Lab 3: Steering Behaviours & Simple Swarm

1. Introduction & Goals

In Lab 1 and Lab 2 you focused on the **high-level planning layer** of game AI:

- **Lab 1**: Finite State Machines + NavMesh – deciding *what state* an NPC is in and *where* to go.
- **Lab 2**: Graphs and A* – computing a *path* through the level (a global plan).

In this lab, you step down to the **movement layer**:

> Instead of just deciding **where** the agent goes, you will implement **how** it moves smoothly, using physics-like steering forces.

You will build a **"Steering Playground"** where multiple agents move and interact using classic **steering behaviours** (Seek, Arrive) and a simple **swarm rule** (Separation).

This lab is **self-contained** in a new scene.
If you do have Lab 1/2 working, there's an optional section where you plug steering into your existing pathfinding.

Learning Goals

By the end of this lab, you should be able to:

Conceptual

- Explain the difference between:
    - a **global plan** (path / waypoint list), and
    - a **local steering behaviour** (a force applied each frame).
- Describe what **Seek**, **Arrive**, and **Separation** do and when you would use them.
- Explain how simple local rules (like Separation) lead to more believable **group movement**.

Practical

- Implement a `SteeringAgent` that:
    - Computes **desired velocities** towards targets.
    - Computes **steering forces** and integrates them into **velocity and position**.
- Implement **Seek** and **Arrive** to control how an agent approaches a target.
- Implement **Separation** so a group of agents can move without overlapping.
- (Optional) Integrate steering with an existing **NavMesh** or **A*** path, so:
    - Pathfinding handles **where** to go.
    - Steering handles **how** to get there.

---

2. Prerequisites

You **do NOT** need a completed Lab 1 or Lab 2.
This lab takes place in a **new scene**.

You should:

- Have reviewed **Lecture 3 materials** on Steering and Boids / flocking.
- Be reasonably comfortable with:
    - Basic C# in Unity.
    - `Vector3` math:
        - `+`, `-`, `.normalized`, `.magnitude`, `.sqrMagnitude`, `Time.deltaTime`.

Nice-to-have but not required:

- Some intuition for basic physics:
    - **Force** → **Acceleration** → **Velocity** → **Position**.
      In this lab we'll treat **steering** as a force and integrate it over time.

---

3. Lab Structure

The lab is split into parts:

- **Part A – Setup: The Steering Playground**
  New scene, agent prefab, steering script, integration step.

- **Part B – Basic Steering: Seek & Arrive**
  One agent moves towards a target with Seek and then Arrive.

- **Part C – The Swarm: Multiple Agents & Separation**
  Spawn multiple agents and give them “personal space”.

- **Part D – Optional Challenge: Full Flocking**
  Add Cohesion and Alignment for full Reynolds-style flocking.

- **Part E – Optional: Integrate Steering with Pathfinding**
  If you already have NavMesh or A* paths, use steering to follow them.

Parts **A–C are required**.
Parts **D–E are optional challenges**.

Part A starts on next page ->

4. Part A – Setup: The Steering Playground

We'll build a small "AI zoo" to test movement.

A.1 Create the Scene

1. In your Unity project, create a new scene:
   `Scenes/SteeringPlayground`.

2. Add:

   - A **Plane** as ground.

   - A few **Cube** obstacles scattered around.

3. Position your camera to get a good view (top-down or 3rd person is fine).

4. Save the scene.

A.2 Create the Agent Prefab

1. Create a new object in the scene, e.g. a **Capsule**, and name it `Agent`.

2. **Physics note**:
   For this lab, we will simulate "physics" manually. You do **not** need a Rigidbody or CharacterController. We will update `transform.position` directly using our own velocity.

3. Turn `Agent` into a **Prefab** (drag it into a `Prefabs` folder).

A.3 The SteeringAgent Script

Create the folder `Scripts/AI/` and add a new script:

`SteeringAgent.cs`

Attach it to your `Agent` prefab.

Use this **template**, which already implements the **integration step** (force → velocity → position):

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

public class SteeringAgent : MonoBehaviour
{
    [Header("Movement")]
    public float maxSpeed = 5f;
    public float maxForce = 10f; // Limit how "fast" we can change
direction (turning radius)

    [Header("Arrive")]
    public float slowingRadius = 3f;

    [Header("Separation")]
    public float separationRadius = 1.5f;
    public float separationStrength = 5f;

    [Header("Weights")]
    public float arriveWeight = 1f;
    public float separationWeight = 1f;

    [Header("Debug")]
    public bool drawDebug = true;

    private Vector3 velocity = Vector3.zero;

    // Optional target for Seek / Arrive
    public Transform target;

    // Static list so agents can find each other
    public static List<SteeringAgent> allAgents = new
List<SteeringAgent>();

    private void OnEnable()
    {
        allAgents.Add(this);
    }

    private void OnDisable()
    {
        allAgents.Remove(this);
```

```
    }

    void Update()
    {
        // 1. Calculate Steering Force
        Vector3 steering = Vector3.zero;

        // TODO in Part B/C:
        // if (target != null)
        //     steering += Arrive(target.position, slowingRadius) *
arriveWeight;
        //
        // if (allAgents.Count > 1)
        //     steering += Separation(separationRadius,
separationStrength) * separationWeight;

        // 2. Limit Steering (Truncate)
        // This prevents the agent from turning instantly.
        steering = Vector3.ClampMagnitude(steering, maxForce);

        // 3. Apply Steering to Velocity (Integration)
        // Acceleration = Force / Mass. (We assume Mass = 1).
        // Velocity Change = Acceleration * Time.
        velocity += steering * Time.deltaTime;

        // 4. Limit Velocity
        velocity = Vector3.ClampMagnitude(velocity, maxSpeed);

        // 5. Move Agent
        transform.position += velocity * Time.deltaTime;

        // 6. Face Movement Direction
        if (velocity.sqrMagnitude > 0.0001f)
        {
            transform.forward = velocity.normalized;
        }
    }

    // -- BEHAVIOUR STUBS --
    public Vector3 Seek(Vector3 targetPos) { return Vector3.zero; }
```

```
    public Vector3 Arrive(Vector3 targetPos, float slowRadius) {
return Vector3.zero; }
    public Vector3 Separation(float radius, float strength) { return
Vector3.zero; }

    private void OnDrawGizmosSelected()
    {
        if (!drawDebug) return;

        Gizmos.color = Color.green;
        Gizmos.DrawLine(transform.position, transform.position +
velocity);
    }
}
```

Right now, your agent will not move. This is expected.
In Part B and C you will fill in `Seek`, `Arrive`, `Separation`, and the `Update()` logic.

Checkpoint A

- Scene exists.
- `Agent` prefab exists with `SteeringAgent` attached.
- Pressing Play does nothing interesting yet (no steering implemented).

---

5. Part B – Basic Steering: Seek & Arrive

We start with **one agent** moving towards a target.

B.1 Add a Target

1. Create a **Sphere** or Empty and name it `Target`.
2. Position it somewhere in front of the agent.
3. Select the `Agent` in the scene:
    - In the Inspector, assign the `Target` Transform to `SteeringAgent.target`.

B.2 The Math of Seek

Goal of Seek:
Move straight towards the target at **max speed**, without worrying about stopping.

The core idea (Reynolds' steering formula):

Steering Force = Desired Velocity – Current Velocity

- **Desired Velocity**:
  - A vector pointing from you to the target,
  - Normalized (direction only),
  - Then multiplied by `maxSpeed`.
- **Steering**:
  - The difference between *where you want to be heading* (desired) and *where you are currently heading* (velocity).

B.3 Implement Seek

Fill in the `Seek` method:

```
public Vector3 Seek(Vector3 targetPosition)
{
    Vector3 toTarget = targetPosition - transform.position;

    // If we are already there, stop steering
    if (toTarget.sqrMagnitude < 0.0001f)
        return Vector3.zero;

    // Desired Velocity: Full speed towards target
    Vector3 desired = toTarget.normalized * maxSpeed;

    // Reynolds' Steering Formula
    return desired - velocity;
}
```

Now **hook it up** in `Update()` (temporarily using Seek only):

```
void Update()
{
    Vector3 steering = Vector3.zero;

    if (target != null)
    {
        steering += Seek(target.position); // We'll replace this
with Arrive soon
    }

    steering = Vector3.ClampMagnitude(steering, maxForce);

    velocity += steering * Time.deltaTime;
    velocity = Vector3.ClampMagnitude(velocity, maxSpeed);

    transform.position += velocity * Time.deltaTime;

    if (velocity.sqrMagnitude > 0.0001f)
    {
        transform.forward = velocity.normalized;
    }
}
```

Checkpoint B1

- Press Play.
- The agent should accelerate towards the target, overshoot it, and orbit / loop around it.
  This is normal for pure **Seek** – it doesn't know how to brake.

Try moving the `Target` around while the game is running: the agent should chase it.

B.4 Implement Arrive

Goal of Arrive:
Move towards the target but **slow down** when close and **stop** at the target.

Arrive is almost identical to Seek, but with a **slowing radius**:

1. Compute distance to target.

2. If distance < slowingRadius, reduce desired speed proportionally to distance.

Fill in the `Arrive` method:

```
public Vector3 Arrive(Vector3 targetPosition, float slowingRadius)
{
    Vector3 toTarget = targetPosition - transform.position;
    float dist = toTarget.magnitude;

    if (dist < 0.0001f)
        return Vector3.zero;

    float desiredSpeed = maxSpeed;

    // Ramp down speed if within radius
    if (dist < slowingRadius)
    {
        desiredSpeed = maxSpeed * (dist / slowingRadius);
    }

    Vector3 desired = toTarget.normalized * desiredSpeed;
    return desired - velocity;
}
```

Now switch `Update()` to use **Arrive**:

```
void Update()
{
    Vector3 steering = Vector3.zero;

    if (target != null)
    {
        steering += Arrive(target.position, slowingRadius) *
arriveWeight;
    }

    steering = Vector3.ClampMagnitude(steering, maxForce);

    velocity += steering * Time.deltaTime;
    velocity = Vector3.ClampMagnitude(velocity, maxSpeed);

    transform.position += velocity * Time.deltaTime;

    if (velocity.sqrMagnitude > 0.0001f)
    {
        transform.forward = velocity.normalized;
    }
}
```

✅ **Checkpoint B2**

- The agent now **slows down** as it approaches the target and comes to rest nearby.
- Adjust `slowingRadius` to see how the braking distance changes.

---

6. Part C – The Swarm: Multiple Agents & Separation

Now we'll spawn **many agents** and give them **personal space** using Separation.

C.1 Agent Spawner

Create a new script `AgentSpawner.cs` and attach it to an empty GameObject in the scene.

```
using UnityEngine;

public class AgentSpawner : MonoBehaviour
{
    public SteeringAgent agentPrefab;
    public int agentCount = 10;
    public Vector2 spawnAreaSize = new Vector2(10f, 10f);

    void Start()
    {
        for (int i = 0; i < agentCount; i++)
        {
            Vector3 offset = new Vector3(
                Random.Range(-spawnAreaSize.x * 0.5f,
spawnAreaSize.x * 0.5f),
                0f,
                Random.Range(-spawnAreaSize.y * 0.5f,
spawnAreaSize.y * 0.5f)
            );

            Vector3 spawnPos = transform.position + offset;
            Instantiate(agentPrefab, spawnPos, Quaternion.identity);
        }
    }

    private void OnDrawGizmos()
    {
        Gizmos.color = Color.cyan;
        Gizmos.DrawWireCube(transform.position,
            new Vector3(spawnAreaSize.x, 0.1f, spawnAreaSize.y));
    }
}
```

- Assign your `Agent` prefab to `agentPrefab`.
- Set `agentCount` (e.g. 10–20).
- Place the spawner in the middle of your ground.

Checkpoint C1

- When you press Play, you see multiple agents.
- They currently all try to Arrive at the **same** target (or stand still if no target).

C.2 Implement Separation (Personal Space)

We want agents to **push away** from neighbours that are too close.

Logic:

For every other agent within `separationRadius`:

1. Calculate a vector **away** from them.
2. Scale it by how close they are (closer → stronger push).
3. Combine/average these pushes to form a final Separation steering force.

Fill in `Separation.` Try to implement it yourself before looking at the reference code on the next page!

```
public Vector3 Separation(float separationRadius, float
separationStrength)
{
    Vector3 force = Vector3.zero;
    int neighbourCount = 0;

    foreach (SteeringAgent other in allAgents)
    {
        if (other == this) continue;

        Vector3 toMe = transform.position -
other.transform.position;
        float dist = toMe.magnitude;

        // If they are within my personal space
        if (dist > 0f && dist < separationRadius)
        {
            // Weight: 1/dist means closer neighbours push MUCH
harder
            force += toMe.normalized / dist;
            neighbourCount++;
        }
    }

    if (neighbourCount > 0)
    {
        force /= neighbourCount; // Average direction

        // Convert "move away" direction into a steering force
        force = force.normalized * maxSpeed;
        force = force - velocity;
        force *= separationStrength;
    }

    return force;
}
```

C.3 Combining Forces (Weighted, Then Truncate Once)

In your `Update()` loop, calculate **each behaviour separately**, add them up, then clamp the total.

Example:

```
void Update()
{
    Vector3 totalSteering = Vector3.zero;

    // 1. Arrive (or Seek) towards target, if any
    if (target != null)
    {
        totalSteering += Arrive(target.position, slowingRadius) *
arriveWeight;
    }

    // 2. Separation: only if there are neighbours
    if (allAgents.Count > 1)
    {
        totalSteering += Separation(separationRadius,
separationStrength) * separationWeight;
    }

    // 3. Clamp total force (agents have finite strength)
    totalSteering = Vector3.ClampMagnitude(totalSteering, maxForce);

    // 4. Integration (same as before)
    velocity += totalSteering * Time.deltaTime;
    velocity = Vector3.ClampMagnitude(velocity, maxSpeed);

    transform.position += velocity * Time.deltaTime;

    if (velocity.sqrMagnitude > 0.0001f)
    {
        transform.forward = velocity.normalized;
    }
}
```

Checkpoint C2

- With `agentCount > 1`:
    - Agents move towards the target (or wherever you steer them).
    - They **don't sit on top of each other**.
    - When they get too close, they **push away** and spread out.

Experiment:

- Increase `separationRadius` → they leave more space.
- Increase `separationStrength` → they react more violently.
- Play with `arriveWeight` vs `separationWeight`:
    - High arrive, low separation → cluster tightly around target.
    - Low arrive, high separation → spread out more, loosely around target.

At this point, you've built a simple **swarm** that moves together without overlapping.

---

7. Part D (Optional Challenge) – Full Flocking

If you have the core swarm working and want more challenge, you can implement **Reynolds' Boids**:

- **Separation** – you already have this (avoid crowding).
- **Cohesion** – steer towards the centre of mass of neighbours.
- **Alignment** – match neighbours' heading/velocity.

You can iterate through neighbours just like you did in Separation.

D.1 Cohesion (Stay Together)

Goal: Move towards the **average position** of neighbours.

Logic:

1. Sum the positions of all neighbours within some radius.
2. Divide by count to get `averagePosition`.
3. Use your existing `Seek(averagePosition)` or `Arrive(averagePosition, someRadius)` to compute a steering force.

D.2 Alignment (Fly Together)

Goal: Match the **average velocity** (direction) of neighbours.

Logic:

1. Sum the **velocity** of all neighbours in range.
2. Divide by count to get `averageVelocity`.
3. Normalize this average, multiply by `maxSpeed` to get desired.
4. Steering = `desired - velocity`.

D.3 Combine with Weights

Add more weights:

```
public float cohesionWeight = 0.5f;
public float alignmentWeight = 0.5f;
```

Then in `Update()`:

```
if (allAgents.Count > 1)
{
    totalSteering += Separation(separationRadius,
separationStrength) * separationWeight;
    totalSteering += Cohesion(...) * cohesionWeight;
    totalSteering += Alignment(...) * alignmentWeight;
}
```

You don’t need this for lab completion, but it’s a nice playground for emergent behaviour and for your final project.

---

8. Part E (Optional) – Integrate with Pathfinding

If you already have **NavMesh** (Lab 1) or **A* paths** (Lab 2):

1. Compute a path as a `List<Vector3>` of waypoints.
2. Give that list to your `SteeringAgent`.
3. Treat each waypoint as a **mini target** and use **Arrive** to move from one to the next:
    - When distance to current waypoint < threshold, advance to the next.
4. Keep Separation enabled so multiple agents can follow paths without overlapping.

This shows the full stack working together:

> **Pathfinding** decides *where* to go globally.
> **Steering** decides *how* to move there locally.

You don't have to hand this in separately, but you can reuse it directly in your project.

---

9. Hand-in Instructions

Hand in on **Omniway**:

- A link to your **git repository**
  or
- A zip containing (with your name in the file name):
    - Your `Assets` folder (or entire Unity project if it's small).
    - A **short video or GIF** showing:
        - A single agent using **Arrive** at a target.
        - Multiple agents moving together with **Separation** active.
    - A brief **reflection file** (`Lab3_ReflectionYourName.txt` or `.md`) with answers to the questions below.

---

10. Reflection Questions

Answer these with a few sentences each:

1. In your own words, what is the difference between:
    - A **global path / waypoint list**, and
    - A **local steering behaviour** like Seek or Arrive?
2. What visual difference did you notice between:
    - An agent using **Seek**, and
    - An agent using **Arrive**?
3. How did **Separation** change the behaviour of your group?
    - What happens if you set `separationStrength` very low? Very high?
4. (If you tried flocking or path integration)
   What new behaviours did you see when you:
    - Added Cohesion / Alignment, or
    - Combined steering with NavMesh / A* paths?
5. Looking ahead to your **final project**:
    - Name at least one NPC, enemy, or unit that could use this `SteeringAgent`.
    - How might you combine **steering** with your **FSM** or **pathfinding** in that project?