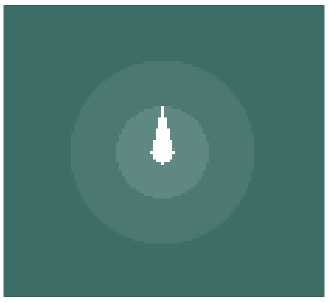
We can experiment with various values for the Alignment Radius and Alignment weighting, resulting in different scopes of vision, as well as different scaling towards Alignment.

4.1.3 – Cohesion

We will then implement the rule of Cohesion.

(Reynolds, 1987) describes that the Cohesion rule enforces that a Boid must try to move towards the centre of mass of all nearby Boids.

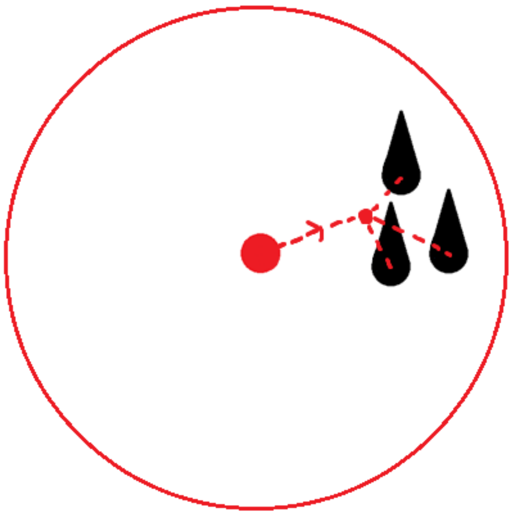
To begin this implementation, we will define another imaginary radius around our Boid, which we will refer to as the Cohesion Radius, as can be seen as the larger radius in the illustration below (Same size as Alignment Radius in this example):



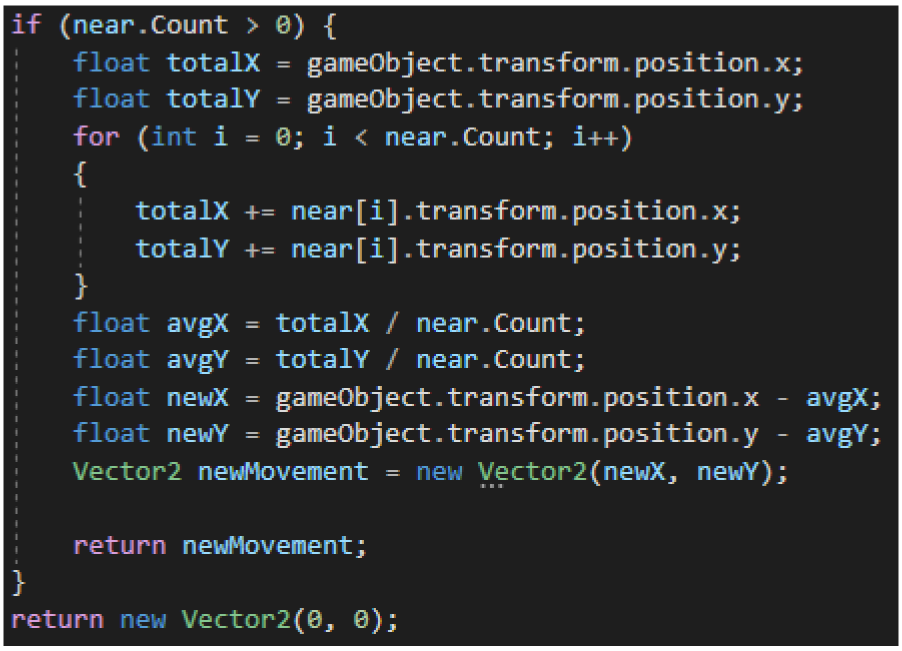
For each frame of the simulation, we will need to check if any other Boids are within this Cohesion Radius.

We will take every detected Boid and find their position coordinates. From this we can calculate the total average position coordinates for every nearby Boid (including the primary Boid).

We will then calculate a movement vector from our Boid's current position to the average position (centre of mass) of all the nearby Boids. In a real-world implementation without a universal coordinates system, we could work out the centre of mass by the relative distance between each Boid.



The code for this method is shown below:



```
if (near.Count > 0) {
    float totalX = gameObject.transform.position.x;
    float totalY = gameObject.transform.position.y;
    for (int i = 0; i < near.Count; i++)
    {
        totalX += near[i].transform.position.x;
        totalY += near[i].transform.position.y;
    }
    float avgX = totalX / near.Count;
    float avgY = totalY / near.Count;
    float newX = gameObject.transform.position.x - avgX;
    float newY = gameObject.transform.position.y - avgY;
    Vector2 newMovement = new Vector2(newX, newY);

    return newMovement;
}
return new Vector2(0, 0);
```

This will get the Boids to stay together, AKA cohere together into a flock of sorts, as can be seen in the example below (with **only** cohesion enabled):

See "Videos/7 – Boids Cohesion Only.mp4"

By experimenting with the Cohesion Radius and Weighting, we can fine tune a realistic detection radius and scale, with too small a radius/scaling resulting in minimal flocking, and too large a radius/scaling resulting in irregular behaviour as the Boids get overwhelmed with cohering together.

4.2 – Obstacle Avoidance

With (Reynolds, 1987)'s Swarm Intelligence rules applied, the Boids will simulate flocking behaviour and avoid colliding with one another but will not attempt to avoid any collisions with the environment.

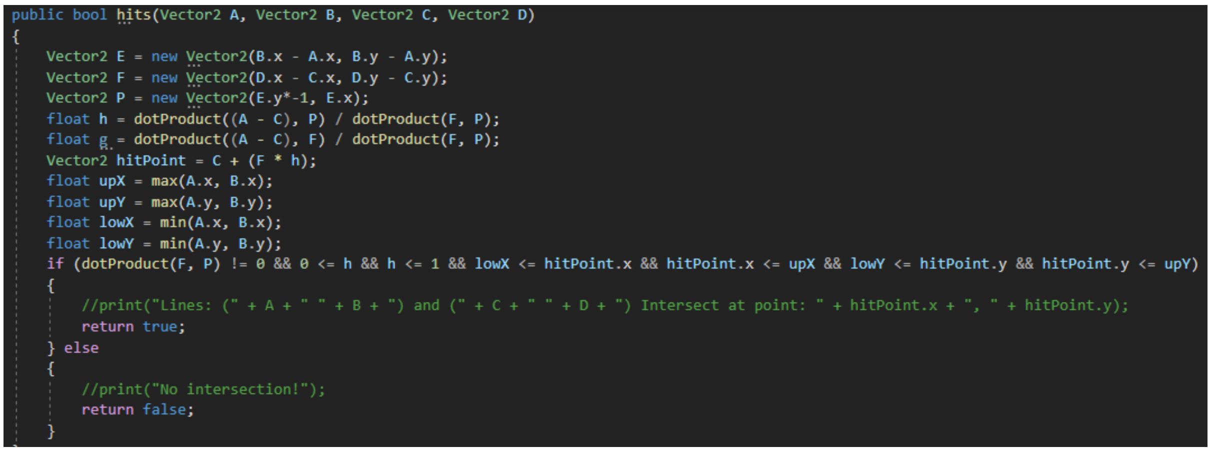
To solve this issue, we will implement an Obstacle Avoidance oracle (rule).

To detect whether a collision with an obstacle is incoming and to find a safe heading, I have decided to implement Ray Casting.

Ray Casting

For each frame of the simulation, we will first cast a ray out in the direction of travel from our Boid, and then check if this ray intersects with any objects.

To check for intersection in 2 dimensions, we will use the following function:



```
public bool hits(Vector2 A, Vector2 B, Vector2 C, Vector2 D)
{
    Vector2 E = new Vector2(B.x - A.x, B.y - A.y);
    Vector2 F = new Vector2(D.x - C.x, D.y - C.y);
    Vector2 P = new Vector2(E.y*-1, E.x);
    float h = dotProduct((A - C), P) / dotProduct(F, P);
    float g = dotProduct((A - C), F) / dotProduct(F, P);
    Vector2 hitPoint = C + (F * h);
    float upX = max(A.x, B.x);
    float upY = max(A.y, B.y);
    float lowX = min(A.x, B.x);
    float lowY = min(A.y, B.y);
    if (dotProduct(F, P) != 0 && 0 <= h && h <= 1 && lowX <= hitPoint.x && hitPoint.x <= upX && lowY <= hitPoint.y && hitPoint.y <= upY)
    {
        //print("Lines: (" + A + " " + B + ") and (" + C + " " + D + ") Intersect at point: " + hitPoint.x + ", " + hitPoint.y);
        return true;
    } else
    {
        //print("No intersection!");
        return false;
    }
```

This function was designed with methodology as described by Gareth Rees in this forum post: https://stackoverflow.com/questions/563198/how-do-you-detect-where-two-line-segments-intersect

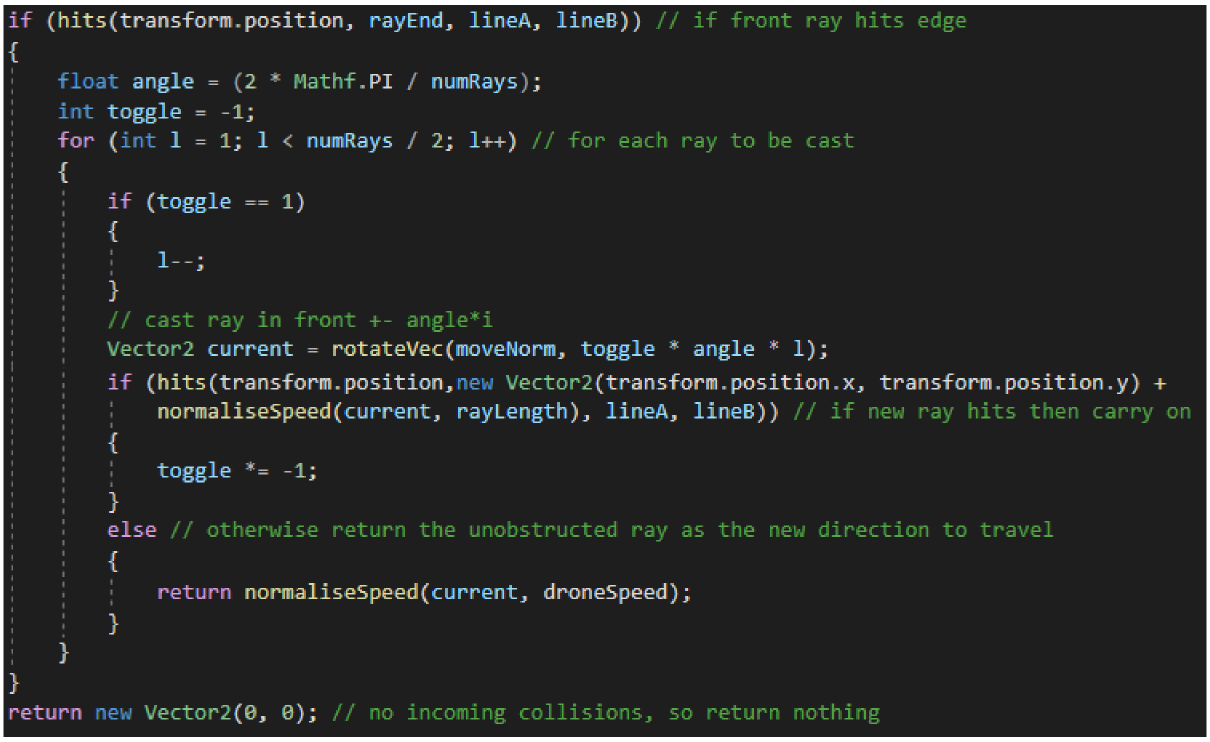
(Lague, 2019) describes a ray fanning algorithm, wherein we cast rays at increasing angles from the movement heading, alternating left and right.

Shown below is a visualisation of this ray fanning algorithm:

See "Videos/8 – Lague Ray Fanning.gif"

(sourced from (Lague, 2019)*)*

If the front ray intersects with an object, we will then begin the route-finding loop as described below:



```
if (hits(transform.position, rayEnd, lineA, lineB)) // if front ray hits edge
{
    float angle = (2 * Mathf.PI / numRays);
    int toggle = -1;
    for (int l = 1; l < numRays / 2; l++) // for each ray to be cast
    {
        if (toggle == 1)
        {
            l--;
        }
        // cast ray in front +- angle*i
        Vector2 current = rotateVec(moveNorm, toggle * angle * l);
        if (hits(transform.position,new Vector2(transform.position.x, transform.position.y) +
            normaliseSpeed(current, rayLength), lineA, lineB)) // if new ray hits then carry on
        {
            toggle *= -1;
        }
        else // otherwise return the unobstructed ray as the new direction to travel
        {
            return normaliseSpeed(current, droneSpeed);
        }
    }
}
return new Vector2(0, 0); // no incoming collisions, so return nothing
```

This pseudocode casts rays at alternating angles from the frontRay (left, right, left, right… from "frontRay" by ("it" x "angle") radians), checking if any of these rays do not collide with the object, and if any ray does not collide then that ray is returned as the new direction of travel.

This allows the Boid to detect incoming collisions with objects in the world and generate a safe direction to head in.

Once we implement this algorithm, we can see illustrated below that our Boid tries to avoid the obstacle by finding a non-colliding route:

See "Videos/9 – Static Obstacle Avoidance.mp4"

We can now allow the Boid to move, and demonstrate the Collision Avoidance in motion, as can be shown below:

See "Videos/10 – Boid Obstacle Avoidance.mp4"

Notes

This OA implementation went through numerous iterations not only for optimisation but also to fix a number of bugs, most notably both: a bug where the Boid would occasionally fly straight through an obstacle when the other oracles provided inverse suggestions, as well as a bug where the Boid would fly through corners of an obstacle / fly directly through a thin obstacle. (See Appendix 9.3)

4.3 – Rule Recombination

Now that the oracles (rules) are providing us suggestions for movement, we need to define some scheme to recombine these suggestions.

There are several schemes that one could implement, ranging from very simple, yet ineffective, to more complex:

Recombination Scheme 1: Importance

The simplest scheme would be to choose the suggestion that we say is “most important”. For this we would need to order our oracles in order of importance and then choose the suggestion of the highest precedence.

This scheme is very ineffective, and we can do much better by combining the suggestions together.

Recombination Scheme 2: Average

(Reynolds, 1987) proposes another simple scheme wherein one would calculate the average of all the suggestions.

This scheme is simple to implement but comes with disadvantages. For example, this scheme assumes equal “importance” for each of the oracles. This can be ineffective as high danger rules (i.e.: Separation and Obstacle Avoidance) might be overruled and cause collisions with members of the swarm, or even worse, objects in the environment.

Recombination Scheme 3: Weighted Average

Building upon the previous scheme, (Reynolds, 1987) further describes how we can apply a weighting to each oracle, so that more “important” suggestions are applied a higher weighting.

Recombination Scheme 4: Top N Important

For this scheme we would need to first sort the oracles into order of “importance” and apply weightings.

Then upon every frame of the simulation, the top N oracles would be averaged together.

This scheme works on the premise that the most “important” oracles (e.g.: Separation and Collision Avoidance) are only required in times on high danger, and the less “important” (flocking) oracles are required otherwise.

As during normal operation (ie: no danger), Separation and Collision Avoidance should return no suggestion, and the flocking oracles will take effect. However, during times of high danger, the low importance oracles will be ignored, and the high importance oracles will take effect.

Recombination Scheme 5: Capped Average Summation

(Reynolds, 1987) also goes on to describe the basis of a more complex scheme that one could implement, although deemed unnecessary in 1987.

This scheme consists of three steps:

- Sort oracles by importance
- Apply weighting to oracles
- Begin summation average up to cap

The “summation average up to cap” consists of a simple loop which keeps track of the magnitude (speed) of the summed suggestions, and averages them together, one by one, while making sure that the magnitude is <= some cap (defined as droneSpeed in the parameter list).

This scheme builds upon Scheme 4 by utilizing a magnitude cap instead of the top N oracles. This less rigid structure allows for more fluid introduction of the less “important” rules.

As discussed previously, this is particularly effective as during times of high danger, more “important” rules would be enforced first and less “important” rules ignored, and during times of low danger, the less “important” rules would instead take effect as the “important” oracles would return nothing (i.e.: flocking rules take precedence when there is no danger).

Throughout testing, schemes 4 and 5 both performed to a much greater extent compared to the simpler schemes. As scheme 5 has a slight edge in fluidity over scheme 4, it will be used from now on.

Additional Oracles / Urges

For this simulation to run smoothly, I have implemented some additional Oracles which will also contribute to the final heading:

- Previous Movement Oracle
  - ⇒ Urges Boid to continue upon its previous heading
    - Prevents Boids from getting “stuck” when they have no information
- Goal Oracle
  - ⇒ Urges Boid towards the goal
- Acceleration Urge
  - ⇒ Applies a small acceleration to the Boid
    - Prevents Boids from getting stuck at very low speeds

4.4 - Final Implementation in 2D

Now that we have created each oracle and developed a scheme to combine them into one heading, we can now release the Boids into the 2D space with a goal that they will be urged towards.

Final 2D simulation:

See “Videos/11 – Boids 2D Final.mp4”

In this simulation, we can see that the Boids effectively form a flock, as well as splitting into sub-flocks when avoiding obstacles in the environment.

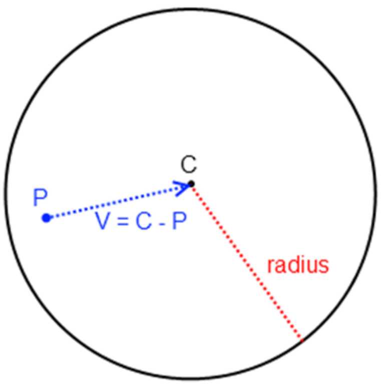
The final Unity scene is organised as such: (see Appendix 9.1)

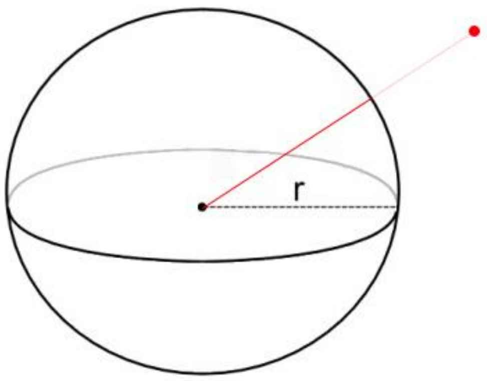
4.5 - Swarm Intelligence Rules in 3D

Now that we have defined and implemented our algorithms in 2 dimensions, we can begin to extend these to 3 dimensions.

The base rules for Swarm Intelligence can be extended to the third dimension with relative ease.

This can be conceptualised by thinking of the detection circles from the 2D simulation as detection spheres in the 3D simulation.





The conversion process consists of modifying the Separation, Alignment, and Cohesion algorithms to also include the Z coordinate.

One major component that contributes to the ease of conversion is the detection mechanism in which we are using. To detect neighbouring Boids, we compare the Cartesian distance from our Boid to every Boid in the scene, and test whether each distance is <= the detection radius.

This detection scheme acts the same in 3 dimensions, as we can calculate the Cartesian distance between points with relative ease and compare that to the radius of the detection sphere.

4.6 - Obstacle Avoidance in 3D

Unlike the Swarm Intelligence rules, Obstacle Avoidance in 3 dimensions is much trickier.

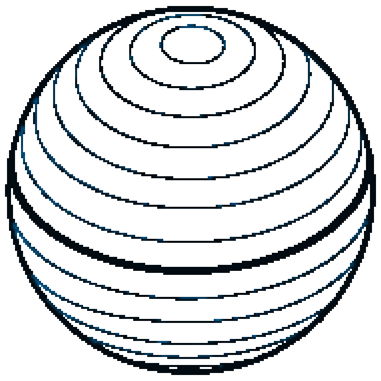
This is made very clear when we attempt to generate points about a sphere.

The problem in 2 dimensions can be simplified to generating points about a circle, which is trivial (define an increasing angle with a static radius).

However, in 3 dimensions we would need to generate points about a sphere.

4.6.1 - Generating Points about a Sphere

(Deserno, 2004)'s algorithm could be effective in this case as this algorithm generates equidistant points around a sphere. However, the order of these generated points is in rings around the sphere. We could consider a method using these rings, by rotating points and iterating through each ring to branch out.

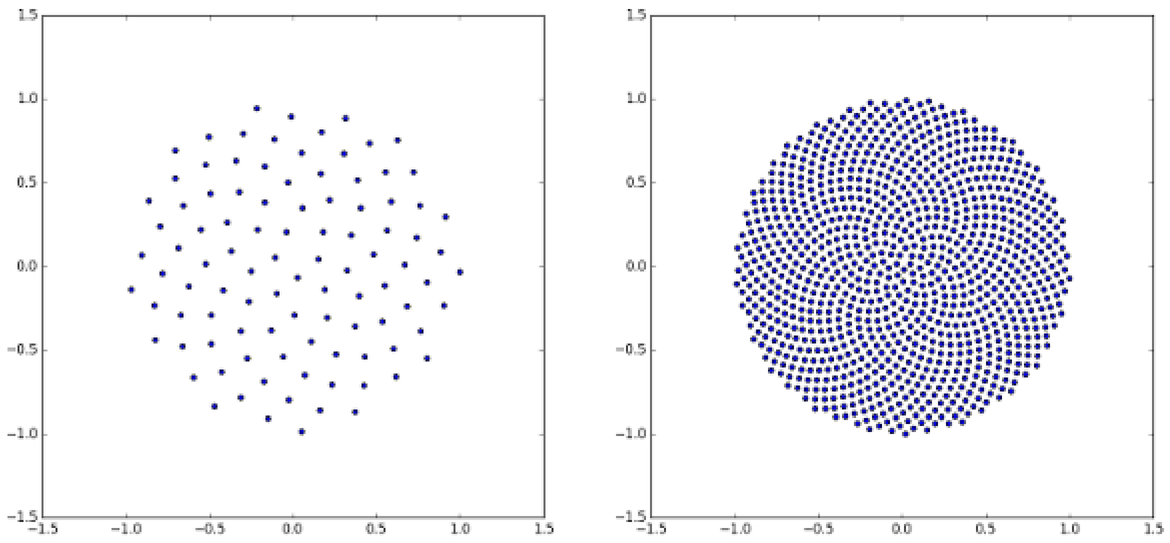


(Drost, 2017)'s algorithm, however, is more beneficial in our case.

(Drost, 2017)'s Algorithm

To generate points about a sphere, we can consider the Fibonacci Spiral.

Fibonacci spiral visualised in python with n = 100 and 1000 respectively



Sourced from CR Drost's response on:
https://stackoverflow.com/questions/9600801/evenly-distributing-n-points-on-a-sphere

In 2 dimensions, we can generate a spiral using the Golden Ratio (1.618...) (denoted by φ) as the turn fraction value.

(Drost, 2017) describes an algorithm to generate this spiral in 2 dimensions (in Python):

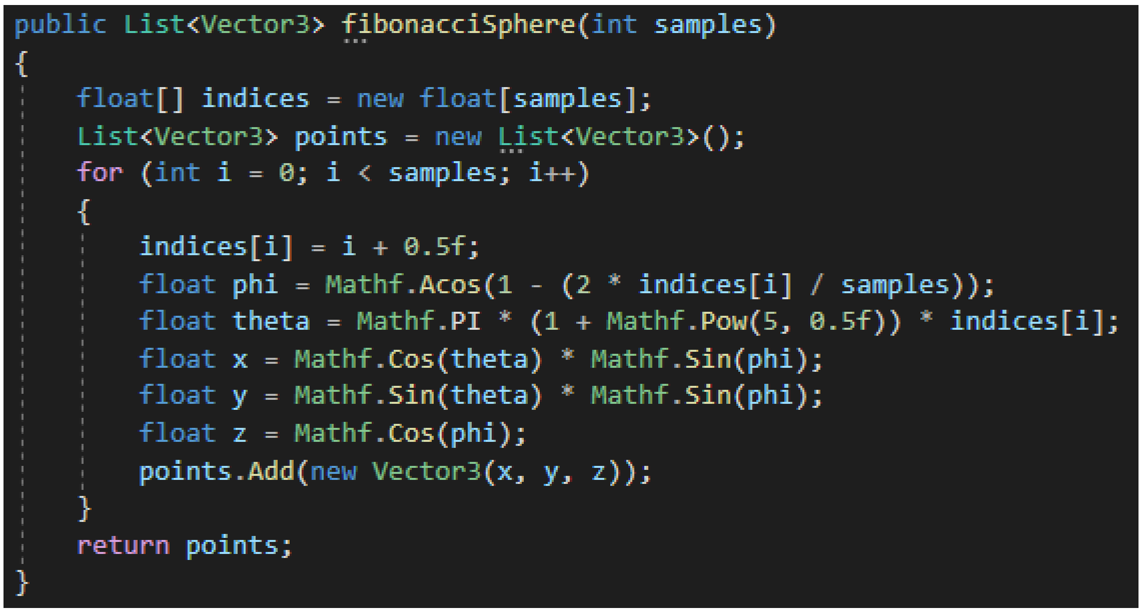
```python
from numpy import pi, cos, sin, sqrt, arange
import matplotlib.pyplot as pp

num_pts = 100
indices = arange(0, num_pts, dtype=float) + 0.5

r = sqrt(indices/num_pts)
theta = pi * (1 + 5**0.5) * indices

pp.scatter(r*cos(theta), r*sin(theta))
pp.show()
```

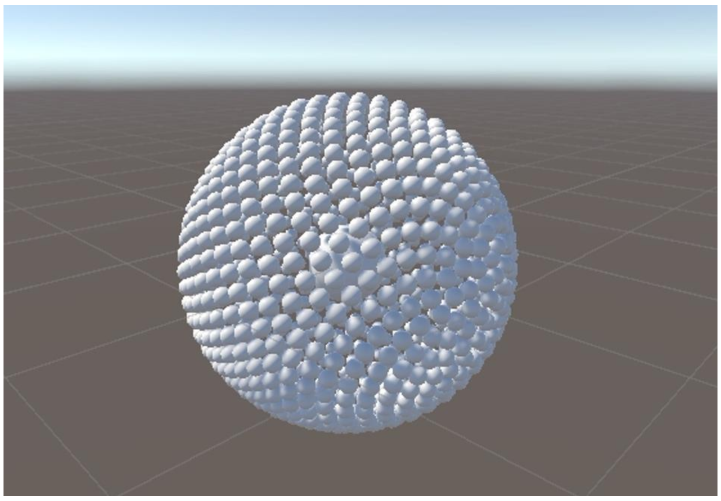
We can then project this spiral into 3 dimensions using the function described below (in C#):



```
public List<Vector3> fibonacciSphere(int samples)
{
    float[] indices = new float[samples];
    List<Vector3> points = new List<Vector3>();
    for (int i = 0; i < samples; i++)
    {
        indices[i] = i + 0.5f;
        float phi = Mathf.Acos(1 - (2 * indices[i] / samples));
        float theta = Mathf.PI * (1 + Mathf.Pow(5, 0.5f)) * indices[i];
        float x = Mathf.Cos(theta) * Mathf.Sin(phi);
        float y = Mathf.Sin(theta) * Mathf.Sin(phi);
        float z = Mathf.Cos(phi);
        points.Add(new Vector3(x, y, z));
    }
    return points;
}
```

(Drost, 2017) describes how this conversion is essentially just swapping the polar coordinates in 2 dimensions for spherical coordinates in 3 dimensions.

Upon running this function in Unity, we can see two valuable properties arise:



This method generates an even distribution of points about a sphere, under some defined number of points (1000 points gives a good distribution for this use case).

But upon further inspection, we notice a much more valuable property: the order of generated points.

As we iterate through the list of generated points, we move around the sphere in a spiral, essentially doing what our 2D implementation did: increasing the angle from the origin, except now in 3 dimensions.

This is shown in the example below:

See "Videos/12 – Fibonacci Points.mp4"

This property is especially useful to us as, for our obstacle avoidance algorithm, we want to cast a ray in the direction of movement, and then if a collision is detected, cast a new ray out in a new direction at an increased angle from the origin. The order of our points allows us to simply iterate through the list as each collision is detected.

The order of ray casting is visualised in the example below:

See "Videos/13 – Fibonacci Rays.mp4"

4.6.2 - 3D Ray Casting

Now that we can generate points around a sphere, we can cast rays to / through these points to detect potential collisions.

Unity Ray Casting

For my final implementation, I would like for the Boids to be able to navigate around complex environments, which intern will consist of complex shapes.

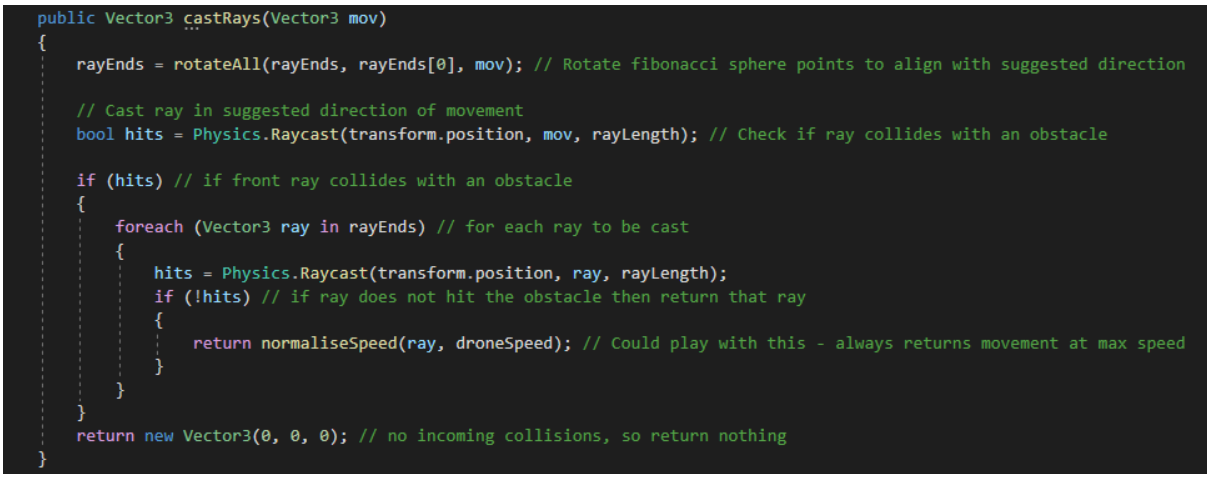
My 2D implementation used a function that would detect whether two lines intersected, which is somewhat simple to define in 2 dimensions. However, for 3 dimensions the complexity of the shapes in the environment would require a much more complex detection function / library.

So, to simplify the collision detection, I will be utilising Unity's Ray Casting library.

This allows us to detect whether our ray collides with any GameObject that has a Collider attached to it. Therefore, we can import objects with complex meshes and attach Mesh Colliders to them, allowing for our obstacle avoidance to work seamlessly with any object.

We also need to rotate the sphere of points such that the first point aligns with our direction of movement.

Shown below is the final Ray Casting function:



```
public Vector3 castRays(Vector3 mov)
{
    rayEnds = rotateAll(rayEnds, rayEnds[0], mov); // Rotate fibonacci sphere points to align with suggested direction

    // Cast ray in suggested direction of movement
    bool hits = Physics.Raycast(transform.position, mov, rayLength); // Check if ray collides with an obstacle

    if (hits) // if front ray collides with an obstacle
    {
        foreach (Vector3 ray in rayEnds) // for each ray to be cast
        {
            hits = Physics.Raycast(transform.position, ray, rayLength);
            if (!hits) // if ray does not hit the obstacle then return that ray
            {
                return normaliseSpeed(ray, droneSpeed); // Could play with this - always returns movement at max speed
            }
        }
    }
    return new Vector3(0, 0, 0); // no incoming collisions, so return nothing
}
```

4.7 - Final Implementation in 3D

Now that each of the algorithms have been defined in 3 dimensions, we can combine them using our desired recombination scheme and generate some Boids.

See "Videos/14 – 3D Boids in Basic Space.mp4"

In the example above, we can see the swarm formation in a simple environment. We can see that they effectively flock together, avoiding collisions while seeking the goal.

Gravity

For this simulation I would like for the Boids to be affected by gravity, therefore I have decided to implement a gravitational force.

This will limit the ascension speed of the Boids, and cause them to fall at a much greater rate.

This will increase the realism of the simulation, creating a more realistic representation of how drones fly in the real world.

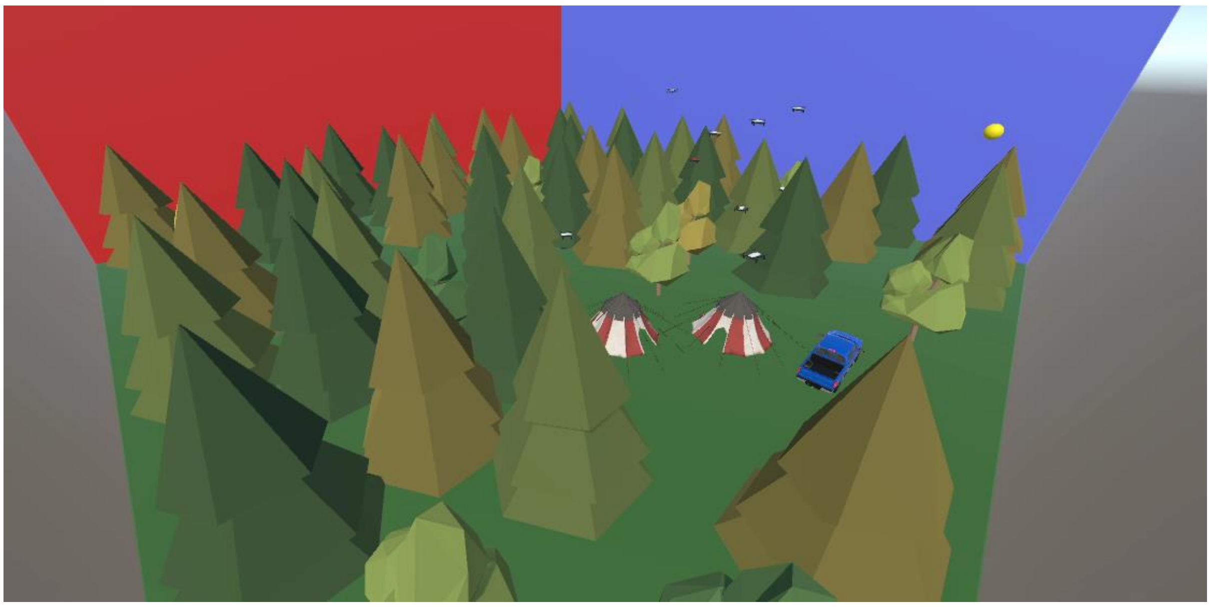
This value will be experimented with to find a good value to approximate drone movement.

Complex Environments

Although we have seen success in the 3D simulation, I would like to test this implementation in a more complex environment. Utilising the Unity Asset Store, one can create a wide variety of environments to test and showcase how well this implementation performs. I have also converted the Boid body to a drone model that I sourced from the Asset Store.

To effectively test and showcase the performance of this simulation, I have created two complex modern environments for the Boids to flock within.

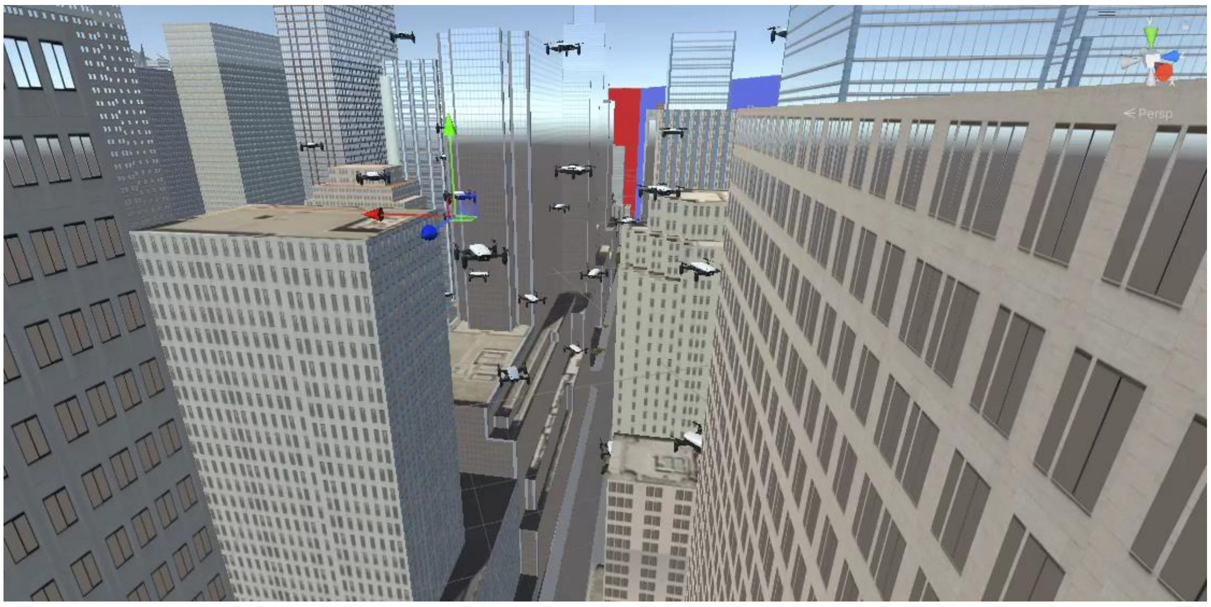
The first curated environment is a forest scene, containing a combination of open-air space towards the top of the scene, as well as a complex series of objects (trees) towards the lower end of the scene.



This environment can be seen in the example below:

See "Videos/15 – 3D Boids in Forest Scene.mp4"

The second environment consists of a modern city, utilising the asset created by INSERT. This scene consists of a large area of fairly uniform buildings, which would often split the swarm as they traverse the city.



This environment can be seen in the example below:

See "[Videos/16 – 3D Boids in City Scene.mp4]"

These scenes effectively showcase the swarm's ability to navigate through densely populated and complex environments whilst seeking a target.

The final Unity scene is organised as such: (see Appendix 9.2)

Notes

Within both examples above, one might notice clipping of the drone models with the environment.

For this simulation, the Boids have a defined size (conceptualised as a sphere), with the drone model's size not bound to this.

The assets that I have used for these scenes also may not have precise collision meshes, causing some overlap between the models.

I assure the reader that these overlaps are not errors within the implementation and that, programmatically, the Boids **are** avoiding the objects in the environment, however, the models sourced may not integrate effectively together.

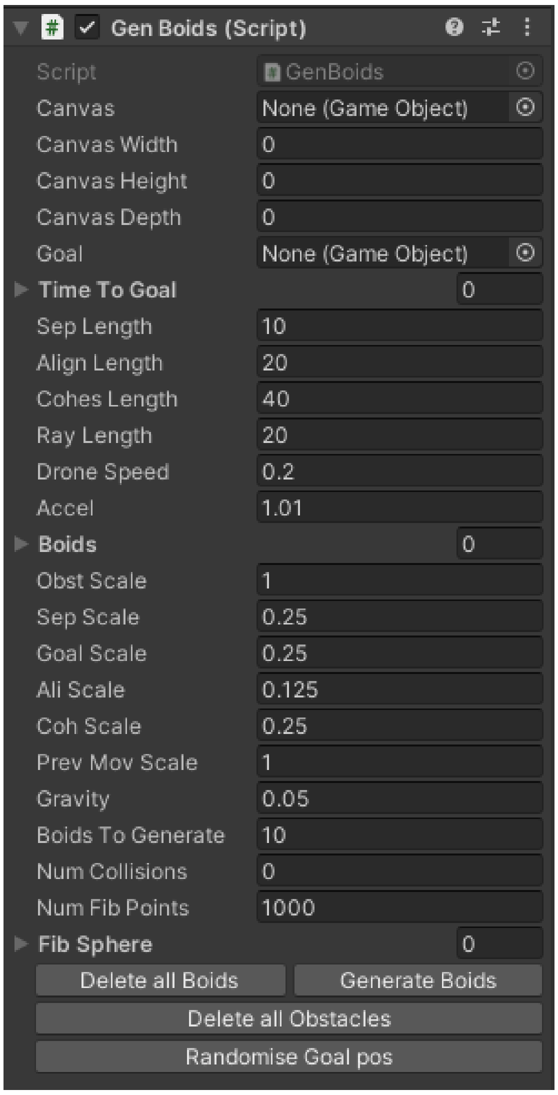
If reverted to simple spheres, one would notice no overlaps, however, to increase the simulation's realism, the inclusion of drone models and complex environments seems necessary.

5 – Testing & Parameter Tuning

Throughout the project, I have been required to tune a wide variety of parameters while testing different implementations.

To do this, I must first define some testing mediums / metrics, so that I can effectively evaluate the performance of an implementation.

Pictured below is the inspector window for the simulation. This window allows me to adjust parameters and gather metrics for the simulation in real time.


Gen Boids (Script)

Script	GenBoids
Canvas	None (Game Object)
Canvas Width	0
Canvas Height	0
Canvas Depth	0
Goal	None (Game Object)
Time To Goal	0
Sep Length	10
Align Length	20
Cohes Length	40
Ray Length	20
Drone Speed	0.2
Accel	1.01
Boids	0
Obst Scale	1
Sep Scale	0.25
Goal Scale	0.25
Ali Scale	0.125
Coh Scale	0.25
Prev Mov Scale	1
Gravity	0.05
Boids To Generate	10
Num Collisions	0
Num Fib Points	1000
Fib Sphere	0

5.1 - Testing Metrics

I have implemented the continuous gathering of testing metrics to provide values in which I can evaluate the performance of the simulation.

Metrics:

- Time to Goal
  - ⇒ List containing times taken to reach each goal
- Percent Flocked
  - ⇒ Percentage of Boids that are "flocked" (updated live)
- Time to All Flock
  - ⇒ Time taken for all Boids to be "flocked"

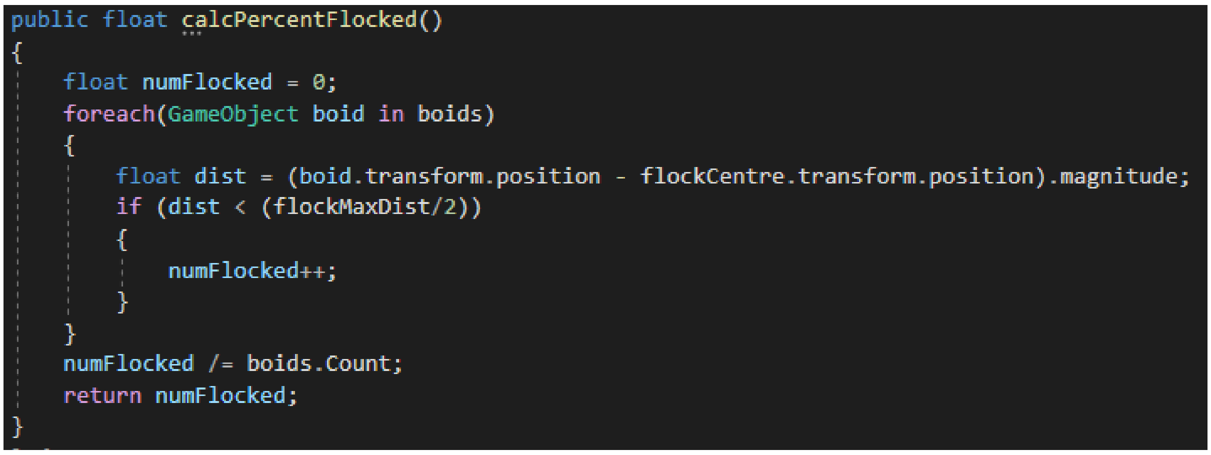
Defining Flocked

For the flocking metrics, I must define what is means to be "flocked".

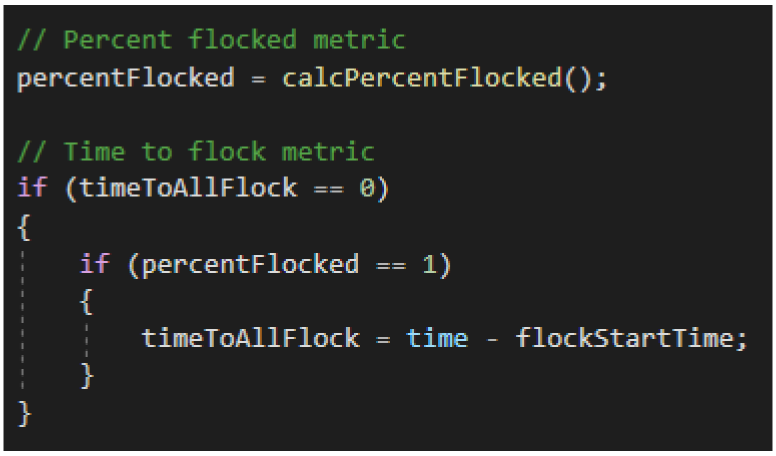
In my implementation, I have defined a flocking radius (which can be adjusted via the inspector window) which is continually positioned at the centre of mass (average position) of all the Boids.

The Boids that are within this radius of the centre of mass are considered “flocked”.

The number of Boids that are within this radius can be calculated, and thus the percentage flocked and time to flock metrics can be trivially calculated:



```
public float calcPercentFlocked()
{
    float numFlocked = 0;
    foreach(GameObject boid in boids)
    {
        float dist = (boid.transform.position - flockCentre.transform.position).magnitude;
        if (dist < (flockMaxDist/2))
        {
            numFlocked++;
        }
    }
    numFlocked /= boids.Count;
    return numFlocked;
}
```



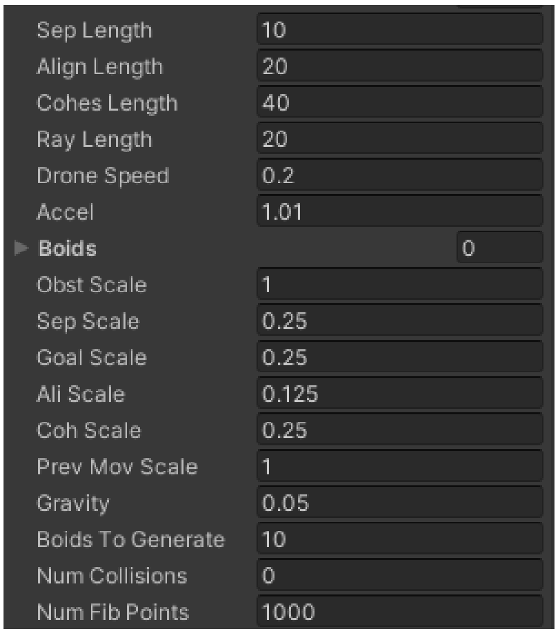
```
// Percent flocked metric
percentFlocked = calcPercentFlocked();

// Time to flock metric
if (timeToAllFlock == 0)
{
    if (percentFlocked == 1)
    {
        timeToAllFlock = time - flockStartTime;
    }
}
```

5.2 - Parameters

There are many parameters that need tuning for this project, wherein small adjustments could have a large impact on the performance of the simulation. The parameter tuning portion of this project holds great significance to the overall outcome and has taken a large portion amount of time, with respect to design and implementation.

Listed Below are the parameters for this simulation:



Sep Length	10
Align Length	20
Cohes Length	40
Ray Length	20
Drone Speed	0.2
Accel	1.01
▶ **Boids**	0
Obst Scale	1
Sep Scale	0.25
Goal Scale	0.25
Ali Scale	0.125
Coh Scale	0.25
Prev Mov Scale	1
Gravity	0.05
Boids To Generate	10
Num Collisions	0
Num Fib Points	1000

Lengths
The length parameters control the Boid’s detection radii, for the SI rules, as well as Obstacle Avoidance.

Scales
The scale parameters control the individual scale factors applied to each oracle, as well as additional oracles and forces.

Misc.
The remaining parameters control various other aspects of the simulation, for example, the maximum speed of each Boid, the number of Boids to be generated, as well as the number of rays generated for Ray Casting.

5.3 – Testing
As we tune the parameters at runtime, we can examine the simulation as well as the metrics to determine the performance of that set of parameters.

We can see with larger radii for Separation, Alignment, and Cohesion, these rules have noticeably different effects on the composition of the swarm.

Separation Radius
As we adjust the separation radius, we can see the swarm immediately react by spreading out / bunching together with respect to the radius value.

This can be seen in the example below:

See “Videos/17 – Separation Radius Tuning.mp4”

Alignment Radius
As we adjust the alignment radius, we can see how the Boids’ alignment vision affects the swarm composition.

With a larger alignment vision, the swarm appears to be much more uniform in heading, in comparison to a smaller alignment vision.

This can be seen in the example below:

See "Videos/18 – Alignment Radius Tuning.mp4"

Cohesion Radius

As we adjust the cohesion radius, we can see how the Boids' cohesion vision affects the swarm composition.

With a smaller cohesion vision, the swarm appears to split into sub swarms at a much higher rate, in comparison to a larger cohesion vison.

A larger cohesion vision leads to a flocked swarm in considerably less time when compared to a small cohesion vision.

This can be seen in the example below:

See "Videos/19 – Cohesion Radius Tuning.mp4"

Weightings / Scales

By adjusting the weighting of each oracle, we can see a considerable change in the behaviour of the swarm, with lower goal scaling, for example, resulting in less of a "urge" to reach the goal and instead fulfil the other rules.

Separation Scale

Adjusting the separation scale changes how much each Boid values avoiding collisions with one another. This value must be fine tuned to minimise collisions, while also to not overrule other oracles. With a low separation scale, one can see a considerably higher number of collisions within the swarm, in comparison to more balanced value. A high separation scale results in erratic behaviour within the swarm as other rules are overruled and ignored as the Boids desperately try to avoid collisions with one another.

These behaviours can be seen in the example below:

See "Videos/20 – Separation Scale Tuning.mp4"

Goal Scale

Adjusting the goal scale is particularly interesting, as with lower values one can see the Boids slowly bank towards the goal and instead put much more attention into swarm behaviours. With a low goal scale, one also sees that "stragglers" are more focused on the goal than the swarm, as they do not have vision on the swarm, so are primarily concerned with the goal. As seen previously, a high scale results in other rules being ignored, resulting in lots of collisions, as well as limited flocking behaviour.

These behaviours can be seen in the example below:

See "Videos/21 – Goal Scale Tuning.mp4"

Another interesting behaviour occurs when we set the goal scale to a negative value. This converts the goal into a repelling entity, performing a similar role as the shepherding agent that (Deng, et al., 2022) discuss.

Poor Tuning

In the example below, one can see an example of a poor parameter set. In this example, one can see extremely limited flocking behaviour, many collisions within the swarm, as well as limited goal acquisition.

See "Videos/22 – Poor Tuning.mp4"

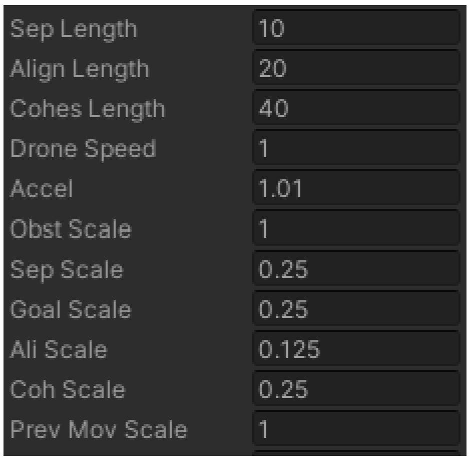
In the following example, one can see that this parameter set is much more suited to a natural swarm simulation, with much more natural behaviours being exhibited. This parameter set would be much better suited to a natural flock simulation, a flock a birds for example.

See "Videos/23 – Natural Tuning.mp4"

These examples clearly show the importance of this stage of the project. Finding an optimal paramter set for my use-case is key to overall success of the project.

5.4 - Testing Conclusion

After thorough testing and parameter tuning, I have arrived upon the parameter set below:



Sep Length	10
Align Length	20
Cohes Length	40
Drone Speed	1
Accel	1.01
Obst Scale	1
Sep Scale	0.25
Goal Scale	0.25
Ali Scale	0.125
Coh Scale	0.25
Prev Mov Scale	1

This parameter set yields what I consider to be the best for my simulation, as it performs the best with regards to the testing metrics, furthermore with clear flocking behaviour and minimal collisions.

This parameter set also performs optimally in 3 dimensions, as all other factors are kept as similar as possible between simulations.

6 – Project Evaluation & Conclusion

One may argue that as this project has limited testing metrics, evaluation criteria are difficult to define numerically. I would argue that as this project is a visual simulation this project is instead much better suited to be evaluated visually.

Visual examination of the final simulations yields success for most of the aims of this project:

- The Boids flock effectively with minimal "stragglers".
- The Boids consistently avoid collisions with one another as well as collisions with the environment.
- The individual Boids have robust intelligence that allows them to make effective decisions.
- Overall computation is kept to a minimum, reducing costs in a real-world example

- The Boids maintain flocking and avoidance whilst reaching the goals (goal object)

From an outside perspective, the Boids appear to flock very effectively, aligning and cohering while avoiding collisions within the swarm, as well as very effectively avoiding collisions with objects, even in complex environments, all while reaching a moving goal.

The project sees success with visual evaluation, however perhaps lacking in numerical evaluation criteria.

From the testing metrics that we do have, we can see that after thorough parameter tuning the Boids not only reach a flocked state quickly, but also maintain a high percentage of this flocked state as the simulation progresses.

7 – Extensions to the Project

If I had further time for this project, I would explore further technologies that could be incorporated into these simulations, as well as adapting these simulations to solve more complex tasks.

One technology that would be interesting to explore would be parameter tuning via Generational Learning. One could replace the manual stage of parameter tuning with a Generational Learning model, which adapts itself based on the testing metrics that have been defined. This could result in a more efficient set of parameters, especially when we consider multiple different environments / tasks.

Another adaptation for this project could be to build a system to simulate drone displays (aerial art displays using drones to display each “pixel”). This could be achieved by creating a master node that assigns different goals (targets) to each Boid based on their relative position to the desired shape. This would update live to accommodate for drone failures and weather events, as well as to allow for animated displays.

8 – References

Chipade, V. S. & Panagou, D., 2019. *Herding an Adversarial Attacker to a Safe Area for Defending Safety-Critical Infrastructure,* s.l.: s.n.

Deng, Y., Ogura, M. & Naoki, W., 2022. *Shepherding Control for Separating a Single Agent from a Swarm.* s.l.:s.n.

Deserno, M., 2004. *How to generate equidistributed points on the surface of a sphere,* s.l.: s.n.

Drost, C., 2017. *Evenly distributing n points on a sphere - The golden spiral method,* s.l.: s.n.

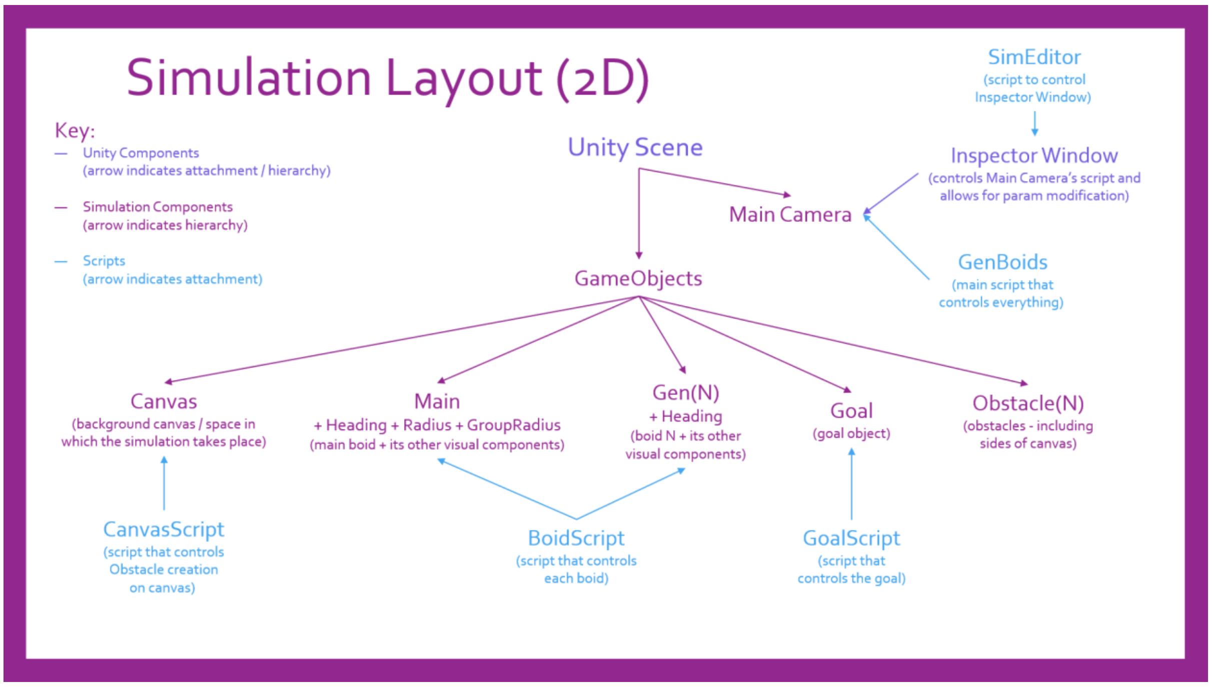
Lague, S., 2019. *Coding Adventure: Boids,* s.l.: s.n.

Parker, C., 2007. *Boids,* s.l.: Stanford University.

Reynolds, C., 1987. *Flocks, Herds, and Schools: A Distributed Behavioral Model,* s.l.: s.n.

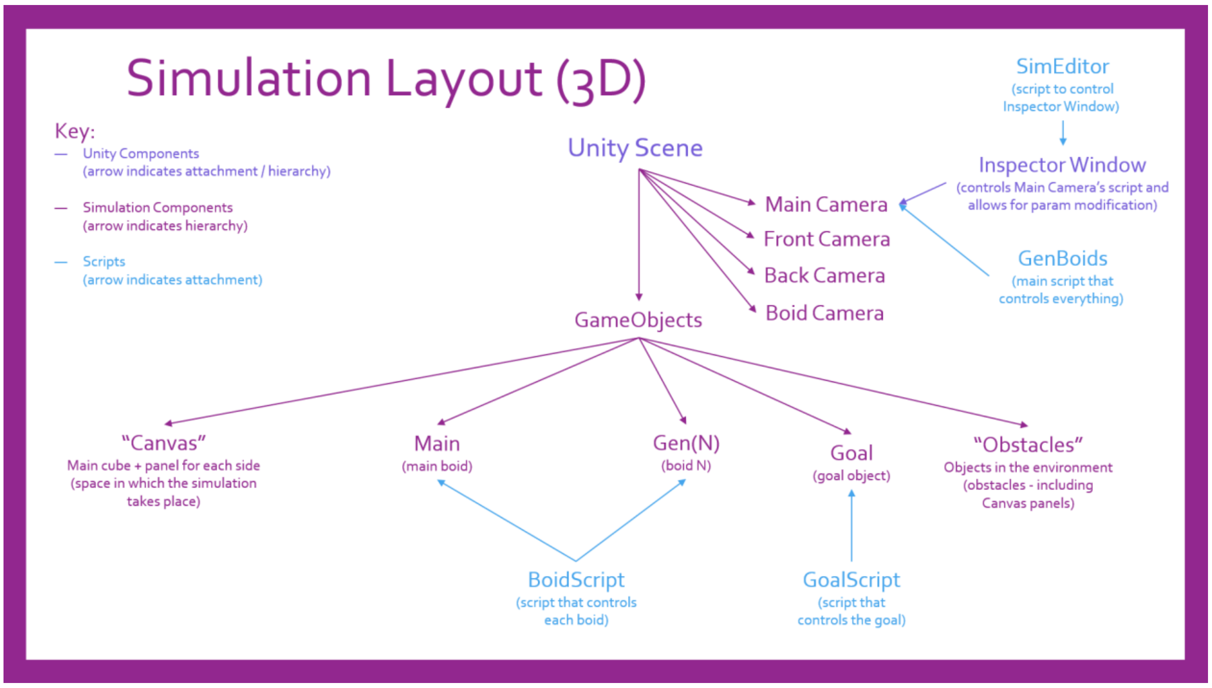
9 – Appendix

9.1 – Unity Scene Layout (2D)



Simulation Layout (2D)
Key:
Unity Components (arrow indicates attachment / hierarchy)
Simulation Components (arrow indicates hierarchy)
Scripts (arrow indicates attachment)
Unity Scene
Main Camera
GameObjects
SimEditor (script to control Inspector Window)
Inspector Window (controls Main Camera's script and allows for param modification)
GenBoids (main script that controls everything)
Canvas (background canvas / space in which the simulation takes place)
Main + Heading + Radius + GroupRadius (main boid + its other visual components)
Gen(N) + Heading (boid N + its other visual components)
Goal (goal object)
Obstacle(N) (obstacles - including sides of canvas)
CanvasScript (script that controls Obstacle creation on canvas)
BoidScript (script that controls each boid)
GoalScript (script that controls the goal)

9.2 – Unity Scene Layout (3D)



Simulation Layout (3D)
Key:
Unity Components (arrow indicates attachment / hierarchy)
Simulation Components (arrow indicates hierarchy)
Scripts (arrow indicates attachment)
Unity Scene
Main Camera
Front Camera
Back Camera
Boid Camera
GameObjects
SimEditor (script to control Inspector Window)
Inspector Window (controls Main Camera's script and allows for param modification)
GenBoids (main script that controls everything)
"Canvas" Main cube + panel for each side (space in which the simulation takes place)
Main (main boid)
Gen(N) (boid N)
Goal (goal object)
"Obstacles" Objects in the environment (obstacles - including Canvas panels)
BoidScript (script that controls each boid)
GoalScript (script that controls the goal)

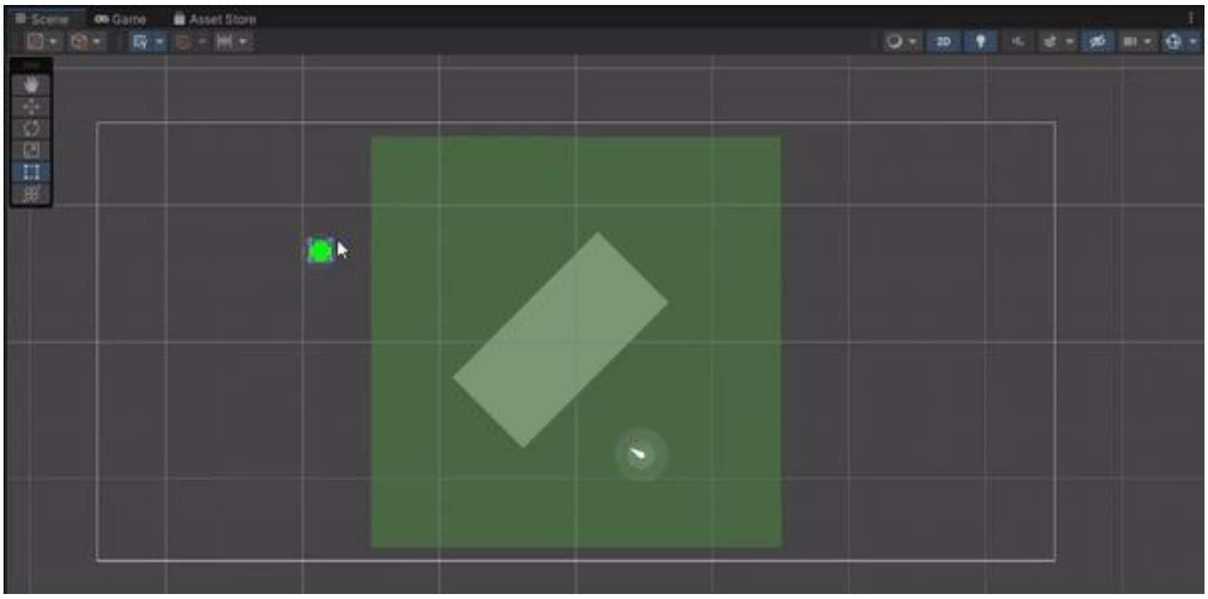
9.3 – Bugs and Oversights

Throughout development, I ran into a number of bugs and oversights, most notably:

- The Boid would occasionally pass right through the object
- The Boid would occasionally pass through the corners of the object

9.3.1 – Bug 1

Below you can see the first bug, as the Boid passes through the centre of the object:



The first bug is due to the rule combination scheme that was used at that time: Scheme 2: Average (see 4.3)

The limitations of this scheme become clear in this case: Obstacle Avoidance suggestions are combined with inverse suggestions from the other oracles, resulting in overruling of OA, thus resulting in OA to be ignored.

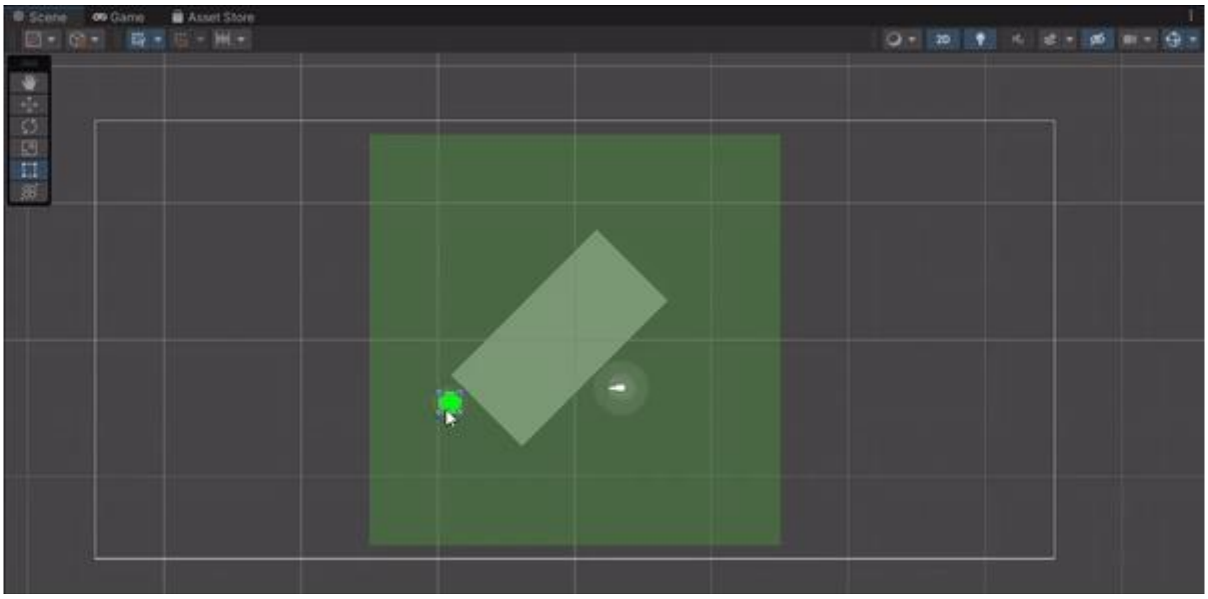
More advanced recombination schemes fix this issue, even the small upgrade to Scheme 3: Weighted Average.

This would allow us to skew the decision heavily in favour of the Collision Avoidance algorithm's recommendation of direction.

Schemes 4 and 5 would be best suited to remedy this bug as they would allow for OA to take priority when obstacle collisions are incoming.

9.3.2 – Bug 2

Below you can see the second bug, as the Boid passes through the corners of the object:



The second bug is due to the scheme in which collisions were being detected.

In this scheme, we were looping through each edge of the object, and checking whether the Boid's front ray intersects, and if so, beginning the Collision Avoidance algorithm.

This is a poor implementation as when the Boid is heading towards a corner of the object, its front ray will intersect two edges, and the detection loop will proceed with Collision Avoidance for the first edge in the loop that intersects.

Note – This bug also applies to opposite edges, if the object is thin enough such that the front ray can intersect both edges

We instead needed to modify our detection scheme such that either the **closest edge** is chosen to begin the Collision Avoidance algorithm, or we instead for each ray cast check if the ray collides with **any** obstacle (this method is used in the 3D implementation).

Another fix for this issue was to simplify the obstacles to lines. This simplification allowed for simpler OA as well as allowing for the creation of obstacles at runtime.