Lab 6 – Flocking

COMP396-Game Programming 2

Purpose: Implementing Flocking Behaviour with Craig Reynolds algorithm.

1 Introduction

Let's implement flocking behaviour. We'll try two variations:

- The **first** one is based on an old flocking behavior demo that has been circulating in the Unity community since since the game engine was created.
- The **second** variation is based on Craig Reynold's original flocking algorithm from 1986.

2 Basic flocking behavior

Let's hear from the “Horse's Mouth” :)

> In 1986 I made a computer model of coordinated animal motion such as **bird flocks** and **fish schools**. It was based on three dimensional computational geometry of the sort normally used in computer animation or computer aided design. I called the generic simulated flocking creatures **boids**. The basic flocking model consists of **three simple steering behaviors** which describe how an individual boid **maneuvers** based on the **positions** and **velocities** its nearby flockmates:
>
> **Separation:** steer to avoid **crowding** local flockmates
> **Alignment:** steer towards **the average heading** of local flockmates
> **Cohesion:** steer to move toward the **average position** of local flockmates
> — Craig Reynolds, Boids, Background and Update

Note that, historically, Craig wrote his “Steering...” paper much later than his “Flocks...” paper. In his “Flocks...” paper he doesn't even use the word “steer”, and, also, he uses different terms altogether (albeit with the same meaning):

> ... **Simulated Flocks**
> To build a simulated flock, we start with a **boid model** that supports geometric flight. We add behaviors that correspond to the **opposing forces** of **collision**

avoidance and the **urge to join** the flock. Stated briefly as rules, and in order of decreasing precedence, the behaviors that lead to simulated flocking are:

> **Collision Avoidance:** avoid **collisions** with nearby flockmates
> **Velocity Matching:** attempt to match **velocity** with nearby flockmates
> **Flock Centering:** attempt to **stay close** to nearby flockmates
> — Craig Reynolds, Flocks, Herds, and Schools

Notice also the substitution of **local** for **nearby**.

3 Setting Up the Demo scene

- Create a word document named **Lab6_Snapshots_{YourInitials}** to hold your snapshots.
- Open the project **COMP396_001_F24_{YourInitials}** from last labs.
- Create a folder under Scenes folder named **Lab6_{YourInitials}**.
- Create a new scene named **Lab6_Flocking_Version1_{YourInitials}** and save it in the above folder.
- *Take Snapshot*
- Set up the scene with the following game objects:

Game Object	Type	Parent	P(x,y,z)	R(x,y,z)	S(x,y,z)	Color
Floor	Plane	Root	(0,0,0)	(0,0,0)	(20,1,20)	Beige
Flock	Empty	Root	(0,0,0)	(0,0,0)	(1,1,1)	-
Boid 1-10	Cube	Flock	Random	Random	Random	White

- *Take Snapshot* - Copy the scene to another scene named **Lab6_Flocking_Version2_{YourInitials}** and save it in the above folder. - Note that **Flock** is the controller.

3.1 Version 1

- Open the scene named **Lab6_Flocking_Version1_{YourInitials}** and and work there. Let's create a demo scene with flocks of objects and implement the flocking behavior in C#. - Open the project from last time: ** For this first version, we compute all the rules by ourselves. Also, we will create a boid commander that leads the crowd to control and track the general position of the flock easily. You can see the Hierarchy scene in the following screenshot. As you can see, we have several boid entities named UnityFlock, under a controller named UnityFlockController. UnityFlock entities are individual boid objects that refer to their parent UnityFlockController entity, using it as a leader. The controller updates the next destination point randomly once it reaches the current destination point:

Figure 5.1 – The scene hierarchy UnityFlock is a prefab with just a cube mesh and a UnityFlock script. We can use any other mesh representation for this prefab to represent something more interesting, such as birds. You can add as many UnityFlock prefabs as you like. The algorithm will automatically check the number of children in the UnityFlockController object.

3.2 Individual behavior

Boid is a term coined by Craig Reynold that refers to bird-like objects. We use this term to describe each object in our flock. The boid behaviour consists of a group of objects, each having their individual position, velocity, and orientation. Now, let's implement the boid behavior. You can find the behavior that controls each boid in the flock in the UnityFlock.cs script, which we'll examine now:

```
using UnityEngine;
using System.Collections;
public class UnityFlock : MonoBehaviour {
  public float minSpeed = 20.0f;
  public float turnSpeed = 20.0f;
  public float randomFreq = 20.0f;
  public float randomForce = 20.0f;
  //alignment variables
  public float toOriginForce = 50.0f;
  public float toOriginRange = 100.0f;
  public float gravity = 2.0f;
  //seperation variables
  public float avoidanceRadius = 50.0f;
  public float avoidanceForce = 20.0f;
  //cohesion variables
  public float followVelocity = 4.0f;
  public float followRadius = 40.0f;
  //these variables control the movement of the boid
  private Transform origin;
  private Vector3 velocity;
  private Vector3 normalizedVelocity;
  private Vector3 randomPush;
  private Vector3 originPush;
  private Transform[] objects;
  private UnityFlock[] otherFlocks;
  private Transform transformComponent;
  private float randomFreqInterval;
```

As public fields, we declare the input values for our algorithm. These can be set up and customized from within the Inspector. In this script, we perform the following operations:

1. We define the minimum movement speed (minSpeed) and rotation speed (turnSpeed) for our boid.
2. We use randomFreq to determine how many times we want to update the randomPush value, based on the randomForce value. Then, we use this force to vary the single boid's velocity and make the flock's movement look more realistic.

3. toOriginRange specifies how much we want the flock to spread out. In other words, it represents the maximum distance from the flock's origin in which we want to maintain the boids (following the previously mentioned cohesion rule). We use the avoidanceRadius and avoidanceForce properties to maintain a minimum distance between individual boids (following the separation rule). Similarly, we use followRadius and followVelocity to keep a minimum distance between the leader or origin of the flock. The origin variable stores the parent object that controls the entire flock; in other words, it is the flock leader. The boids need to know about the other boids in the flock. Therefore, we use the objects and otherFlocks attributes to store the neighboring boid's information.

This is the initialization method for our boid:

```
void Start () {
  randomFreqInterval = 1.0f / randomFreq;
  origin = transform.parent;   // Assign the parent as origin
  transformComponent = transform; // Flock transform
  Component[] tempFlocks= null;  // Temporary components
  // Get all the unity flock components from the parent transform in the group
  if (transform.parent) {
    tempFlocks = transform.parent.GetComponentsInChildren<unityflock>();
  }
  // Assign and store all the flock objects in this group
  objects = new Transform[tempFlocks.Length];
  otherFlocks = new UnityFlock[tempFlocks.Length];
  for (int i = 0;i < tempFlocks.Length;i++) {
    objects[i] = tempFlocks[i].transform;
    otherFlocks[i] = (UnityFlock)tempFlocks[i];
  }
  // Null Parent as the flock leader will be UnityFlockController object
  transform.parent = null;
  // Calculate random push depends on the random frequency provided
  StartCoroutine(UpdateRandom());
}
```

We set the parent of the object of our boid as origin, meaning that this is the controller object for the other boids to follow. Then, we grab all the other boids in the group and store them in the otherFlocks attribute for later reference.

3.3 COROUTINES

Put simply, coroutines are functions that can be paused. With coroutines, you can run a method, pause the execution for a desired amount of time (for example, a single frame or several seconds), and then continue from the following line as if nothing happened. They have two primary use cases: to run a function after a specific interval (without keeping track of every frame of elapsedTimes, as we did in other examples) or to split the computation of some heavy algorithm over multiple frames (and, therefore, not incur in frame drops). Coroutines, it turns out, are a pretty helpful tool to master. You can read more at Coroutines. Now, we can implement the UpdateRandom coroutine. As a

coroutine, the function never actually terminates, but we run the body of the while loop for each random time interval:

1. We define the UpdateRandom method as a coroutine by specifying the IEnumerator return type:

```
IEnumerator UpdateRandom() {
  while (true) {
    randomPush = Random.insideUnitSphere * randomForce;
    yield return new WaitForSeconds(randomFreqInterval
       + Random.Range(-randomFreqInterval / 2.0f, randomFreqInterval / 2.0f));
  }
}
```

2. The UpdateRandom() method updates the randomPush value throughout the game with an interval based on randomFreq. Random.insideUnitSphere returns a Vector3 object with random x, y, and z values within a sphere, with a radius of the randomForce value.
3. We wait for a certain random amount of time before resuming while(true).
4. Loop to update the randomPush value again.
5. Now, here is our boid behavior's Update() method, which helps the boid entity comply with the three rules of the flocking algorithm:

```
void Update() {
    //Internal variables
    float   speed = velocity.magnitude ;
    Vector3 avgVelocity = Vector3.zero;
    Vector3 avgPosition = Vector3.zero;
    int     count = 0;
    Vector3 myPosition = transformComponent.position;
    Vector3 forceV;
    Vector3 toAvg;
    for (int i = 0; i < objects.Length; i++) {
      Transform boidTransform = objects[i];
      if (boidTransform != transformComponent) {
        Vector3 otherPosition = boidTransform.position;
        // Average position to calculate cohesion
        avgPosition += otherPosition;
        count++;
        //Directional vector from other flock to this flock
        forceV = myPosition - otherPosition;
        //Magnitude of that directional vector(Length)
        float directionMagnitude = forceV.magnitude;
        float forceMagnitude = 0.0f;
        if (directionMagnitude < followRadius)
        {
          if (directionMagnitude < avoidanceRadius) {
            forceMagnitude = 1.0f - (directionMagnitude / avoidanceRadius);
            if (directionMagnitude > 0)
              avgVelocity += (forceV / directionMagnitude) * forceMagnitude *
avoidanceForce;
          }
          forceMagnitude = directionMagnitude / followRadius;
```

```
          UnityFlock tempOtherBoid = otherFlocks[i];
          avgVelocity += followVelocity * forceMagnitude *
tempOtherBoid.normalizedVelocity;
        }
      }
    }
```

The preceding code implements the separation rule. First, we check the distance between the current boid and the other boids, and then we update the velocity accordingly, as explained in the comments in the preceding code block. 6. We now calculate the average velocity vector of the flock by dividing the current velocity vector by the number of boids in the flock:

```
    if (count > 0) {
      //Calculate the average flock velocity(Alignment)
      avgVelocity /= count;
      //Calculate Center value of the flock(Cohesion)
      toAvg = (avgPosition / count) - myPosition;
    } else {
      toAvg = Vector3.zero;
    }
    //Directional Vector to the leader
    forceV = origin.position - myPosition;
    float leaderDirectionMagnitude = forceV.magnitude;
    float leaderForceMagnitude = leaderDirectionMagnitude / toOriginRange;
    //Calculate the velocity of the flock to the leader
    if (leaderDirectionMagnitude > 0)
      originPush = leaderForceMagnitude * toOriginForce * (forceV /
leaderDirectionMagnitude);
    if (speed < minSpeed && speed > 0) {
      velocity = (velocity / speed) * minSpeed;
    }
    Vector3 wantedVel = velocity;
    //Calculate final velocity
    wantedVel -= wantedVel * Time.deltaTime;
    wantedVel += randomPush * Time.deltaTime;
    wantedVel += originPush * Time.deltaTime;
    wantedVel += avgVelocity * Time.deltaTime;
    wantedVel += gravity * Time.deltaTime * toAvg.normalized;
    velocity = Vector3.RotateTowards(velocity, wantedVel, turnSpeed *
Time.deltaTime, 100.00f);
    transformComponent.rotation = Quaternion.LookRotation(velocity);
    //Move the flock based on the calculated velocity
    transformComponent.Translate(velocity * Time.deltaTime, Space.World);
    normalizedVelocity = velocity.normalized;
  }
```

7. We add up all the factors, such as randomPush, originPush, and avgVelocity, to calculate the final target velocity vector, wantedVel. We also update the current velocity to wantedVel with a linear interpolation by using the Vector3.RotateTowards method.
8. We move our boid based on the new velocity using the Translate method.

9. As a final touch, we create a cube mesh, to which we add the UnityFlock script, and then save it as a prefab, as shown in the following screenshot:

Figure 5.2 – The UnityFlock prefab

3.4 Controller

Now, it is time to create the controller class. This class updates its position so that the other individual boid objects know where to go. The origin variable in the preceding UnityFlock script contains a reference to this object. The following is the code in the UnityFlockController.cs file:

```
using UnityEngine;
using System.Collections;
public class UnityFlockController : MonoBehaviour {
  public Vector3 bound;
  public float speed = 100.0f;
  public float targetReachedRadius = 10.0f;
  private Vector3 initialPosition;
  private Vector3 nextMovementPoint;
  // Use this for initialization
  void Start () {
    initialPosition = transform.position;
    CalculateNextMovementPoint();
  }
  // Update is called once per frame
  void Update () {
    transform.Translate(Vector3.forward * speed * Time.deltaTime);
    Quaternion targetRot= Quaternion.LookRotation(nextMovementPoint -
transform.position);
    transform.rotation = Quaternion.Slerp(transform.rotation, targetRot, 1.0f *
Time.deltaTime);
    if (Vector3.Distance(nextMovementPoint, transform.position) <=
targetReachedRadius)
      CalculateNextMovementPoint();
  }
In the Update() method, we check whether our controller object is near the target
destination point. If it is, we update the nextMovementPoint variable again with the
CalculateNextMovementPoint() method that we just discussed:
void CalculateNextMovementPoint () {
  float posX = Random.Range(initialPosition.x - bound.x, initialPosition.x +
bound.x);
  float posY = Random.Range(initialPosition.y - bound.y, initialPosition.y +
bound.y);
  float posZ = Random.Range(initialPosition.z - bound.z, initialPosition.z +
bound.z);
  nextMovementPoint = initialPosition + new Vector3(posX, posY, posZ);
}
```

The CalculateNextMovementPoint() method finds the next random destination position in a range between the current position and the boundary vectors. Finally, we put all of this

together, as shown in Figure 5.1, which should give you flocks of squares flying around realistically in the sunset:

Figure 5.3 – A demonstration of the flocking behavior using the Unity seagull sample The previous example gave you the basics of flocking behaviors. In the next section, we will explore a different implementation that makes use of Unity's Rigidbody component.

4 Alternative implementation

In this section, we use the Unity physics engine to simplify the code a bit. In fact, in this example, we attach a Rigidbody component to the boids to use the Rigidbody properties to translate and steer them. In addition, the Rigidbody component is also helpful in preventing the other boids from overlapping with each other. In this implementation, we have two components: the individual boid behavior and the controller behavior (the element referred to as the flock controller in the previous section). As before, the controller is the object that the rest of the boids follow. The code in the Flock.cs file is as follows:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
public class Flock : MonoBehaviour {
  internal FlockController controller;
  private new Rigidbody rigidbody;
  private void Start() {
    rigidbody = GetComponent<rigidbody>();
  }
  void Update () {
    if (controller) {
      Vector3 relativePos = Steer() * Time.deltaTime;
      if (relativePos != Vector3.zero)
       rigidbody.velocity = relativePos;
      // enforce minimum and maximum speeds for the boids
      float speed = rigidbody.velocity.magnitude;
      if (speed > controller.maxVelocity) {
        rigidbody.velocity = rigidbody.velocity.normalized * controller.maxVelocity;
      } else if (speed < controller.minVelocity) {
        rigidbody.velocity = rigidbody.velocity.normalized * controller.minVelocity;
      }
    }
  }
```

We will create FlockController in a moment. In the meantime, in the Update() method in the previous code block, we calculate the boid's velocity using the Steer() method and apply the result to the boid's rigid-body velocity. Next, we check whether the current speed of the Rigidbody component falls inside our controller's maximum and minimum velocity ranges. If not, we cap the velocity at the preset range:

```
private Vector3 Steer () {
  Vector3 center = controller.flockCenter - transform.localPosition;       //
```

```
cohesion
  Vector3 velocity = controller.flockVelocity - rigidbody.velocity;        //
allignement
  Vector3 follow = controller.target.localPosition - transform.localPosition; //
follow leader
  Vector3 separation = Vector3.zero;
  foreach (Flock flock in controller.flockList) {
    if (flock != this) {
      Vector3 relativePos = transform.localPosition - flock.transform.localPosition;
      separation += relativePos.normalized;
    }
  }
  // randomize
  Vector3 randomize = new Vector3( (Random.value * 2) - 1, (Random.value * 2) - 1,
(Random.value * 2) - 1);
  randomize.Normalize();
  return (controller.centerWeight  * center +
      controller.velocityWeight  * velocity +
      controller.separationWeight * separation +
      controller.followWeight    * follow +
      controller.randomizeWeight * randomize);
}
```

The steer() method implements the separation, cohesion, alignment, and follows the leader rules of the flocking algorithm. Then, we add up all the factors with a random weight value. We use this Flock script together with the Rigidbody and SphereCollider components to create a Flock prefab, as shown in the following screenshot (make sure to disable the gravity by unchecking Use Gravity):

Figure 5.4 – Flock It is now time to implement the final piece of the puzzle: the FlockController component. FlockController This FlockController component is similar to the one in the previous example. In addition to controlling the flock's speed and position, this script also instantiates the boids at runtime:

1. The code in the FlockController.cs file is as follows:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
public class FlockController : MonoBehaviour {
  public float minVelocity = 1;
  public float maxVelocity = 8;
  public int flockSize = 20;
  public float centerWeight = 1;
  public float velocityWeight = 1;
  public float separationWeight = 1;
  public float followWeight = 1;
  public float randomizeWeight = 1;
  public Flock prefab;
  public Transform target;
  Vector3 flockCenter;
  internal Vector3 flockVelocity;
```

```
  public ArrayList flockList = new ArrayList();
  void Start () {
    for (int i = 0; i < flockSize; i++) {
      Flock flock = Instantiate(prefab, transform.position, transform.rotation) as
Flock;
      flock.transform.parent = transform;
      flock.controller = this;
      flockList.Add(flock);
    }
  }
```

2. We declare all the public properties to implement the flocking algorithm and then start generating the boid objects based on the flock size input.
3. We set up the controller class and the parent Transform object, as we did last time.
4. We add every boid object we create to the flockList array. The target variable accepts an entity to be used as a moving leader. In this example, we create a sphere entity as a moving target leader for our flock:

```
  void Update() {
    //Calculate the Center and Velocity of the whole flock group
    Vector3 center = Vector3.zero;
    Vector3 velocity = Vector3.zero;
    foreach (Flock flock in flockList) {
        center += flock.transform.localPosition;
        velocity += flock.GetComponent<rigidbody>().velocity;
    }
    flockCenter = center / flockSize;
    flockVelocity = velocity / flockSize;
  }
}
```

5. In the Update method, we keep updating the average center and velocity of the flock. These are the values referenced from the boid object and are used to adjust the cohesion and alignment properties with the controller:

Figure 5.5 – Flock Controller We need to implement our Target entity with the Target Movement (Script). The movement script is the same as what we saw in our previous Unity3D sample controller's movement script:

Figure 5.6 – The Target entity with the TargetMovement script 6. Here is how our TargetMovement script works: we pick a random point nearby for the target to move to, and when we get close to that point, we pick a new one. The code in the TargetMovement.cs file is as follows:

```
using UnityEngine;
using System.Collections;
public class TargetMovement : MonoBehaviour {
  // Move target around circle with tangential speed
  public Vector3 bound;
  public float speed = 100.0f;
  public float targetReachRadius = 10.0f;
  private Vector3 initialPosition;
```

```
  private Vector3 nextMovementPoint;
  void Start () {
    initialPosition = transform.position;
    CalculateNextMovementPoint();
  }
  void CalculateNextMovementPoint () {
    float posX = Random.Range(initialPosition.x = bound.x,
initialPosition.x+bound.x);
    float posY = Random.Range(initialPosition.y = bound.y,
initialPosition.y+bound.y);
    float posZ = Random.Range(initialPosition.z = bound.z,
initialPosition.z+bound.z);
    nextMovementPoint = initialPosition + new Vector3(posX, posY, posZ);
  }
  void Update () {
    transform.Translate(Vector3.forward * speed * Time.deltaTime);
    transform.rotation =
     Quaternion.Slerp(transform.rotation,
    Quaternion.LookRotation(nextMovementPoint - transform.position),
Time.deltaTime);
    if (Vector3.Distance(nextMovementPoint, transform.position) <=
targetReachRadius)
     CalculateNextMovementPoint();
  }
}
```

7. After we put everything together, we should see a nice flock of cubic boids flying around in the scene, all chasing the spheric target:

Figure 5.7 – Flocking with Craig Reynold's algorithm Now that we have implemented flocking behavior in two different ways, we can experiment with different parameters and tweak the boids' behavior until we find a movement we like. I want to conclude this chapter with an important takeaway: note that we do not need complex algorithms to have a natural-looking behavior. For that, we just need simple reactive rules and a way to combine them.

5 Summary

In this chapter, we learned how to implement flocking behaviors in two ways. First, we examined and learned how to implement a basic flocking algorithm using nothing other than our scripts. Next, we implemented the same algorithm using Unity's Rigidbody component to control the boid's movement and Sphere Collider to avoid collision with other boids. In our example, we always referred to boids as bird-like entities. However, we can use flocking for many other applications: fishes swimming in the sea, sheep grazing on a plane, a swarm of insects, and even groups of people walking on the street can show flocking behavior. To adapt the algorithm to different scenarios, we just need to change the flocking rules' values and eventually lock the movement to a plane. In the next chapter, we will go beyond random movement and look at how to follow a specific path. This is the first step toward learning how to avoid obstacles that are in your way.

==========

>>> **Create a Unity Flocking Game** Here's a basic implementation of a Flocking game in Unity. This example includes three types of flocking behaviors: Cohesion, Separation, and Alignment.

Flock.cs

This script will serve as the brain for our flocking entities.

```
using UnityEngine;

public class Flock : MonoBehaviour
{
 public int maxNeighbors = 5;      // maximum number of neighbors to consider for cohesion and alignment
 public float cohesionStrength = 1.0f;  // how strong is the desire to stay close to others?
 public float separationStrength = 1.0f; // how strong is the desire to avoid collisions?
 public float alignmentStrength = 1.0f; // how strong is the desire to align with others?

 private FlockBehavior cohesion;
 private FlockBehavior separation;
 private FlockBehavior alignment;

 void Start()
 {
  cohesion = new Cohesion(maxNeighbors, cohesionStrength);
  separation = new Separation(maxNeighbors, separationStrength);
  alignment = new Alignment(maxNeighbors, alignmentStrength);

  // Initialize the flock's position and velocity
  transform.position = Random.insideOut(10f);
  GetComponent<rigidbody>().velocity = Vector3.zero;
 }

 void FixedUpdate()
 {
  cohesion.Update(transform.position);
  separation.Update(transform.position);
  alignment.Update(transform.position);

  // Combine the forces to get a new velocity
  Vector3 newVelocity = cohesion.GetForce() + separation.GetForce() + alignment.GetForce();

  // Normalize the velocity vector
  newVelocity.Normalize();

  // Apply the force to the entity's Rigidbody
  GetComponent<rigidbody>().AddForce(newVelocity * 10f);
 }
```

```
 void Update()
 {
  // Keep the flock within a certain radius
  if (Vector3.Distance(transform.position, transform.parent.position) > 20f)
  {
   transform.position = Vector3.MoveTowards(transform.position,
transform.parent.position, 1f);
  }
 }
}
```

FlockBehavior.cs

This script defines an abstract class for the different flocking behaviors. Each behavior should inherit from this and implement its own logic.

```
using UnityEngine;
public abstract class FlockBehavior
{
 public float strength; // how strong is this behavior?
 protected int maxNeighbors;
 protected Vector3[] neighbors;
 void Start()
 {
  // Initialize the neighbors array to store nearby entities
  maxNeighbors = 5;
  neighbors = new Vector3[maxNeighbors];
 }
 abstract public Vector3 GetForce(); // returns a vector representing the force of
this behavior
 abstract public void Update(Vector3 position); // updates the neighbors array and
calculates the force for this behavior
}
```

Cohesion.cs

This script implements the cohesion behavior, which is the tendency to move towards other entities in the flock.

```
using UnityEngine;

public class Cohesion : FlockBehavior
{
 public override Vector3 GetForce()
 {
  // Calculate the average position of all neighbors
  float sumX = 0f;
  float sumY = 0f;
  float sumZ = 0f;
  for (int i = 0; i < maxNeighbors; i++)
  {
   if (neighbors[i] != Vector3.zero)
```

```
   {
    sumX += neighbors[i].x;
    sumY += neighbors[i].y;
    sumZ += neighbors[i].z;
   }
  }
  float averageX = sumX / maxNeighbors;
  float averageY = sumY / maxNeighbors;
  float averageZ = sumZ / maxNeighbors;
  Vector3 averagePosition = new Vector3(averageX, averageY, averageZ);
  // Return a force vector pointing towards the average position
  return (averagePosition - transform.position).normalized * cohesionStrength;
 }

 public override void Update(Vector3 position)
 {
  // Find nearby entities within a certain radius
  float radius = 10f;
  int count = 0;
  for (int i = 0; i < maxNeighbors; i++)
  {
   Vector3 neighborPosition = position + Random.onUnitSphere() * radius;
   Flock entity = GameObject.Find("FlockEntity").GetComponent<flock>();
   if (entity != null && Vector3.Distance(position, neighborPosition) <= radius)
   {
    neighbors[count++] = neighborPosition;
   }
   if (count == maxNeighbors) break;
  }
 }
}
```

Separation.cs

This script implements the separation behavior, which is the tendency to avoid collisions with other entities in the flock.

```
using UnityEngine;
public class Separation : FlockBehavior
{
 public override Vector3 GetForce()
 {
  // Check for nearby entities and calculate a force vector away from them
  float minDistance = 0.1f;
  Vector3 separationForce = new Vector3(0f, 0f, 0f);
  foreach (Vector3 neighbor in neighbors)
  {
   if (neighbor != Vector3.zero && Vector3.Distance(transform.position, neighbor) <
minDistance)
   {
    separationForce += (transform.position - neighbor).normalized *
separationStrength;
   }
  }
```

```
 return separationForce;
 }

 public override void Update(Vector3 position)
 {
  // Find nearby entities within a certain radius
  float radius = 10f;

  int count = 0;
  for (int i = 0; i < maxNeighbors; i++)
  {
   Vector3 neighborPosition = position + Random.onUnitSphere() * radius;

   Flock entity = GameObject.Find("FlockEntity").GetComponent<flock>();

   if (entity != null && Vector3.Distance(position, neighborPosition) <= radius)
   {
    neighbors[count++] = neighborPosition;
   }

   if (count == maxNeighbors) break;
  }
 }
}
```

Alignment.cs

This script implements the alignment behavior, which is the tendency to align with other entities in the flock.

```
using UnityEngine;

public class Alignment : FlockBehavior
{
 public override Vector3 GetForce()
 {
  // Calculate the average velocity of all neighbors
  Vector3 sum=Vector3.zero;

  for (int i = 0; i < maxNeighbors; i++)
  {
   if (neighbors[i] != Vector3.zero)
   {
    Flock entity = GameObject.Find("FlockEntity").GetComponent<flock>();
    Vector3 velocity = entity.GetComponent<rigidbody>().velocity;
    sum += velocity;
   }
  }
  //float averageX = sumX / maxNeighbors;
  //float averageY = sumY / maxNeighbors;
  //float averageZ = sumZ / maxNeighbors;
  //Vector3 averageVelocity = new Vector3(averageX, averageY, averageZ);
  Vector3 averageVelocity = sum/maxNeighbors; //new Vector3(averageX, averageY,
```

```
averageZ);

  // Return a force vector pointing in the direction of the average velocity
  return (averageVelocity - transform.position).normalized * alignmentStrength;
 }

 public override void Update(Vector3 position)
 {
  // Find nearby entities within a certain radius
  float radius = 10f;

  int count = 0;
  for (int i = 0; i < maxNeighbors; i++)
  {
   Vector3 neighborPosition = position + Random.onUnitSphere() * radius;

   Flock entity = GameObject.Find("FlockEntity").GetComponent<flock>();

   if (entity != null && Vector3.Distance(position, neighborPosition) <= radius)
   {
    neighbors[count++] = neighborPosition;
   }

   if (count == maxNeighbors) break;
  }
 }
}
```

Example Use Case

To create a flocking game in Unity using this script, follow these steps:

1. Create a new scene and add a `FlockEntity` GameObject to it.
2. Attach the `Flock` script to the `FlockEntity`.
3. Add multiple instances of `FlockEntity` to the scene to create a flock.
4. Each `FlockEntity` should have its own instance of `Cohesion`, `Separation`, and `Alignment` scripts attached.
5. Run the game and observe how the flock behaves.

Tips and Variations

- You can adjust the strength of each behavior by modifying the `cohesionStrength`, `separationStrength`, and

`alignmentStrength` variables in the `FlockEntity`. * To add more variety to your flock, you can create different types of entities with unique behaviors. * Experiment with different values for `maxNeighbors` to see how it affects the flock's behavior. * Consider adding a visual representation for each entity (e.g., a circle or a square) to make the game more engaging.