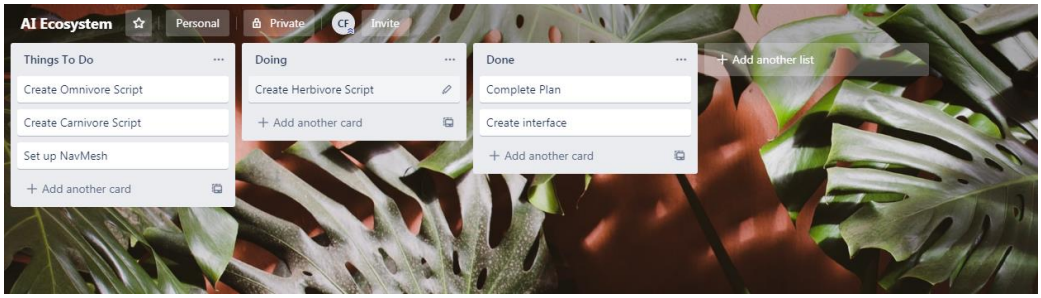
AI Ecosystem Documentation Created by Charlie Frost

Before I begin documenting the whole production process, please note that I have chosen to complete this project without following any complete tutorials; If I get severely stuck on any aspect of this project, I may need help, however that will be documented also. I have realised I do not do this enough and feel this way of working may help improve my thinking patterns in the future. It will also help me gain experience in using inheritance (especially polymorphism) and interfaces (if I end up using them, which I believe is likely!).

Note: This was a crucial learning process for me in terms of practical and theoretical research surrounding inheritance/composition/aggregation etc. I am aware that there were many mistakes made and have tried my best to include them all in the documentation process. For instance, for a lot of this document I use inheritance incorrectly, however correct myself and talk about it as necessary. Finally, if I mention simple/obvious things such as the Update function being called every frame, it is probably for my own benefit/anyone who is learning to code that wants to read this document.

Planning Section

Due to the nature of this project I need to have a good plan. Many things could be missing or be completed in the wrong order if not. For timekeeping sake and to avoid becoming confused at any point, I am using Trello to manage my tasks:

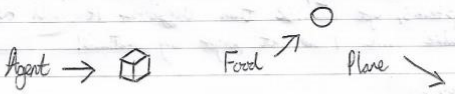


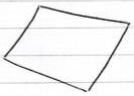
Here are the rough plans for my project:

Charlie Frost

AI Ecosystem Planning

Main rundown: There will be AI agents on a flat plane competing for food. There will be lots of AI running around, going for randomly spawning food. Upon eating food they will grow larger.



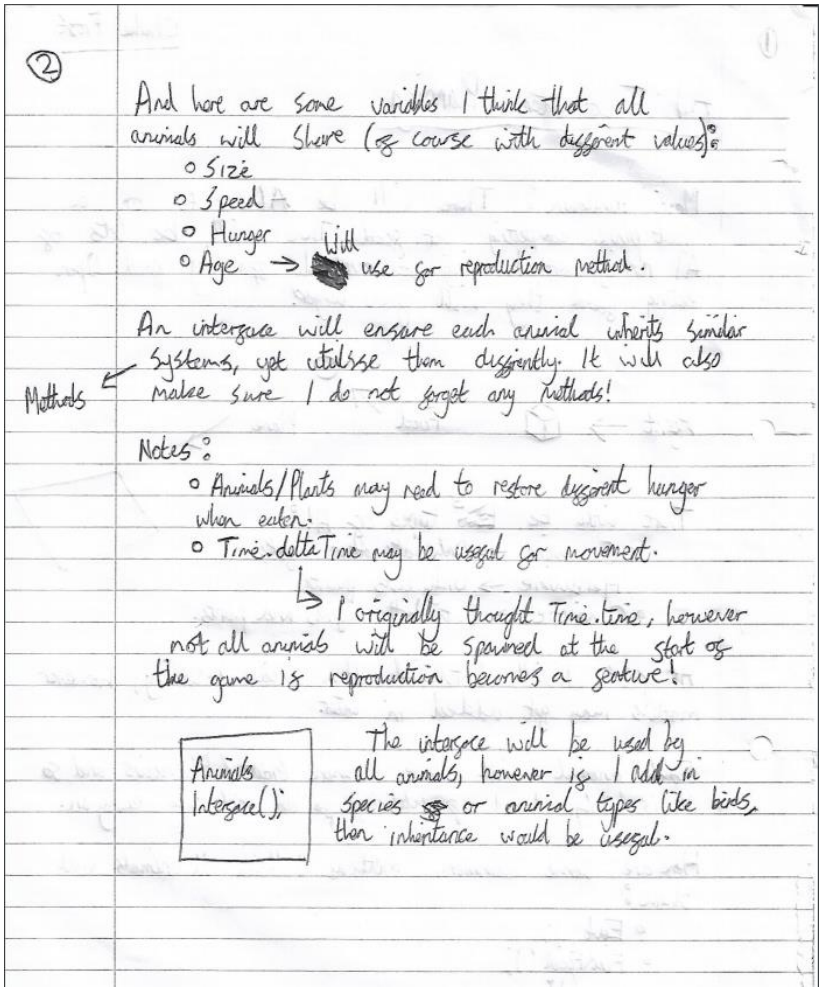


There will be ~~two~~ 3 types of AI:
- Carnivore → Only eats other agents
- Herbivore → Only eats plants.
- Omnivore → Eats both agents and plants.

At first I will distinguish them via colouring, however models may get added in later.

Each animal will share common traits / behaviours and so it is likely I end up utilising an interface early on.

Here are some common methods I think all animals will share:
- Eat();
- FindFood();
- Move();
- Grow();
- Reproduce();



Blogging Section

The first thing I have done is to create the ground. I have simply created a plane object and added a grass texture that I downloaded from the Unity Store. Next up is to create the interface 'Animals'. It will contain all the methods that EVERY animal should be able to have access to:

```
interface AnimalsInterface
{
    //methods
    void FindFood();
    void Move();
    void Eat();
    void Grow();
    void Reproduce();
}
```

Next up I am going to create the food that the planteaters will eat. I have made sure to make it's collider a trigger so they can later be picked up. I then created tags for both the ground and the food and assigned them as necessary. I also moved the camera ready for game view for later testing.

The first AI type script I am going to create is the Herbivore script. This is because there are clearly less features to implement than in that of the Omnivore's or Carnivore's scripts. I first inherit from the interface I just made and I am ready to start coding!

I got a few minutes into starting my code, then realised it would be much easier using Herbivores as a super class. This is because the Omnivores and Carnivores will be utilising all of the Herbivore's methods, along with some extra ones (therefore inheritance is extremely useful/important). The interface is still useful however because it stops me from forgetting methods that I would have possibly forgotten otherwise.

Here is the start of the Herbivores script:

```
public class Herbivores : MonoBehaviour, Animals
{
    //variables
    public float size;
    public float speed;
    public float hunger;
    public float age;
    public float ageRate;


    void Start()
    {
        age = 0;
    }

    void Update()
    {
        age += (ageRate * Time.deltaTime);
    }

    public void Eat()
    {
        throw new NotImplementedException();
    }

    public void FindFood()
    {
        throw new NotImplementedException();
    }

    public void Grow()
    {
        throw new NotImplementedException();
    }

    public void Move()
    {
        throw new NotImplementedException();
    }

    public void Reproduce()
    {
        throw new NotImplementedException();
    }
}
```

As you can see it inherits from MonoBehaviour and our interface 'Animals'. The base variables are declared and the main methods are there ready for me to create. I also added a very simple system for age which works as intended for the time being.

I then had another change of plan and realised it is much easier to work with a creature that searches for food, when the food has actually spawned! I created a simple spawner that spawns a 'food' prefab on a timer:

```
using UnityEngine;

public class FoodSpawner : MonoBehaviour
{
    // Variables
    // Spawn rate
    public float foodSpawnRate = 2f;
    // Time until next spawn
    public float nextFoodSpawn = 1f;


    // Food prefab I want to spawn
    public GameObject Food;


    void Update()
    {
        // If it is time to spawn food
        if (ShouldSpawnFood())
        {
            SpawnFood();
        }

    }

    // This function spawns the food
    // It then resets the nextFoodSpawn to the foodSpawnRate
    // And instantiates the food in a random location
    private void SpawnFood()
    {
        nextFoodSpawn = Time.time + foodSpawnRate;
        Instantiate(Food, transform.position, transform.rotation);
    }

    // This method returns if time has surpassed the next food spawn
    private bool ShouldSpawnFood()
    {
        return Time.time > nextFoodSpawn;
    }
```

This works fine; however I want to adapt it quite a bit, in order to spawn the food in random locations on the plane and ideally assign them in a list so that the ‘Searching’ method will be easier to create. At first, I thought this would be simple but after trying a few different methods I tried the following:

```
void Update()
{
    // If it is time to spawn Food
    if (ShouldSpawnFood())
    {
        SpawnFood();
    }

    AddFoodToFoodsList(Food);
}

// This method returns if time has surpassed the next Food spawn
private bool ShouldSpawnFood()
{
    return Time.time > nextFoodSpawn;
}

// This function spawns the Food
private void SpawnFood()
{
    // Instantiate the Food
    Instantiate(Food, transform.position, transform.rotation);

    // Reset the nextFoodSpawn to the foodSpawnRate
    nextFoodSpawn = Time.time + foodSpawnRate;
}

private void AddFoodToFoodsList(GameObject Food)
{
    if (!Foods.Contains(Food))
        Foods.Add(Food);
}
```

The problem I was having is that I want to instantiate my objects using the SpawnFood method (which works), however I only want to add each food item to the list once. Finding objects to add by tag adds them all every time and so I tried to use the Boolean above to prevent this happening. The problem now is that ANY Food object only gets added once. After much deliberation I had an epiphany:

```
// Instantiate the Food
var thisFood = Instantiate(Food, transform.position, transform.rotation);

if (!Foods.Contains(thisFood))
    Foods.Add(thisFood);
```

The solution was much simpler than I first thought. I have simply named the instantiated object 'thisFood' and then checked if that variable had been added to the list!

I then added an additional check for food spawning – the if statement now reads "if (ShouldSpawnFood() && Foods.Count < maxFood)". This will always stop the scene from going over the maximum food count. I then changed the spawning points of the Food to use Random.Ranges (this may end up being a placeholder if I improve the spawning code later).

Now I have Food spawning I have something to work with. Back to the Herbivore code! I am going to work on the 'FindFood' method, as the food is now spawning properly – this seems to be the next logical step.

```
Transform GetClosestFood(List<Transform> Foods, Transform fromThis)
{
    Transform bestTarget = null;
    float closestDistanceSqr = Mathf.Infinity;
    Vector3 currentPosition = fromThis.position;
    foreach (Transform potentialTarget in Foods)
    {
        Vector3 directionToTarget = potentialTarget.position - currentPosition;
        float dSqrToTarget = directionToTarget.sqrMagnitude;
        if (dSqrToTarget < closestDistanceSqr)
        {
            closestDistanceSqr = dSqrToTarget;
            bestTarget = potentialTarget;
        }
    }
    return bestTarget;
}
```

After a lot of coding and reading on the mathematical side of things, I have ended up with this code. This basically gives us the transform of the 'bestTarget' i.e. the closest Food. There is a problem, however, when I try to utilise this search within a function:

```
public void FindFood()
{
    closestFood = GetClosestFood(FoodSpawnerScript.Foods, this.transform);
}
```

Of course, the 'GetClosestFood' Transform returns exactly that, a transform! Therefore we cannot use the 'GameObject' list. I was struggling with fixing these issues for a few days and so in the end I had to get help from a forum:

https://answers.unity.com/questions/1682499/trying-to-put-items-in-an-updating-list-into-an-ar.html

I have ended up using a List as I first intended, as the problems were not related to having to use an array. I got confused with accessing global and local GameObject variables and have now fixed the issue. To help fix the issue, there were some debugging lines added:

```
// Debug to find how many food items are in the List
Debug.Log("Foods contains " + FoodSpawnerScript.Foods.Count + " Items");

// Debug to find how many food spawners there are
Debug.Log("FoodSpawners Found: " + FindObjectsOfType<FoodSpawner>().Length);
```

These were checking that the Herbivore had a List and a script to access. I have also reduced the animal's speed. There were also a few other bits of code added:

```
// Makes sure the we have access to the Food Spawner's script
if (FoodSpawnerScript == null)
    FoodSpawnerScript = FindObjectOfType<FoodSpawner>();
```

This checks that we are finding the script on the FoodSpawner gameObject.

```
// The FindFood method calls for the closestFood item using the ChooseNearestFood routine
public void FindFood()
{
    // Set the closestFood item using the ChooseNearestFood routine
    closestFood = ChooseNearestFood();
}
```

The FindFood method now sets the closestFood item using the ChooseNearestFood routine.

```
// Delete any Food the animal comes into contact with
void OnTriggerEnter(Collider other)
{
    if (other.tag == "Food")
    {
        Destroy(other.gameObject);

    }
}
```

I have also removed the Eat() method from both the animals interface and the herbivore script, as OnTriggerEnter will suffice instead and serve exactly the same purpose.

Finally, I have added this bit of code (the final line):

```
void Update()
{
    // If it is time to spawn Food
    if (ShouldSpawnFood() && Foods.Count < maxFood)
    {
        SpawnFood();
    }

    Foods.RemoveAll(item => item == null);
}
```

This is within the FoodSpawner script and makes sure that after a Food item has been eaten it is removed from the Foods List.

I realised I was receiving an error message along the lines of “The GameObject you are trying to access cannot be reached as It has been destroyed”. This was displaying an additional time every frame after the animal had eaten a Food. To fix this, I have added in another if statement just after my ‘foreach’ loop:

```
// Go through each Food item in the Foods List
foreach (GameObject Food in FoodSpawnerScript.Foods)
{
    if (Food != null)
    {
        // Work out the distance from each Food item to the animal
        float distToFood = (Food.transform.position - this.transform.position).sqrMagnitude;
        // If the distance is smaller than the distance to the current closest Food
```

This was successful and now the console is completely clear during runtime.

The next thing to work on is the Grow() method, which will increase the animal’s size by a small amount when they eat Food. I have added a Rigidbody to the animal, disabled it’s use of gravity and set it to kinematic, so it does not fly off when size is increased (via hitting the ground).

I have achieved the simple desired effect by using this simple method:

```
// The Grow method simply increases the animal's localscale
public void Grow()
{
    // Amounts to grow by:
    float growthX = 0.05f;
    float growthY = 0.05f;
    float growthZ = 0.05f;

    // Increase localScale by growth amounts
    transform.localScale += new Vector3(growthX, growthY, growthZ);
}
```

I simply increase the localScale by some custom growth amounts (floats).

This is now called here:

```
// Delete any Food the animal comes into contact with
void OnTriggerEnter(Collider other)
{
    if (other.tag == "Food")
    {
        Destroy(other.gameObject);
        Grow();
    }
}
```

Before moving any further I have added a public float ‘hungerRate’ and utilised it in exactly the same way I have used ‘ageRate’. Now the animal’s hunger starts at 0 and increases over time. I have added another condition to this if statement, so that the animal only eats when they are hungry (currently set to 5).

```
// Go through each Food item in the Foods
foreach (GameObject Food in FoodSpawnerScr
{
    if (Food != null && hunger > 5)
    {
```

Just before the animal grows, their hunger is set back to 0 so they do not eat again immediately:

```
// The Grow method simply increases the animal's localscale
public void Grow()
{
    // Set the animal's hunger back to 0 before growing
    hunger = 0f;

    // Amounts to grow by:
```

This is working as intended and now there is a brief period between eating the Food items and moving where the animal stops to recharge their hunger.

I have also created a flat called ‘ageLimit’ and added this code into the animal’s Update function:

```
// Animal (sadly) dies when they reach a certain age
if (age >= ageLimit)
    Destroy(this.gameObject);
```

This ensures the animal is deleted after a certain time period. This is important because some animals may become HUGE is no other animal eats them; an ageLimit prevents an animal becoming too large/powerful.

Next up is the Reproduce method. I have added it to the Start function just to test if it works. Here is the initial code:

```
public void Reproduce()
{
    // Spawn a child animal just to the right of this animal
    GameObject childAnimal = Instantiate(Herbivore, new Vector3(transform.position.x + 0.5f, transform.position.y, transform.position.z), transform.rotation);
}
```

As you can see, this is identical to how I have spawned my Food prefabs in.

As the condition for reproducing, I have awarded the animal a 'reproductionPoint' every time they grow. When this reaches 10, the baby animal of the same type will be instantiated just to the right of the parent animal. This is achieved by an if statement within the Update function.

Now I have the core of the Ecosystem working, I want to refine how the system is working before creating my other types of animals. Firstly, the pace of the ecosystem is rather slow, and so I am going to change the hunger system. Rather than slowly increasing the animal's hunger and eating when they are hungry, I am going to do the opposite; the animal will continue to gain hunger; however, they will always be eating, and will die whenever their hunger reaches a certain value, just like in real life. To achieve this, I have removed the 'if hunger > 5' for finding food and added in an additional condition inside the following if statement: 'if (age >= ageLimit || hunger >= hungerLimit)'. hungerLimit is just a new public float I have created and have set to 20f.

When testing, I realised I needed to reset reproductionPoints at the start of the Reproduce() method and so I have done that. I have also increased the size of the plane (ground).

Next up, I want to make sure that the animals do not clip through each other, especially when instantiated. To do this, I have increased the range from which the animals are instantiated from. I have also tampered with the Rigidbody on the animal to remove ALL sliding whilst waiting for food to spawn. By removing the 'is Kinematic' and increasing both the mass and the drag of the animals to '9999', there is no unwanted movements from the animals.

Another small change I have done is change 'foodSpawnRate' from a private variable to a public one and increased the speed at which food is spawning. Then I created some materials for present and future use (literally just colouring tools) and added in the current ones:

This will allow me to distinguish between animal types later. Now I have all the methods of my Herbivore working, I can start creating Carnivores.

At this point I started researching inheritance and started thinking about how I was going to start creating the Carnivore script. It came to my attention that the concept of inheritance denotes the 'is-a' relationship. Clearly, in this case, a carnivore is not a herbivore and a meat eater combined, they are simply meat eaters. As a result, I have decided to scrap these

plans to use inheritance and looked at using composition where necessary. This is because the concept of composition denotes 'is-a-part-of' instead, and is much more appropriate, as the Carnivore script will use a few (but not all) of the methods inside of the Herbivores script. However, composition works in such a way that if the parent object is deleted, then the child object is also deleted – this is clearly not what we want.

This is the point in which I had a huge realisation – my initial plan was not correct. Indeed, I could still use an animal interface, however I should've had an animal class to begin with, rather than trying to inherit 'some' methods from the Herbivores class. To aid my research, I mainly read this article as well as multiple forums / observed some UML diagrams:

https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/

Therefore, I am now going to document some changes to the existing code.

In the end, I have decided to alter my current Herbivores class into an Animal class (it is nearly already there). This is due to the fact that herbivores can inherit this class and are practically (if not) already completed, whereas Carnivores and Omnivores can simply override the 'FindFood' method and add their own list of prey. I am still using inheritance rather than aggregation because almost all (if not all) of my Animal's methods will be used in the different types of animal's scripts.

To correct my Herbivores class, I have set only the 'Reproduce', 'FindFood' and 'Move' methods to virtual methods. This is because they each contain some code that is specific to the Herbivore GameObjects and will need to be altered for the other 2 types of animals.

I have also added in 2 additional GameObjects as necessary:

```
// Allows us to reproduce
public GameObject Herbivore;
public GameObject Carnivore;
public GameObject Omnivore;
```

This may be slightly lazy and I could've just added them to the individual sub class scripts, however the project is so small that I don't believe it matters too much (as long as I am aware it may not be optimal!).

As well as setting the 'GameObject ChooseNearestFood' to 'public virtual GameObject ChooseNearestFood' as this may also need to be altered.

At this point I have not opted to make any/many protected members as I am not going to add many more scripts to the project – this should not be an issue.

I have created a new script called 'Herbivore' (getting deja vu here!) and removed all code except for the Start, Update and OnTriggerEnter functions. Of course, without these, the

script would do nothing. Since the Herbivore class inherits everything else it needs from the Animal class, I simply had to swap the new script back onto the Herbivore prefab and add it's GameObject references in the inspector, now the script works as intended.

Next onto the Carnivore script. First, I copy over all the code from my Herbivores script again, (Start, Update and OnTriggerEnter functions) and then all of the functions I set as 'virtual' in the base script. Since the logic behind altering this script is easy, I can start basically anywhere; as long as I convert everything to do with Food into Animals, it will be ok.

Of course, before I begin overriding these methods, I need to deal with creating a list of animals that carnivores/omnivores can eat. I could do this within each animal type's script, but this would only complicate things.

To begin solving this issue I have created an Empty GameObject called 'AnimalCounter' and given it the tag 'AnimalsCounter'. I also created a new script called 'AnimalsCounter'. I have created a 'public AnimalsCounter AnimalsCounter' inside the super class 'Animals'. This will give all the subclasses access to the script for when they want to add their instantiated GameObjects into the appropriate Lists.

Before carrying on I removed unnecessary code from the 'Animals' script. The code was the 'Start', 'Update' and 'OnTriggerEnter' functions. I did this because they are not being used by any GameObjects and so were pointless.

Next, I got the code in place for the AnimalsCounter script:

```
public class AnimalsCounter : MonoBehaviour
{
    // Variables

    // Each list of animal types:
    public List<GameObject> HerbivoresCounter = new List<GameObject>();
    public List<GameObject> CarnivoresCounter = new List<GameObject>();
    public List<GameObject> OmnivoresCounter = new List<GameObject>();




    // Functions to add animals to their lists:
    // Add Herbivores to their List
    public void AddHerbivoreToList(GameObject HerbivoreObject)
    {
        // If the Herbivore that has just been instantiated is not already in the list then add it
        if (!HerbivoresCounter.Contains(HerbivoreObject))
            HerbivoresCounter.Add(HerbivoreObject);
    }

    // Add Carnivores to their List
    public void AddCarnivoreToList(GameObject CarnivoreObject)
    {
        // If the Carnivore that has just been instantiated is not already in the list then add it
        if (!CarnivoresCounter.Contains(CarnivoreObject))
            CarnivoresCounter.Add(CarnivoreObject);
    }

    // Add Omnivores to their List
    public void AddOmnivoreToList(GameObject OmnivoreObject)
    {
        // If the Omnivore that has just been instantiated is not already in the list then add it
        if (!OmnivoresCounter.Contains(OmnivoreObject))
            OmnivoresCounter.Add(OmnivoreObject);
    }


}
```

Here, the lists of each animal types are created, and so are the functions in which the reproduced animals are added. Of course, the initial prefab is not added. It may seem lazy not correcting this, however if I did, the initial Herbivore could be eaten by the initial Carnivore immediately and so it turns out to be necessary! Within these functions the same check that I used in the FoodSpawner script is applied. I have also added in the Lambda expressions (something else I've learned about through this project) to remove any animals that have been eaten from each List:

```
void Update()
{
    // Constantly remove all eaten animals from their Lists (if they are null, they have been eaten)
    HerbivoresCounter.RemoveAll(item => item == null);
    CarnivoresCounter.RemoveAll(item => item == null);
    OmnivoresCounter.RemoveAll(item => item == null);
}
```

To actually add the animals into their respective Lists, I have used the following code, inside an overridden version of the Reproduce function:

```
// Override the Reproduce method to add the animal to the animal type's list
public override void Reproduce()
{
    // Reset reproductionPoints
    reproductionPoints = 0;

    // Spawn a child animal just to the right of this animal
    GameObject childAnimal = Instantiate(HerbivoreObject, new Vector3(transform.position.x + 0.5f, transform.position.y, transform.position.z), transform.rotation);


    // Call the 'AddHerbivoreToList' method from the 'AnimalsCounter' script
    GameObject.Find("AnimalsCounter").GetComponent<AnimalsCounter>().AddHerbivoreToList(childAnimal);
}
```

The code simply calls the AddHerbivoreToList method from the 'AnimalsCounter' script attached to the 'AnimalsCounter' GameObject. Of course, the code will be adapted for each animal type's scripts. After this I adjusted the comments in several files.

Next up is to use dynamic polymorphism (override functions!) to adapt the Carnivore's scripts to find the closest Herbivore, rather than the closest Food item. To do this I have overridden the 'ChooseNearestFood' routine to find the nearest HerbivoreObject from within the HerbivoresCounter List, rather than Food. I then had to fix an annoying error: 'Object is not set to an object reference' etc. etc. for this line of code:

```
// Call the 'AddHerbivoreToList' method from the 'AnimalsCounter' script
GameObject.Find("Animals Counter").GetComponent<AnimalsCounter>().AddHerbivoreToList(childAnimal);
```

The fix was to change 'AnimalsCounter' to what it is above, 'Animals Counter'. I felt I had to document this bug to get a bit of frustration out!

Now the Carnivore chases the instantiated Herbivore, I am going to adjust it's collision conditions. This involved simply changing the tag 'Food' to 'Herbivore' as well as the GameObject's tag and making the Herbivore's collider a trigger. Now the carnivore eats the herbivores as intended.

I have tweaked the spawning speed of the Food and decreased the required points for reproduction of the Herbivores to 3 to ensure the carnivore has more food available (he was dying out) however I would rather tweak the project properly at the end as this makes more sense.

Finally, I have deleted all the other functions except 'Reproduce' and altered the code so that CarnivoreObject is instantiated instead of a herbivore, as well as adding it to the CarnivoresCounter List.

Now I can get started on the Omnivore animal type. As we already know, the omnivores will eat both Food and Animals, so I will ultimately need to get 3 lots of nearest GameObjects (Food, Herbivores and Carnivores). To begin, I have copied all the code over from the Herbivores script. I have also taken the routine that finds the closest Herbivore from the Carnivore's script and changed it's variables to represent closestFood2. Below this, I have

created another (almost identical) routine, however changed the variables to represent closestFood3. This routine finds the closest Carnivore.

Next, I have adapted the OnTriggerEnter function condition to the following:

if (other.tag == “Food” || other.tag == “Herbivore” || other.tag == “Carnivore”)

Then I changed the Reproduce function to spawn in Omnivores and add them to the correct List. Now for the part I deem the most difficult – I need to compare the distances between the nearest Food, nearest Herbivore and nearest Carnivore GameObjects.

To do this I have used yet another routine called ‘actualClosestFood’. This takes in GameObjects of type T (generic type parameter). The List stores all of the results from the previous 3 ‘closestFood’ routines, this is done within the FindFood function:

```
// Override the FindFood method to find the nearest out of the 3 food choices
// The FindFood method calls for the closestFood item using the ChooseNearestFood routine
public override void FindFood()
{
    // Get the distances from each 'closest' food
    closestFood = ChooseNearestFood();
    closestFood2 = ChooseNearestFood2();
    closestFood3 = ChooseNearestFood3();

    // Add the closestFoods to the actualClosestFood List
    // If they are not already contained
    if (!actualClosestFoods.Contains(closestFood))
        actualClosestFoods.Add(closestFood);
    if (!actualClosestFoods.Contains(closestFood2))
        actualClosestFoods.Add(closestFood2);
    if (!actualClosestFoods.Contains(closestFood3))
        actualClosestFoods.Add(closestFood3);

    // Set the actual closest food to the CLOSEST of the closest foods!
    actualClosestFood = ChooseActualNearestFood();
}
```

First the first 3 routines are called for and then the results are added to the new List. The FindFood function is called within the Update function and so the results will be accurate at runtime (called every frame). Next the new routine cycles through the items and finds the closest one, just like the previous routines. To make this work, I have also overridden the Move function to use the ‘actualClosestFood’ GameObject reference to move to.

Now I am going to effectively work backwards and move the functionality also needed in the Carnivore class into the base class so that it can be inherited. This includes the routine to find Herbivores to eat. To avoid confusion, I am going to keep ‘FindFood’ the same as well as the ‘actualClosestFoods’ List and the ‘actualClosestFood’ GameObject. as I do not want to change both the base class AND the herbivore class (now it is after the fact).

Before bringing this project towards an end, I wanted to add some kind of simple combat system, because at the moment random animals (or both) are deleted upon colliding. Therefore, within the Animal class I have added a variable called ‘powerPoints’. Each animal is awarded one powerPoint whenever they grow (hence an animal’s size means it is

stronger) and upon collision, the animal with the larger amount of powerPoints wins. If they have equal points, the aggressor wins. Here is a screenshot of the collider code:

```
// Sets up and OnTriggerEnter method for the animal
void OnTriggerEnter(Collider other)
{
    // Delete any Food/Animal the animal comes into contact with
    if (other.tag == "Food" || other.tag == "Herbivore" || other.tag == "Carnivore")
    {
        if (other.tag == "Food" || other.tag == "Herbivore")
        // Destroy the Food/Animal item
        Destroy(other.gameObject);

        if (other.tag == "Carnivore")
        {
            if (this.powerPoints >= other.GetComponent<Animal>().powerPoints)
            {
                Destroy(other.gameObject);
            }
            else
                Destroy (this.gameObject);
        }
        // Call for growth of the animal
        Grow();
    }
```

Finally, to balance the game, I have set the starting animals to 8 herbivores, 6 carnivores and 4 omnivores. I have vastly increased the instantiation rate of the Foods and lowered the reproductionPoints required to reproduce down separately for each animal type and did the same for movement speeds.

Here is a video of the (for now) final project in action:

https://twitter.com/charliegfrost/status/1207805377791827969

Reflection Section

In hindsight, I could have utilised inheritance even better, as some code is repeated throughout the Carnivore and Omnivore scripts, however I am satisfied that I have used it sufficiently in this project and now fully understand it.

Overall, I am happy with the layout and organisation/cleanness of the code, I should probably even comment slightly less! However, I believe anyone could read the code and work with it rapidly and that is very important to me.

If I were to put more hours into this project, I believe it could benefit greatly from an improved pathfinding system (especially for Herbivores, to dodge the other animals) as herbivores rarely ever win by outrunning their opponents to the Food.