Flocking Algorithm Documentation Created by Charlie Frost

Before I begin, it is worth noting that this project was created through following the amazing channel 'Board To Bits Games' - https://www.youtube.com/channel/UCifiUB82IZ6kCkjNXN8dwsQ. I thought I'd follow a Youtube series since I have never delved into this area of AI before. This documentation will be split into three parts – a planning section, a blog-like section and a review/reflection section. The planning section will detail my strategy of attacking the project, including any necessary scribbles/diagrams that may aid me in the creation process. The blogging section will be a simplified step-by-step discussion about how I am creating the Flocking Algorithm and how it is going, along with any issues I run into etc. Finally, the reflection section will be exactly that; I will be talking about how I have found the project, how I would improve it etc.

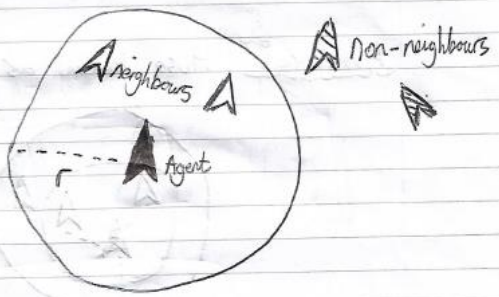
Planning Section

As all good developers do, I have completed my planning process using a pen and paper. This allowed me to scribble and doodle hastily and produce a plan relatively quicker than if I attempted the same process on a computer. It also helped me visualise how the project will (hopefully) come together! Here is the plan:

Flocking Algorithm – General Plan/Ideas

I'm going to:
- Create flocking algorithm
- Make the algorithm modular
- Create room for expansion

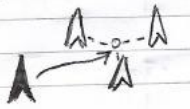
Elements of flocking:
- Cohesion
- Alignment
- Avoidance



Cohesion: Cohesion will encourage each agent to stay within its flock with its neighbour.

↓

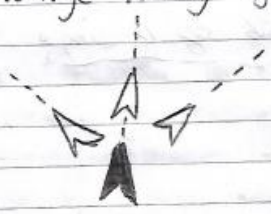
I will achieve this by aiming each agent towards the mid-point of all of its neighbours:



Alignment: Alignment will encourage each agent in a flock to move in a common direction.

↓

I will achieve this by setting each agent's heading to the average heading of all of their neighbours.

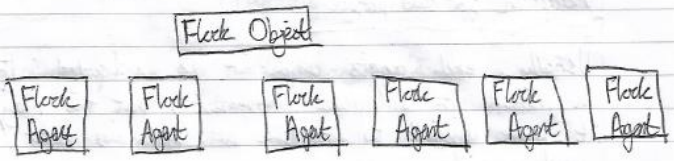


Avoidance: Also known as Separation.

Alignment is another radius, smaller than the neighbour radius, that will ensure that agents do not collide with their neighbours.

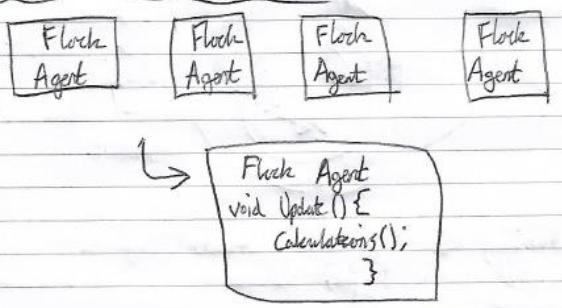
Combining all 3 of these elements using a weighting system devised by me, the agents will each have a destination

Flock Object Method:



One method (common) is to have a flock object, however in many cases this object is NOT coordinating all of the agents. Rather, the object serves as an iterator for each agent to execute its own flocking behaviours. Usually, these behaviours are declared within the object, with each agents info passed in as parameters.

No Flock Object Method:



Using their 'Update' methods, each agent will calculate their movements each frame. The downside to this method is it is more difficult to organise and instanciate.

For these reasons, I shall be using a Flock Object.

Adaption for this project:

Usually, each agents behaviour is hardcoded together in a method to calculate movement, but this may make it more difficult to introduce new behaviours later in a real project.

Therefore, I will be implementing each basic behaviour in its own Scriptable object. Additionally, a composite object that will be able to group multiple behaviours using different weights. This will enable me to change behaviours, store variations of them or apply different behaviours to multiple flocks.

Blogging Section

This section will be somewhat in note form as this is a fairly quick project and I just wanted to detail the journey from start to finish. Because of this reason, the code will partly be explained through commenting in the scripts themselves and I will point out important points myself. Firstly, I have added in a simple arrow sprite and created three blank scripts – Flock, Flock Agent and Flock Behaviour.

I have made the Flock Behaviour script an abstract class and a scriptable object. This is because I want it to contain an abstract method that the agents are going to implement.

```
public abstract class FlockBehaviour : ScriptableObject
{
    // Abstract method that returns a Vector2
    // It will take a FlockAgent called agent, a list of transforms called context and the flock itself
    public abstract Vector2 CalculateMove(FlockAgent agent, List<Transforms> context, Flock flock);
}
```

Next, I worked on my Flock Agent script:

```
[RequireComponent(typeof(Collider2D))]

public class FlockAgent : MonoBehaviour
{
    // Set up my COllider2D:
    Collider2D agentCollider;
    // We want to be able to access this collider without assigning to it:
    public Collider2D AgentCollider { get { return agentCollider; } }



    void Start()
    {
        // Find the instance of the Collider on our object
        agentCollider = GetComponent<Collider2D>();
    }



    public void Move(Vector2 velocity)
    {
        // Turn agent to position it's moving to
        // In Unity2D, the 'up' basically means forwards for the arrows
        transform.up = velocity;

        // Move agent to position it's moving to
        // This line ensures constant movement regardless of framerate
        // We cast velocity as a Vector3 so we aren't adding a Vector2 and Vector3 together; this gives errors
        transform.position += (Vector3)velocity * Time.deltaTime;
    }
```

The script requires the Flock Agent to have a collider attached, finds the collider and adds in some code for movement. I have added this onto an empty object called ‘Flock Agent’, which already has the required Collider2D on it. I then added the sprite as a child object and turned down the radius of the collider.

Next up I created lots of variables to use within the Flock script:

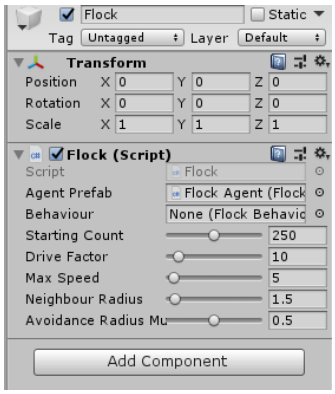
```
public class Flock : MonoBehaviour
{
    // Variables (lots of)
    // This line lets us manually add in our prefabs
    public FlockAgent agentPrefab;
    // Create a new list called agents that will add each prefab to the list
    List<FlockAgent> agents = new List<FlockAgent>();
    // This is where we will put in the scriptable object we made
    public FlockBehaviour behaviour;

    // We will use these ranges to create a slider for the amount of agents within a flock (some other sliders too)
    [Range(1, 500)]
    public int startingCount = 250;
    // The size of our flock circle will depend on how many agents are in the flock
    // This float will let me work out the overall flock radius:
    const float AgentDensity = 0.08f;

    // I have created another variable which we will use to remove any counteracting movements within the flock
    // Without this variable the flock will move very slowly
    [Range(1f, 100f)]
    public float driveFactor = 10f;

    // This float will cap the speed of the flock
    [Range(1f, 100f)]
    public float maxSpeed = 5f;

    // Now I will create two radiuses
    // This one will determine the radius for each agent's neighbours
    [Range(1f, 10f)]
    public float neighbourRadius = 1.5f;

    // This one will be a multiplier, used to work out our avoidance (see plan)
    [Range(0f, 1f)]
    public float avoidanceRadiusMultiplier = 0.5f;


    // The next variables will not be set, but will be used to calculate things
    float squareMaxSpeed;
    float squareNeighbourRadius;
    float squareAvoidanceRadius;
    public float SquareAvoidanceRadius { get { return squareAvoidanceRadius} }
```

The final variables are clearly not calculated yet, and I will do that next in my Start method. The reason for this is that usually when comparing things such as current velocity with maxSpeed, we use a vector's magnitude. However, if you get the magnitude itself, you will have to do some square rooting. This is quite complex and can be taxing for the computer itself to deal with. Therefore, rather than finding the roots every single time, we will just compare the roots with each other, saving a step when it comes to the maths.

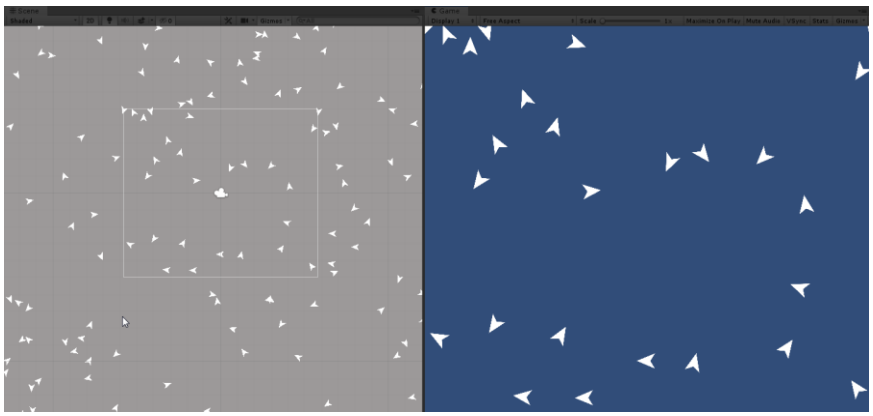
Below is the start method for the Flock script. It completes our maths explained above and then instantiates the flock. It determines where the object will spawn and it's rotation using some of our predefined variables, all from an empty GameObject called 'Flock'. We then give each agent a name and add them to our agents list.

```
void Start()
{
    // The next section is explained in the blogging section of my documentation
    squareMaxSpeed = maxSpeed * maxSpeed;
    squareNeighbourRadius = neighbourRadius * neighbourRadius;
    squareAvoidanceRadius = squareNeighbourRadius * avoidanceRadiusMultiplier * avoidanceRadiusMultiplier;

    // Instanciate our flock:
    for (int i = 0; i < startingCount; i++)
    {
        FlockAgent newAgent = Instantiate(
            // This is the object we will be instanciating:
            agentPrefab,
            // Set the spawn point for our agents:
            // We use the AgentDensity constant so that they are always similarly placed - no huge gaps
            Random.insideUnitCircle * startingCount * AgentDensity,
            // Now we set the rotation (on the z axis) of each agent
            // It requires a quaternion, so I am going to create a random Vector3 between 0 and 360 and then convert it to a quaternion
            Quaternion.Euler(Vector3.forward * Random.Range(0f, 360f)),
            // Set the parent of the agent - the flock itself's transform
            transform
            );
        // Give the agents a name so we can keep track of them more easily
        newAgent.name = "Agent " + i;
        // Now add each agent to our list of agents
        agents.Add(newAgent);
    }
}
```

Here is what our Flock Object looks like with our Flock script added onto it, after dragging our prefab:



After altering the scene view background so that I can clearly see the agents, here is what happens when I click play:



As expected, the agents all spawn as desired in a random manner, clearly some variables need tweaking to create a more flock-like flock and they do not move because I have not given them any behaviours yet, but it's a great start!

Next up, I am going to help the flock iterate through each of it's agents, and apply some behaviours based on the context of their neighbours:

```
void Update()
{
    foreach(FlockAgent agent in agents)
    {
        // Create a list called context, this will deal with things in context to our neighbour radius
        // This list looks at nearby objects using the agent we are currently looking at
        List<Transform> context = GetNearbyObjects(agent);

    }
}

List<Transform> GetNearbyObjects(FlockAgent agent)
{
    // We create a new list of transforms called context
    List<Transform> context = new List<Transform>();
    // We create a circle in space that gets ahold of all colliders within it, hence why we are using our neighbourRadius
    // These colliders will be placed within an array called contextColliders
    Collider2D[] contextColliders = Physics2D.OverlapCircleAll(agent.transform.position, neighbourRadius);
    // For each collider                                    he agent in question, we want to add them to our list
    foreach(Collider2D c
    {
        if (c != agent.AgentCollider)
        {
            context.Add(c.transform);
        }
    }
    // Update our context list
    return context;
}
```

(local variable) Collider2D[] contextColliders

The code above is entirely explained in the commenting, but I have basically created a list which stores neighbours called context, which uses an array of Collider2Ds to find the neighbouring agent's colliders.

Next is some code that may be needed to alter the agent's speed. It speeds them up using our driverFactor variable and then checks they are not exceeding the maxSpeed variable. If they are, their magnitude will be reset to 1, and then multiplied to our maxSpeed value. This ensures no jittering created by speed issues:

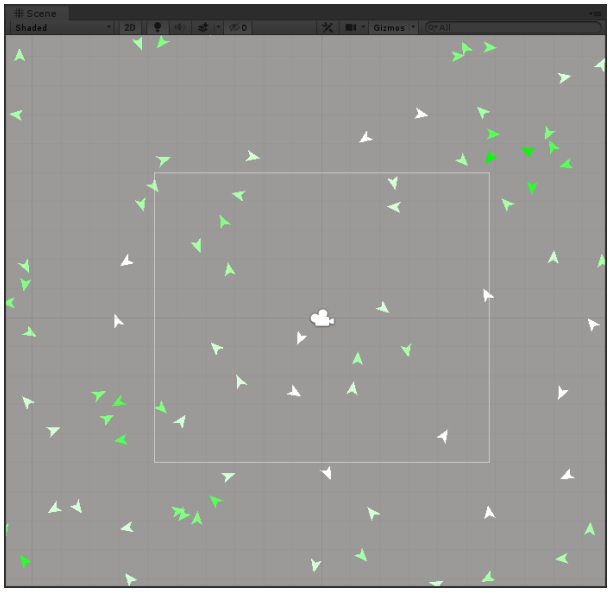
```
    // Use the nearby objects - takes in our agent, the list of neighbours and the flock (this) as parameters
    // This returns back the way in which the agent should move
    Vector2 move = behaviour.CalculateMove(agent, context, this);
    // Speed the agent up
    move *= driveFactor;
    // Then check we have not exceeded our maxSpeed limit
    // If it has, bring speed back to the maximum
    if (move.sqrMagnitude > squareMaxSpeed)
    {
        // This resets magnitude to 1 then multiplies it's current speed to our maxSpeed
        move = move.normalized * maxSpeed;
    }
}
```

Now for the all-important line, the one which actually moves our agent using the 'move' that we have already calculated.

```
agent.Move(move);
```

Before continuing I want to check that our neighbours are being counted properly. To do this I have commented out some code and then added in a line that will alter the agent's colour to appear more green if they have more neighbours. Here is the code and the results:

```
// For demo, checks how many neighbours the agent has, the more neighbours, the greener the agent will appear (from white)
agent.GetComponentInChildren<SpriteRenderer>().color = Color.Lerp(Color.white, Color.green, context.Count / 6f);
```



This just confirms that the neighbour information that I was passing to each agent has worked as intended which is great news. Next, I will be adding some behaviours so that the flocks can move using the code I have just implemented.

As I have just stated I will now be implementing perhaps the most important part of the project, the three key behaviours – cohesion, alignment and avoidance (see plans). To begin, I have created two new folders outside of the scripts folder – Behaviour Scripts and Behaviour Objects. Behaviour Scripts will contain my C# scripts which will inherit from my scriptable object and my Flock Behaviour script, while Behaviour Objects will contain the instances of my scriptable object. Next, inside the Behaviour Scripts folder I have created three new scripts – CohesionBehaviour, AlignmentBehaviour and AvoidanceBehaviour.

First, I will work on the CohesionBehaviour script:

```
// Because it is a scriptable object we need a way to create it, so we use an attribute:
[CreateAssetMenu(menuName = "Flock/Behaviour/Cohesion")]
public class CohesionBehaviour : FlockBehaviour
{
    public override Vector2 CalculateMove(FlockAgent agent, List<Transform> context, Flock flock)
    {
        throw new System.NotImplementedException();
    }
```

As you can see, I have inherited from the abstract FlockBehaviour scriptable object class. I have also created an attribute, so we have a way to create the scriptable objects. I afterwards deleted the throw exception line as it was not necessary.

```
[CreateAssetMenu(menuName = "Flock/Behaviour/Cohesion")]
public class CohesionBehaviour : FlockBehaviour
{
    public override Vector2 CalculateMove(FlockAgent agent, List<Transform> context, Flock flock)
    {
        // If no neighbours, return no adjustment
        if (context.Count == 0)
            return Vector2.zero;

        // Add all points together and find the average point
        Vector2 cohesionMove = Vector2.zero;
        // We go through each items transform in our list of neighbours
        foreach (Transform item in context)
        {
            cohesionMove += (Vector2)item.position;
        }
        // We now average the Vector out again so it is not a huuuge number
        cehesionMove /= context.Count;

        // Change from global position to offset of the agent itself
        // Create offset from agent position:
        cohesionMove -= (Vector2)agent.transform.position;
        // Then we can return the vector
        return cohesionMove;
    }
```

Now I have added some code to the CalculateMove method. We simply find the average Vector2 of all of our neighbours by using the context list and return it.

I then created a scriptable object ‘Cohesion’ in my Behaviour Objects folder and dragged it into the Behaviour slot on my Flock Game Object. I also (somehow) remembered to uncomment my movement code from last session. From the following video, you can see that the Cohesion Behaviour is working properly, as the agents are quickly converging on their neighbour’s position. Due to the fact there are no other behaviours in place, it does look quite strange, but it definitely works!

https://twitter.com/charliegfrost/status/1195384002737254401

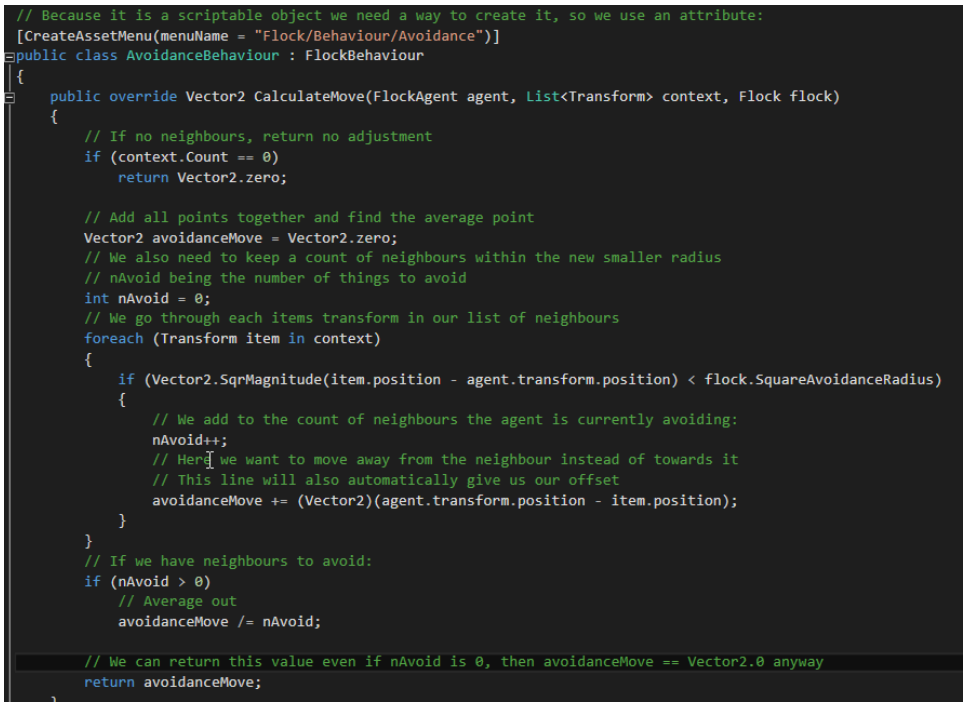
Next, I am setting up the alignment behaviour. It was fairly simple as most of the code is copied from the cohesion behaviour:

```
// Because it is a scriptable object we need a way to create it, so we use an attribute:
[CreateAssetMenu(menuName = "Flock/Behaviour/Alignment")]
public class AlignmentBehaviour : FlockBehaviour
{
    public override Vector2 CalculateMove(FlockAgent agent, List<Transform> context, Flock flock)
    {
        // If no neighbours, maintain current alignment
        if (context.Count == 0)
            return agent.transform.up;

        // Add all points together and find the average point
        Vector2 alignmentMove = Vector2.zero;
        // We go through each items transform in our list of neighbours
        foreach (Transform item in context)
        {
            alignmentMove += (Vector2)item.transform.up;
        }
        // We now average the Vector out again so it is not a huuuge number
        alignmentMove /= context.Count;

        return alignmentMove;
    }
```

The differences are changing any cohesions to alignment, any transform.positions to transform.ups and removing the code for creating an offset. This is because it essentially performs the same calculations – it takes each neighbour’s transform.up (direction) and

finds an average. As they are not moving yet, there is nothing to show for this script at the moment.

As you probably expected, it is now time for the Avoidance script:



```
// Because it is a scriptable object we need a way to create it, so we use an attribute:
[CreateAssetMenu(menuName = "Flock/Behaviour/Avoidance")]
public class AvoidanceBehaviour : FlockBehaviour
{
    public override Vector2 CalculateMove(FlockAgent agent, List<Transform> context, Flock flock)
    {
        // If no neighbours, return no adjustment
        if (context.Count == 0)
            return Vector2.zero;

        // Add all points together and find the average point
        Vector2 avoidanceMove = Vector2.zero;
        // We also need to keep a count of neighbours within the new smaller radius
        // nAvoid being the number of things to avoid
        int nAvoid = 0;
        // We go through each items transform in our list of neighbours
        foreach (Transform item in context)
        {
            if (Vector2.SqrMagnitude(item.position - agent.transform.position) < flock.SquareAvoidanceRadius)
            {
                // We add to the count of neighbours the agent is currently avoiding:
                nAvoid++;
                // Here we want to move away from the neighbour instead of towards it
                // This line will also automatically give us our offset
                avoidanceMove += (Vector2)(agent.transform.position - item.position);
            }
        }
        // If we have neighbours to avoid:
        if (nAvoid > 0)
            // Average out
            avoidanceMove /= nAvoid;

        // We can return this value even if nAvoid is 0, then avoidanceMove == Vector2.0 anyway
        return avoidanceMove;
    }
```

Again, only a few adjustments, but it is working as you can see in the following video. All I did was add an integer for nAvoid which is number of neighbours that need avoiding. We then add them to the count when necessary and move away from their combined position. The cool thing about this script is that the line after 'nAvoid++;' actually finds the offset for us. We will also do not need to add any more code in the case that nAvoid = 0 and we will not receive an error. This is because if nAvoid = 0 then avoidanceMove == Vector2.zero anyway. Here is the short video of the agents moving away from their hostile neighbours:

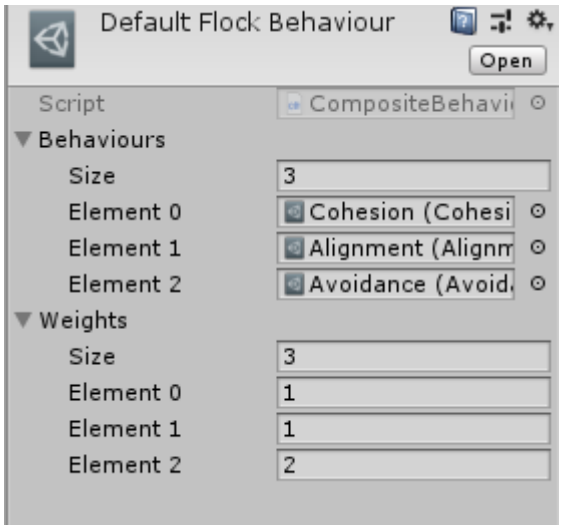
https://twitter.com/charliegfrost/status/1195384725076140032

Now I am going to combine the three behaviours I have just created into one using a weighting system. This system will prioritise which behaviour needs to take place using certain metrics. To do this I have created a script in the Behaviour Scripts folder called CompositeBehaviour. Here it is:

```
// Because it is a scriptable object we need a way to create it, so we use an attribute:
[CreateAssetMenu(menuName = "Flock/Behaviour/Composite")]
public class CompositeBehaviour : FlockBehaviour
{

    // Variables
    // This array will be our behaviours to composite together
    public FlockBehaviour[] behaviours;
    // This array will correlate with the behaviours and is used for weighting
    public float[] weights;

    public override Vector2 CalculateMove(FlockAgent agent, List<Transform> context, Flock flock)
    {
        // Our 2 arrays need to contain the same number of items, so check for that here and debug it
        if (weights.Length != behaviours.Length)
        {
            Debug.LogError("Data mismatch in " + name, this);
            return Vector2.zero;
        }

        // Set up move
        Vector2 move = Vector2.zero;

        // Iterate through behaviours
        // I used a for loop instead of a foreach loop here because behaviours and weights need to be using the same indexes
        for (int i = 0; i < behaviours.Length; i++)
        {
            Vector2 partialMove = behaviours[i].CalculateMove(agent, context, flock) * weights[i];

            // Make sure partialMove is being limited to the extent of the weight
            // If there is some movement being returned:
            if (partialMove != Vector2.zero)
            {
                // Check does the overall movement exceed the weight?
                if (partialMove.sqrMagnitude > weights[i] * weights[i])
                {
                    // Normalise it back to a magnitude of 1, then multiply it by the weight
                    // This means it will be set at the maximum of the weight
                    partialMove.Normalize();
                    partialMove *= weights[i];
                }

                move += partialMove;
            }
        }

        return move;
    }
```

It uses two arrays to give each behaviour we have a weight. We then perform a few checks such as checking there is some sort of movement being returned by our calculateMove method. Finally, some normalising is performed if necessary.

After this I created a new scriptable object called Default Flock Behaviours and altered the arrays to look like this:



Here was the achieved result:

https://twitter.com/charliegfrost/status/1195397032615251971

As you can see, each behaviour is working as intended but we are left with some kind of flickering. This is the Cohesion behaviour somewhat taking over. As the agents are moving around, they are constantly trying to remain at their 'middle point' which means they are swapping directions rapidly and is the cause of the flickering.

To fix this, I have created an updated version of the Cohesion Behaviour script, called Steered Cohesion Behaviour. There are only a few small differences in code as you can see:

```
// Because it is a scriptable object we need a way to create it, so we use an attribute:
[CreateAssetMenu(menuName = "Flock/Behaviour/SteeredCohesion")]
public class SteeredCohesionBehaviour : FlockBehaviour
{
    // Variables
    // Smoothdamp needs this variable:
    Vector2 currentVelocity;
    // How long it should take for the agent to get from it's current state to the calculated state
    // The higher the float, the less direction changes will be apparent
    public float agentSmoothTime = 0.5f;


    public override Vector2 CalculateMove(FlockAgent agent, List<Transform> context, Flock flock)
    {
        // If no neighbours, return no adjustment
        if (context.Count == 0)
            return Vector2.zero;

        // Add all points together and find the average point
        Vector2 cohesionMove = Vector2.zero;
        // We go through each items transform in our list of neighbours
        foreach (Transform item in context)
        {
            cohesionMove += (Vector2)item.position;
        }
        // We now                                    it is not a huuuge number
        cohesionMove /= context.Count;

        // Change from global position to offset of the agent itself
        // Create offset from agent position:
        cohesionMove -= (Vector2)agent.transform.position;

        cohesionMove = Vector2.SmoothDamp(agent.transform.up, cohesionMove, ref currentVelocity, agentSmoothTime);
        // Then we can return the vector
        return cohesionMove;
    }
```

(local variable) Vector2 cohesionMove

Firstly, there is a new Vector2 called currentVelocity. This was created because the SmoothDamp function needs it as a parameter. Next was a new public float called agentSmoothTime. Again, this will be used as a parameter for how fast I want the smoothing process to occur, offering more or less prioritisation when it comes to choosing a destination (between goal point and middle point of flock). I made this variable public, so it is easy to alter in the Unity Editor. The last bit of new code is the second to last one. This is our smoothing process. Here is a video of the improved movement of our agents:

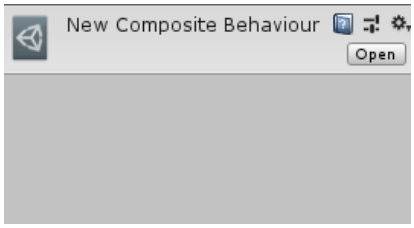
https://twitter.com/charliegfrost/status/1195407611551727616

Next up I am going to work on a custom editor for my Composite Behaviour script. This will allow for more customisation and make everything a little bit easier to handle. Another reason is simply for my personal benefit – I want more practice working with the custom editor tools that unity provides.

I started by creating an 'Editor' folder in my assets folder. Then I created a 'CompositeBehaviourEditor' script inside of that folder. To set up the script, I created a 'Setup' region and added the UnityEditor namespace. Within the setup section, I cast target so we can access 'CompositeBehaviour's variables etc. and then created a rectangle. I also added a warning for if there are no behaviours in the array or no behaviour is being undertaken:

```
using UnityEngine;
// Add this namespace to give access to the custom editor attribute
using UnityEditor;

[CustomEditor(typeof(CompositeBehaviour))]
public class CompositeBehaviourEditor : Editor
{
    public override void OnInspectorGUI()
    {
        #region Setup
        // Here we cast target, this is an object (in the inspector), directly to composite behaviour so we can access it's variables etc.
        CompositeBehaviour cb = (CompositeBehaviour)target;

        // Gives us a rectangle in the inspector
        Rect r = EditorGUILayout.BeginHorizontal();
        // Change it's height to the height of text
        r.height = EditorGUIUtility.singleLineHeight;
        #endregion

        // Check for behaviours
        // If there are no behaviours or the array is empty
        if (cb.behaviours == null || cb.behaviours.Length == 0)
        {
            EditorGUILayout.HelpBox("No behaviours in array.", MessageType.Warning);
        }
    }
}
```

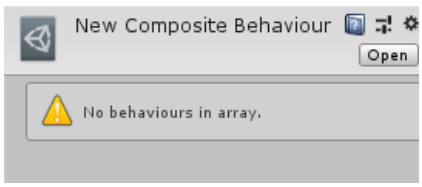
As you can see, the warning is not working (I created a new composite behaviour object):



I then did some research and found out that this method (rect) was outdated and no longer works for most of the 2019 versions of Unity. Now we do not set the x- or y-coordinates, neither the height or width. This means no rect is needed whatsoever. After doing some more searching I found the updated versions and added them in (ignore commenting, updated after):

```
// Gives us a rectangle in the inspector
EditorGUILayout.BeginHorizontal();
// Change it's height to the height of text
EditorGUILayout.EndHorizontal();
#endregion
```

Now the warning works:



Then I got the bulk of the coding done:

```
else
{
    // If there are behaviours (whole reason of this script!)
    EditorGUILayout.BeginHorizontal();
    // Each one of these simple creates a label field with a min width and max width
    // These two widths mean that even if the inspector window is resized, the text will remain where we it want
    EditorGUILayout.LabelField("Number", GUILayout.MinWidth(60f), GUILayout.MaxWidth(60f));
    EditorGUILayout.LabelField("Behaviours", GUILayout.MinWidth(60f));
    EditorGUILayout.LabelField("Weights", GUILayout.MinWidth(60f), GUILayout.MaxWidth(60f));
    EditorGUILayout.EndHorizontal();

    // Display each behaviour:
    for (int i = 0; i < cb.behaviours.Length; i++)
    {
        EditorGUILayout.BeginHorizontal();
        // Make each item a string
        EditorGUILayout.LabelField(i.ToString(), GUILayout.MinWidth(60f), GUILayout.MaxWidth(60f));
        // Show a field for the behaviours
        // False means we wont take any objects that come from the scene - we wont only scriptable objects
        cb.behaviours[i] = (FlockBehaviour)EditorGUILayout.ObjectField(cb.behaviours[i], typeof(FlockBehaviour), false, GUILayout.MinWidth(60f));
        // Do the same for the weights:
        cb.weights[i] = EditorGUILayout.FloatField(cb.weights[i], GUILayout.MinWidth(60f), GUILayout.MaxWidth(60f));
        EditorGUILayout.EndHorizontal();
```

To begin, we create labelfields for number, behaviour and weights. Then we go through the behaviours array and make each item a string, then create object fields for them. Finally, we create fields for the weights. Here is what the inspector currently looks like:

Success! The biggest looming problem with this custom editor, however is that we have lost functionality of the arrays – we cannot change the size etc. like we could before. To fix this issue, I have created some new buttons. We have one for adding behaviours and one for removing behaviours, as you can see:

```
// Clear inspector
EditorGUILayout.EndHorizontal();

if (OnInspectorGUI.Button("Add Behaviour"))
{
    // Add behaviour



}

// We only want the add button to appear if there are behaviours to remove
if (cb.behaviours != null && cb.behaviours.Length > 0)
{
    if (OnInspectorGUI.Button("Remove Behaviour"))
    {
        // Remove behaviour



    }
```

Now I am going to create the methods that these buttons will be used for. First up is the add behaviour method. Here it is:

```
void AddBehaviour(CompositeBehaviour cb)
{
    // Get original size of the array
    int oldCount = (cb.behaviours != null) ? cb.behaviours.Length : 0;
    // If array is empty we will start with 1 item
    FlockBehaviour[] newBehaviours = new FlockBehaviour[oldCount + 1];
    // Create new array
    float[] newWeights = new float[oldCount + 1];
    // Iterate through and add the original values
    for (int i = 0; i < oldCount; i++)
    {
        newBehaviours[i] = cb.behaviours[i];
        newWeights[i] = cb.weights[i];
    }
    // newWeights can't be zero otherwise new behaviours can't take effect
    // The user would think something has gone wrong if we don't set it to 1f:
    newWeights[oldCount] = 1f;
    cb.behaviours = newBehaviours;
    cb.weights = newWeights;
}
```

This method contains some error-prevention checks that make the experience of using the custom editor far more user friendly. This can be seen in the commenting. Next was the remove behaviour code:

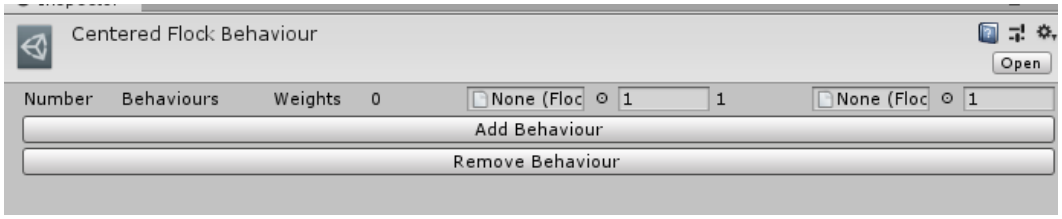
```
void RemoveBehaviour(CompositeBehaviour cb)
{
    // We know the array wont be null, as the button would not appear
    int oldCount = cb.behaviours.Length;

    // If there is only one behaviour, we can just clear it out:
    if (oldCount == 1)
    {
        cb.behaviours = null;
        cb.weights = null;
        return;
    }

    // If array is empty we will start with 1 item
    FlockBehaviour[] newBehaviours = new FlockBehaviour[oldCount - 1];
    // Create new array
    float[] newWeights = new float[oldCount - 1];
    // Iterate through and add the original values
    for (int i = 0; i < oldCount - 1; i++)
    {
        newBehaviours[i] = cb.behaviours[i];
        newWeights[i] = cb.weights[i];
    }
    // We do not need to assign anything to newWeights here because it has already been assigned
    cb.behaviours = newBehaviours;
    cb.weights = newWeights;
}
```

For this method I copied the AddBehaviour code and made some alterations. Mostly numerical changes from +1 to -1 etc. however I have also added in an if loop that will clear out the arrays straight away if they are 1. It is also worth noting that the line ‘newWeights[oldCount] = 1f;’ has been removed, as it has already been assigned the value of 1. I then added the calls for these methods within my button code. After this I added in ‘EditorUtility.SetDirty(cb)’. After either AddBehaviour or RemoveBehaviour has been called I add this line in. This let’s Unity know that the scriptable object has been changed and needs to be saved. Then I used the ‘EndChangeCheck’ method to SetDirty again. This also checks if something has been changed and tells Unity.

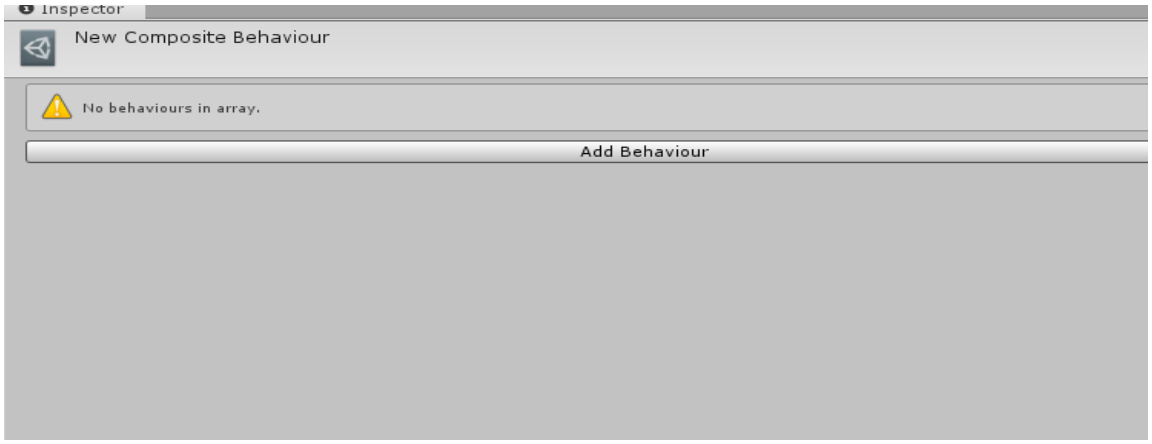
I had finished all of the coding and went to test my new custom inspector (although I checke. However, since this is my first custom inspector there were bound to be problems! I

found that none of my behaviours showed up, buttons weren't appearing/working etc. A fter many hours of adapting the layout via BeginHorizontal(); and EndHorizontal(); and removing all ExitGUI(); functions, I have finally had success!

The layout of the inspector is not ideal as I would rather have the buttons being added vertically rather than horizontally, but here is the result:



Of course, when I created a new composite behaviour, we get the error message and an add behaviour button:



Now, I am going to solve one of the minor issues that I have with the current project – The agents moving out off the screen in runtime. To fix this I am going to create a new behaviour for the agents that will encourage them to stay within a certain radius. This won't prevent them leaving the screen, but it will incline them to stay.

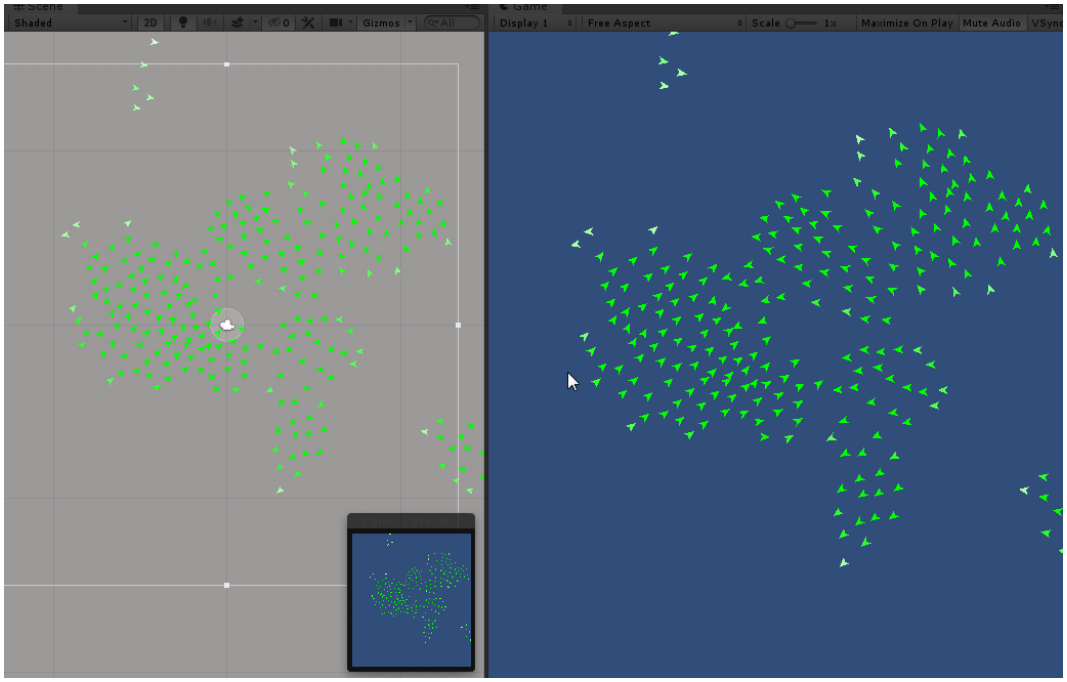
Here is the script:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "Flock/Behaviour/Stay In Radius")]
public class StayInRadiusBehaviour : FlockBehaviour
{
    // This vector will default to 0,0 which is fine
    public Vector2 center;
    public float radius = 15f;

    // Override this vector
    public override Vector2 CalculateMove(FlockAgent agent, List<Transform> context, Flock flock)
    {
        // Find out how far away from the center the agent is, may have to call them back!
        // This line gives us the distance from the center of the screen
        Vector2 centerOffset = center - (Vector2)agent.transform.position;

        // If t is 0 we are at the center
        // Costly, however if I used SqrMagnitude the ratios would not be entirely correct
        float t = centerOffset.magnitude / radius;

        // Check distance from the radius:
        // If within 90% of the radius, don't bother to act upon it
        if (t < 0.9f)
        {
            return Vector2.zero;
        }


        return centerOffset * t * t;
    }
}
```

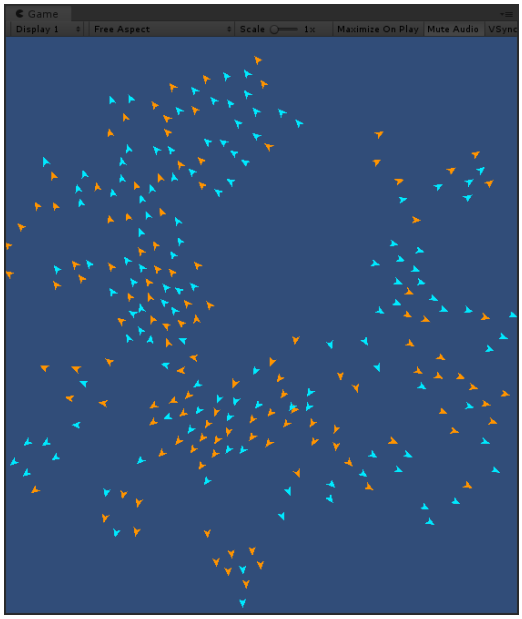
It simply uses a vector at the center of the screen called t. We then minus the agent's position from the centerOffset vector to find out where the agent needs to move. The 'if' statement only takes effect if the agent is not within 90% of the radius.

I then created my scriptable object and called it 'Stay In R15' for radius 15f.

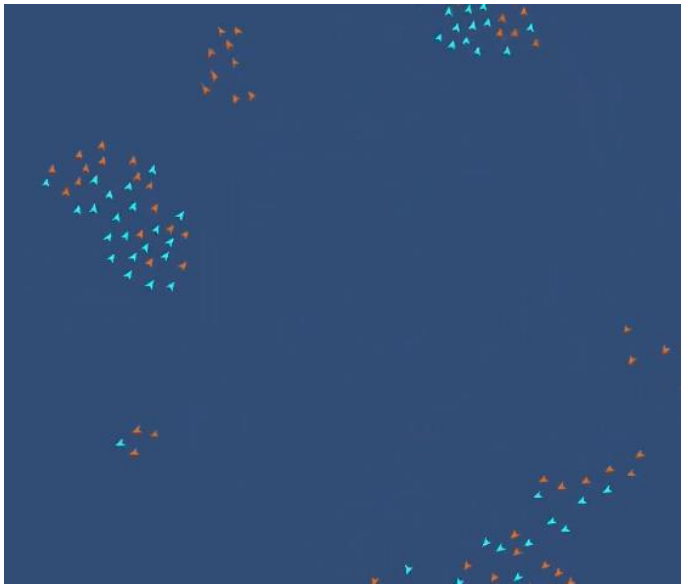
I then duplicated the 'Default Flock' object and called the new one 'Centered Flock Behaviour'. I then added in the new behaviour and altered it's weight to '0.25'. Now, there is a clear influence on the agents that is causing them to stay towards the middle of the screen. If the value is set to something like 0.8 then there is more of an obvious circle pattern, however 0.25 is perfect for simply keeping the flocks (almost) on screen as you can see:



Next up I am going to create a filtration system so that agents can make decisions based off of their neighbours and/or the identities of their neighbours. To help this become apparent, I am going to create an additional flock. To begin I have duplicated my Flock Agent prefab twice and called one ‘Flock Agent Orange’ and one ‘Flock Agent Blue’. Then I altered their sprite renderer’s colours to match. Then I duplicated the flock gameobject in my scene and did the same, so I now have a blue flock and an orange flock (of course changing the prefab they’re both using to match). This gave the following result:



However, as things are left for a few seconds to work…



You can see that the flocks still treat each other as one large flock. They stick together and avoid each other just as if they weren't separated. I want to change this, so we have two completely separate flocks acting independently.

The first thing I need to implement, is a way for each agent to know which flock they are a part of. Secondly, I will apply a filter that takes in a list of the transforms around an agent and filters them via a criterion that I am going to create. Because I am using scriptable objects, these filters will also be scriptable objects. This way, the behaviours will be able to apply them.

To get started, I have added a few variables to the FlockAgent script.

```
// Variable
Flock agentFlock;
public Flock AgentFlock { get { return agentFlock; } }
```

Next I have created this method in the same script:

```
public void Initialise(Flock flock)
{
    agentFlock = flock;
}
```

This essentially assigns the new Flock type to a flock. Then I have called this method within the 'flock' script where the agent gets instantiated:

```
// Give the agents a name so we can keep track of them more easily
newAgent.name = "Agent " + i;
// When the agent is created it gets added to it's particular flock
newAgent.Initialise(this);
// Now add each agent to our list of agents
agents.Add(newAgent);
```

Now I need a filter to go through the neighbouring transforms and discards some depending on my criteria.

Then I created two new folders called ‘Filter Scripts’ and ‘Filter Objects’. In the former, I have created two new scripts. One called ‘SameFlockFilter’ and one called ‘ContextFilter’. Here is the entire ‘ContextFilter’ script (explained via commenting):

```
// Abstract class inheritting from ScriptableObject
public abstract class ContextFiler : ScriptableObject
{
    // New method returns a list of transforms, called Filter and takes in the flock agent (for comparisons) and the original List of
    // Neighbour's transforms
    public abstract List<Transform> Filter(FlockAgent agent, List<Transform> original);
}
```

And here is the ‘SameFlockFilter’ script (also explained via commenting):

```
[CreateAssetMenu(menuName = "Flock/Filter/Same Flock")]
public class SameFlockFilter : ContextFilter
{
    public override List<Transform> Filter(FlockAgent agent, List<Transform> original)
    {
        // Set up a new filtered List
        List<Transform> filtered = new List<Transform>();

        // Iterate through original List and see if
        // #1 It is a flock agent?
        // #2 Is it a member of the same flock?
        // If both are true, it can be added to the new filtered List
        foreach (Transform item in original)
        {
            FlockAgent itemAgent = item.GetComponent<FlockAgent>();
            // If this item isn't a flock agent, the item will be null
            if (itemAgent != null && itemAgent.AgentFlock == agent.AgentFlock)
            {
                filtered.Add(item);
            }
        }
        return filtered;
    }
}
```

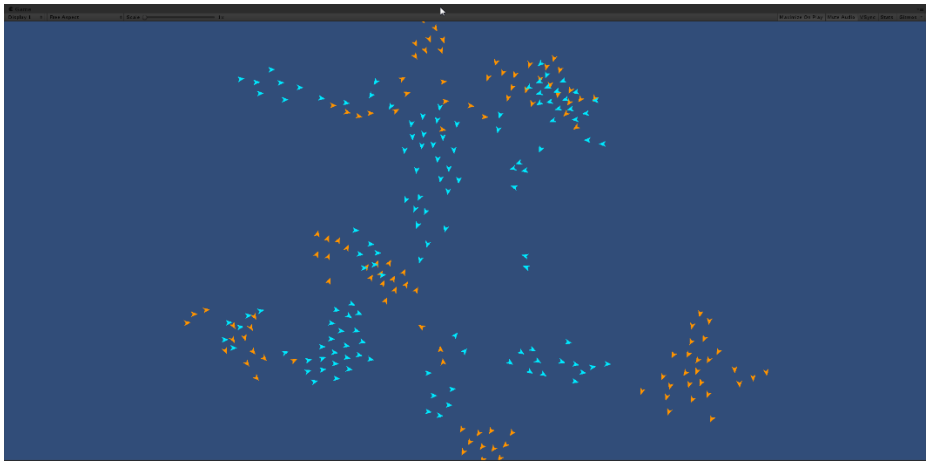
Next, I have created a new filter scriptableobject inside the ‘filter objects’ folder, called ‘Same Flock Filter’. At the moment, there is nowhere to reference this object. I have to be careful about how I am implementing it because some behaviours should not have the filter, such as the ‘stay in radius’ behaviour. It does not care about other objects, only about it’s center and radius. To deal with this, I am going to create a middleman abstract class. I say middleman because the new filter will inherit from FlockBehaviour. This means some things will inherit from the filter AND FlockBehaviour, whereas others will ONLY inherit from FlockBehaviour. This sounds complicated but is actually quite simple. Here is the new filter class (afterwards I took the spaces out the script’s name and changed it to FilteredFlockBehaviour in-script):

```
using System.Collections.Generic;
using UnityEngine;

public abstract class Filtered : FlockBehaviour
{
    public ContextFilter filter;
}
```

This inherits from FlockBehaviour like I just stated, and so in the classes I want to add the filter to, I will simply change their parent class from FlockBehaviour to Filtered. To classes I am filtered are: 'AlignmentBehaviour', 'AvoidanceBehaviour', 'CohesionBehaviour' and 'SteeredCohesionBehaviour'. I then went back into Unity and applied the filter to each scriptable object. At this stage, the filters are assigned correctly but are not actually applied. To apply the filter where necessary, I used this code in each of the 4 scripts (in place of searching through items 'in context'):

```
// We go through each items transform in our list of neighbours
// If using filter, we choose the filtered list of transforms, otherwise ignore this
List<Transform> filteredContext = (filter == null) ? context : filter.Filter(agent, context);
foreach (Transform item in filteredContext)
```

As you can see, now each coloured flock is behaving independently and each behaviour is only applying to their actual neighbours, so success:



Finally, I am going to add some more functionality to the flocks in the form of obstacle avoidance. This would be very useful in certain cases such as flocking fish avoiding sharks in a deep-sea game, for example.

To begin, I have created a 2D sprite, used the default 'knob' sprite and put it on a new layer called 'Obstacle'. I have also added a CircleCollider2D to the obstacle. I then duplicated it to make some sort of wall. To get the functionality I want I'm going to create another filter. This will look at nearby Transforms and avoid objects on the obstacle's physics layer. To get started I have created a new script inside the filter scripts folder called 'PhysicsLayerFilter'. I have copied the main filter layout from the previous script so it can be altered accordingly.

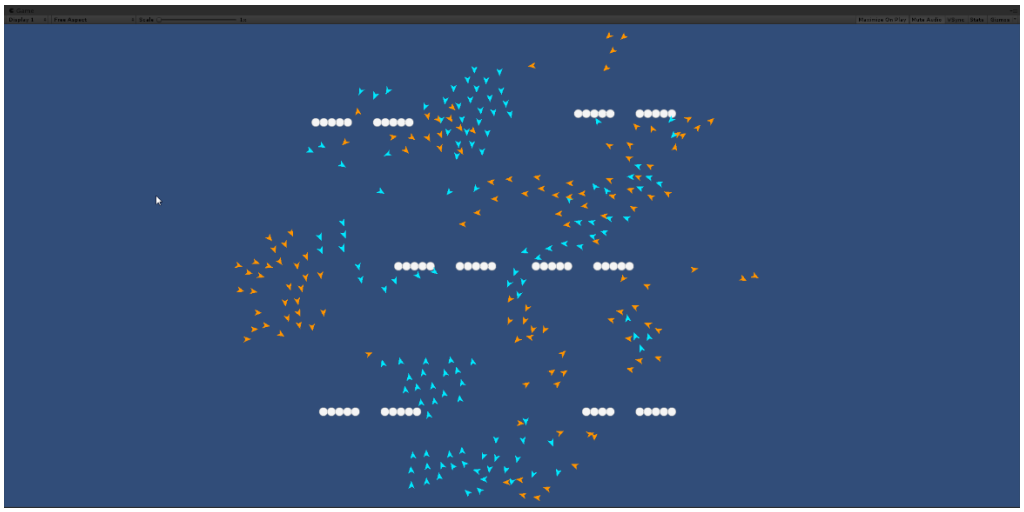
To get the functionality I want, I will be using a process called Bit-shifting as well as logic gates (for the first time). Here is the short piece of code (explained via commenting)

```
[CreateAssetMenu(menuName = "Flock/Filter/Physics Layer")]
public class PhysicsLayerFilter : ContextFilter
{
    // Variables
    public LayerMask mask;

    public override List<Transform> Filter(FlockAgent agent, List<Transform> original)
    {
        // Set up a new filtered List
        List<Transform> filtered = new List<Transform>();

        // Iterate through original List and see if
        // #1 It is a flock agent?
        // #2 Is it a member of the same flock?
        // If both are true, it can be added to the new filtered List
        foreach (Transform item in original)
        {
            // << represents bitshifting
            if (mask == (mask | (1 << item.gameObject.layer))){
                // Item is on a layer where mask is checked off
                filtered.Add(item);
            }
        }
        return filtered;
    }
```

:

I then created a new physics layer filter in the filter objects folder. I called it Obstacle Layer Filter and set its 'mask' to Obstacle. Then, in the behaviour objects folder I have duplicated the 'avoidance' behaviour and called it 'Avoid Obstacles'. I have then assigned the new object's filter to the one I just made. Next, I have duplicated the 'Centered flock behaviour' object and called the new one 'Obstacle flock behaviour'. This means I can keep the old behaviour on one flock and apply the obstacle avoidance script to the other flock. Using my new inspector I added the 'Avoid Obstacles' behaviour I just made and set it's weight to 5. I need the weight to be high because avoiding the obstacles must be a huge priority for the agents, or they will appear broken!

Here is a picture of the final project in action, where the orange flock is avoiding the obstacles and the blue flock is doing it's own thing:



Reflection Section

During this project I have learnt a LOT. Perhaps most importantly how to structure such complex systems using scriptable objects, inheritance, filters and bunching up different behaviours into one agent. The diagrams at the start of the project ended up helping me greatly so I will definitely make sure to plan properly in the future also. I have also learnt a huge amount about how custom inspectors work and how to code them (thanks to all my troubles!), since I had only glanced over them before. I learnt about using [CreateAssetMenu] which I wouldn't have otherwise known existed and practiced using properties like getters and setters. I know how to approach larger scale AI projects in the future with more organisation and a more confident approach to the code involved!