FlockingDead

By Kamiel de Visser | 500838438

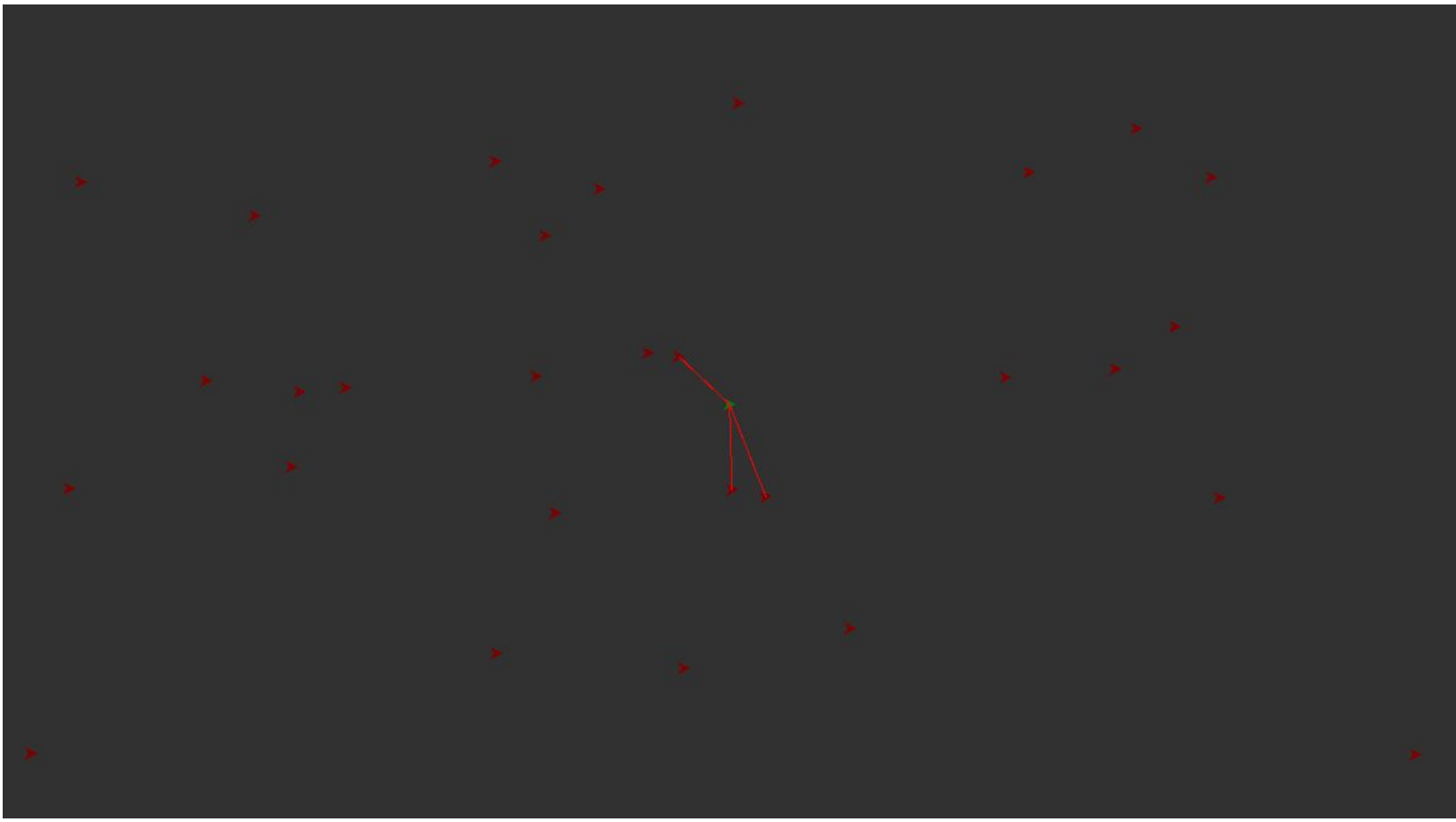


Table of contents

Introduction

In this document I will describe my thought process whilst making the FlockingDead assignment.
I will do this on a step by step basis.
We were provided a start project in which we had to implement a few artificial intelligence concepts.

This assignment was done during the Artificial Intelligence course at the HvA.

The assignment

Question 1

Finish the `Hunt` function in the `Agent` class, such that zombies will chase after other agents.

The original function is as followed:

```
private void Hunt(List<Agent> agents)
{
    float range = float.MaxValue;
    Agent prey = null;
    foreach (Agent a in agents)
    {
        if (!a.isZombie)
        {
            float distance = Distance(position, a.position);
            if (distance < sight && distance < range)
            {
                range = distance;
                prey = a;
            }
        }
    }
    if (prey != null)
    {
        // Move towards prey.
        // dX += TODO
        // dY += TODO
    }
}
```

First I implemented standard ('dumb') movement, for this I just take the directional vector between the zombie and the prey, and normalize it. I then multiply that normalized vector with the movement speed. This results in the sought after `dX` and `dY`. Note that these values are then clamped to adjust for the fact that zombies are slower than their prey (in `CheckSpeed()`).

```
// Move towards prey.
Vector2 direction = (prey.transform.position - transform.position).normalized;
Vector2 deltaPos = movementSpeed * direction;
dX += deltaPos.x;
dY += deltaPos.y;
```

To make our zombies move a little smarter, I'll implement a (still pretty 'dumb') version of movement prediction. I want our zombies to predict where their prey will be going, and adjust their direction accordingly.

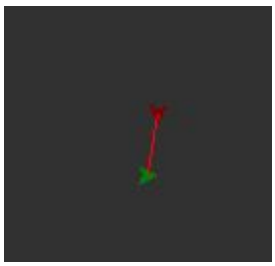
For this each zombie will crudely guess where their prey will end up being by the time the zombie gets to them. They do this by first calculating the distance between them and the prey (`distanceToPrey`). Then they will guess how long it will take them before they would get to the prey (`timeToPrey`) (should the prey not move in the meanwhile).
Knowing the time it would take for the zombie to travel to a still prey, the zombie then first checks how fast the prey is moving (`preyCurrentSpeed`), and calculates where the prey would end up should they not change direction, stay at their current speed, and move for the duration it would take the zombie to get to the prey were they to stay still.
I draw a debug ray to see what the predicted location of a prey is for each zombie.
Finally, the zombies move to the predicted location.

```
// Guess where prey will be
float distanceToPrey = Vector2.Distance(transform.position, prey.position);
float timeToPrey = distanceToPrey / (movementSpeed / 3f);
float preyCurrentSpeed = new Vector2(prey.dY, prey.dY).magnitude;
Vector3 predictedPreyPosition = prey.transform.position + preyCurrentSpeed *
    movementSpeed * prey.transform.right * timeToPrey;

// Move towards prediction
Debug.DrawLine(this.transform.position, predictedPreyPosition, Color.red);
Vector2 direction = (predictedPreyPosition - transform.position).normalized;
Vector2 deltaPos = movementSpeed * direction;
dX += deltaPos.x;
dY += deltaPos.y;
```

Since the zombies base the predicted position of the prey off of the time it would take to get to their prey if the prey didn't walk, these guesses are quite crude. But since the guesses are done each and every time the agents move, this movement method results in smarter pathing for the zombies, as they now move towards where their prey will (approximately) be at the time they even get close, instead of always chasing to the exact location of the prey.

As you can see in this example the zombies (red units) will now directly move towards non-zombies (green units) who stand still:



But zombies also move towards their predictions of where the non-zombie units will be, if the non-zombie units are moving.

Question 2

Finish the `Evade` code -inside the `Flock` function -in such a way that agents will try to evade zombies.

The Flock function looks as followed:

```
private void Flock(List<Agent> agents)
{
    foreach (Agent a in agents)
    {
        float distance = Distance(position, a.position);
        if (a != this && !a.isZombie)
        {
            if (distance < sight)
            {
                // Cohesion
                //dX += TODO
                //dY += TODO
            }
            else if (distance < space)
            {
                // Separation
                dX += (position.x - a.position.x);
                dY += (position.y - a.position.y);
            }
            if (distance < sight)
            {
                // Alignment
                //dX += TODO
                //dY += TODO
            }
        }
        if (a.isZombie && distance < sight)
        {
            // Move away from spotted zombie.
            Vector2 direction = (a.transform.position -
transform.position).normalized;
            Vector2 deltaPos = movementSpeed * - direction;
            dX += deltaPos.x;
            dY += deltaPos.y;
        }
    }
}
```

Evasion is just the reverse of chasing. So we take the simple movement code from Q1, but make sure we move *away* from the direction from the non-zombie to the zombie this time instead of *along* it.

```
if (a.isZombie && distance < sight)
{
    // Move away from spotted zombie.
    Vector2 direction = (a.transform.position - transform.position).normalized;
    Vector2 deltaPos = movementSpeed * - direction;
    dX += deltaPos.x;
    dY += deltaPos.y;
}
```

Since this evasion effect is quite 'unfair' to the zombies (the zombies already move 3x slower than the non-zombies do, and therefore each and every non-zombie agent will move out of a zombie's sight quite instantaneously, giving them almost no way to catch up), we level the playing field a little.

I'll give zombie agents more sight than non-zombie agents (and explain that away by the 'fact' that our zombies have heightened senses).

```
private float sight = 100f;
private static float zombieSight = 150f;

private void Initialise()
{
    ...
    if (isZombie)
    {
        sight = zombieSight
    }
    ...
}

private void BecomeZombie()
{
    ...
    sight = zombieSight;
}
```

And also let zombies move a little faster (in `CheckSpeed` and the code of **Q1** we ensure `movementSpeed` is no longer divided by 3 for zombies, but by 2.

Question 3

Add a parameter to the `Flock` function to be used as a weight for separation. Use it to scale the values separation adds to the velocity (`dX` and `dY`).

First I'll add a new public variable to our Agent class called `seperationWeight` so Ican set and test the `seperationWeight` by editing it in the Unity editor.
Then I'll add a parameter with the same name to the Flock function. (Arguably, since it's already added to the class as a variable I may have as well skipped this step since the variable would be available in the function either way).

```
private void Flock(List<Agent> agents, float seperationWeight) { ... }
```

And then we scale the values added to `dX` and `dY` by multiplying them with `seperationWeight`.

```
else if (distance < space)
{
    // Separation
    dX += seperationWeight * (position.x - a.position.x);
    dY += seperationWeight * (position.y - a.position.y);
}
```

Question 4

Explain why separation is important for flocking.

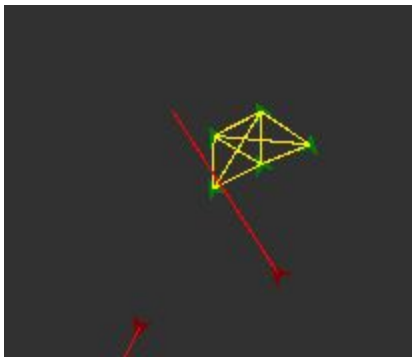
Separation is important for flocking as it makes sure agents don't bump into each other. If you're taking a stroll through the *Kalverstraat*, you'll notice that people tend to keep some distance from one another, so as to give each other a little space. Implementing separation therefore makes the movement of your agents more believable, it gives it a more natural feel. It's hard to immerse yourself in a game where every character keeps bumping into or passing through one another.

Question 5

Implement cohesion. Use another parameter as a weight for cohesion. Use it to scale the values cohesion adds to the velocity.

To implement cohesion, I'll need to write a piece of code that's more complex than the separation.
Cohesion works by looking at the positions of every neighbouring agent, and then determining their average position. If an agent then moves towards this position, it'll move closer towards the group.

First, in the `foreach` loop (which is called for every agent in `Swarm.cs`) I check if there are any other agents close enough to call for cohesion, but far away enough that the unit's don't need to separate yet.
If an agent is in range for cohesion, we increment `numCohesionNeighbours` and add the directional vector towards that agent to `cohesionDirections`.
After we're done checking every agent, we then look at the average of all those directions (`cohesionDirections / numCohesionNeighbours`), giving us the average direction this agent should head to to end up at the average position of every found agent.
For debugging sake, we also draw a yellow debug line which can show us which units are enacting cohesion.
Finally, we move towards that direction with a factor of `cohesionWeight`.

In this example you can see a group of units grouping together, but also keeping their positions apart. This was achieved with a separation weight of 0.5 and a cohesion weight of 0.02	

```
private void Flock(List<Agent> agents, float seperationWeight, float
cohesionWeight)
{
    int numConhesionNeighbours = 0;
    Vector3 cohesionDirections = Vector2.zero;
    foreach (Agent a in agents)
    {
        float distance = Distance(position, a.position);
        if (a != this && !a.isZombie)
        {
            if (distance < space){ // Separation }
            else if (distance < sight)
            {
                numConhesionNeighbours++;
                Debug.DrawLine(a.transform.position, this.transform.position,
                               Color.yellow);
                cohesionDirections += a.transform.position -
                                      this.transform.position;
            }
            if (distance < sight) { // Alignment }
        }
        if (a.isZombie && distance < sight) { // Evasion }
    }
    if (numConhesionNeighbours != 0)
    {
        Vector3 averageDirection = (cohesionDirections /
                                    numConhesionNeighbours).normalized;

        Vector2 deltaPos = averageDirection * cohesionWeight;
        dX += deltaPos.x;
        dY += deltaPos.y;
    }
}
```

Question 6

Explain why cohesion is important for flocking.

Cohesion is important to implement when implementing flocking to ensure your flocking algorithm can support more interesting emergent behaviour. Cohesion ensures that your units don't just keep wandering around aimlessly. If an agent finds other units closeby, the agents will try to move towards each other, providing the user the illusion that each agent prefers to be near other agents.
It ensures agents try to stay together as a group.
If each unit keeps running away from their squad in a game like *StarCraft*, the game becomes very unmanageable.

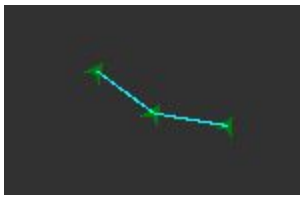
Question 7

Implement alignment. Use another parameter as a weight for alignment. Use it to scale the values alignment adds to the velocity.

For alignment I used practically the same setup as cohesion. The only real difference is that this time, instead of looking for the average position of agents around an agent, we want to know the average velocity of said agents, and adjust the 'source' agent's velocity accordingly.
Alignment is shown by cyan debug lines.

```
numAlignmentNeighbours++;
Debug.DrawLine(a.transform.position, this.transform.position, Color.cyan);
alignmentVelocities += new Vector2(a.dX, a.dY);
```

```
if (numConhesionNeighbours != 0)
{
    ...
}
if (numAlignmentNeighbours != 0)
{
    Vector3 averageVelocity = (alignmentVelocities /
                               numAlignmentNeighbours).normalized;
    Vector2 deltaPos = averageVelocity * alignmentWeight;
    dX += deltaPos.x;
    dY += deltaPos.y;
}
```

In this example you can see a group of agents all aligned and moving in the same direction.	

Question 8

Explain why alignment is important for flocking.

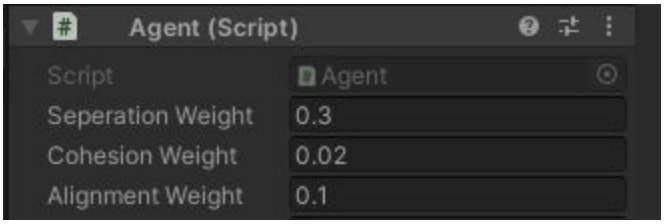
Alignment (once again) provides more emergent behaviour to a Flocking algorithm. The purpose of alignment is to ensure all agents in a group move somewhat along towards the same direction.
For example, a group of soldiers in *Rome: Total War* would look off if they were all looking at each other. If they're all facing and moving towards the same direction, they look a lot more organised.

Question 9

Using trial and error testing, find out what parameter values for separation, cohesion and alignment work well to create the scenarios below. If necessary, change the number of zombies or their speed to make them faster or slower. Explain (for each scenario) your choice of parameters using screenshots, and describe how you achieved the desired effects.

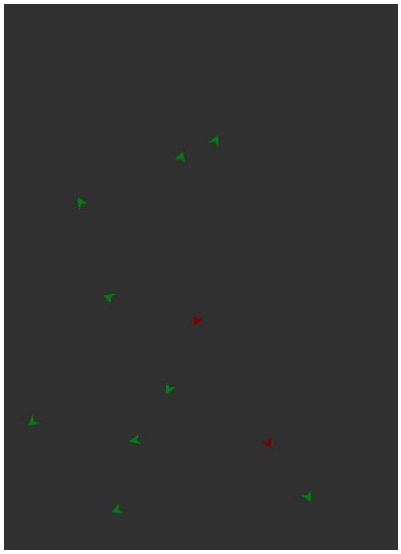
I made no further changes other than the changes noted in **Q2**, or stated here otherwise.

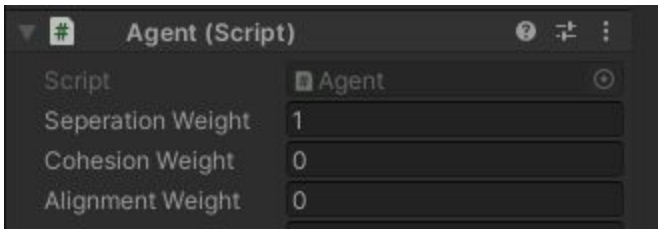
a)Agents crowd together in fear of the zombies. Make sure the agents do not all overlap each other, while they crowd together



Agents try to group and run away from zombies together. They never overlap due to an influence from the separation, group together thanks to the (small) cohesion weight, and move together thanks to the (small) alignment weight.

b)Agents spread out, making it harder for the zombies to catch them.





Agents do not align or cohese in this scenario. All they do is run away from the closest zombie. These parameters ensure the agents never collide but they can still come pretty close together by chance. However, they aren't grouping like they would with alignment or cohesion turned on.

c) A single agent is trying to outrun 14 very slow zombies.

Since this scenario deals with just 1 agent, seperation, cohesion and alignment will not be a factor.

To make sure we encounter this scenario more easily, we alter Swarm.cs to spawn just a single non-zombie, and make the rest of the agents zombies.

We also increase the zombieSight variable to 300, making zombies a lot more aware of the non-zombie agent.