2025

Personal Project Overview

MADE BY: MIHAIL BOGDANOV

Class: V-OL-MAIFS-AIS-M-RB01 Semester 7

Contents

1. Introduction

In this document I will be showcasing the work I have done on my personal project for this semester. You can see the process of making this whole project from scratch and can also serve as a guide how to recreate it.

2. Project Overview

Description of project

My project is an Ecosystem Simulation. It was intended as a personal learning exercise to explote Reinfrorcment Learning. This project aims to show the importance of ecosystem balance by simulating predator-prey dynamics. This is done by having three main components:

- Autonomous Moose
- Autonomous Wolf
- Tree

The idea is that the moose eat trees and the wolf hunt moose. All three components should reproduce, eat and run to survive. The idea is that a person will be prompted how much to put of each component in the simulation. This should then showcase that if you have too much of one and too less of the other the ecosystem may fail to survive.

This project is built in Unity. Unity gives us an easy to use library called 'ML-Agents'. This lets us train a model that can be used in unity and then put on a game object. All scripts are written in C#.

Current state of project

The project currently has trained agents for the Wolf and Moose. The moose is trained to eat trees when hungry and the wolf is trained to eat and chase the moose. Trees are also in the game as static objects. Each agent can also die of hunger. When the simulation starts a couple of moose, wolf and trees spawn. You can then watch how the ecosystem lives.

The features that I will try to incorporate during the last weeks of the semsetr will be:

- Graphs of how the population of each species fluxtuated.
- The graphs will be compared to other graphs of the Lotka Volterra Model.
- (Optional) Water seeking behaviour and make use of memeory parameter to remember water locations

- (Optional) Improve training of agents to wander around instead of camping food
- Create a better map

3. Project Setup

ML-Agents library installation

To use ML-Agents we need a couple of things:
- Anaonda
- Python venv
- ML-Agents package in untiy

1. Install Conda

- To install conda you can go to the Anaconda website and install their anaconda navigator.

2. Venv

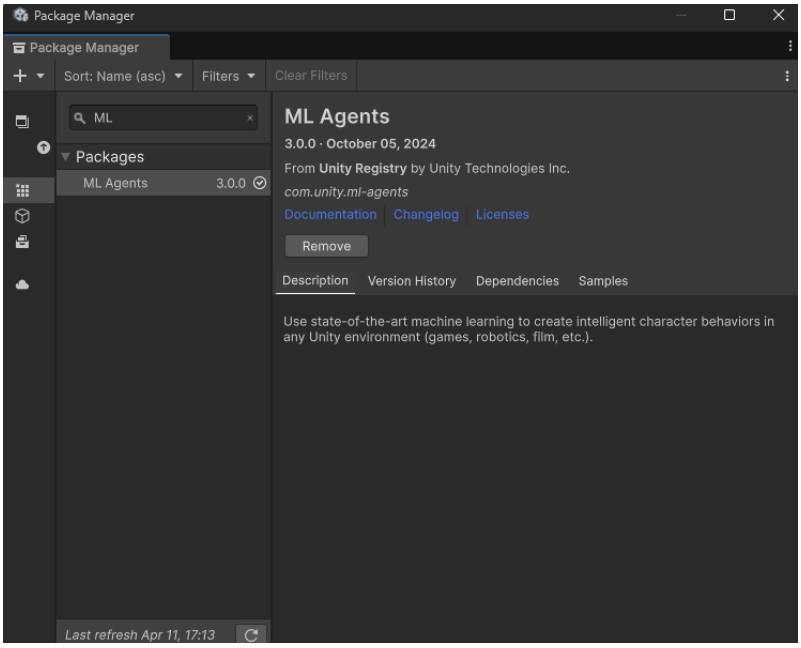
- To utilize the commands from ML-Agents we need a virtual python environment with the proper packages.
- From the Anaconda Navigator open the Anaconda Prompt
- Type: ***conda create --name (name of env) python=(desired version of python)***
- What has worked for me is to use **python version 3.9** since its compatible.
- Now you can go into the env using ***conda activate (name of env)***
- Now you have to install the proper packages with the command ***pip install (package name)***
- The packages we need are: torch, mlagents, onnx, protobuf

```
pip install torch --index-url https://download.pytorch.org/whl/cu
pip install mlagents==0.30.0
pip install onnx==1.16.1
pip install protobuf==3.20.3
```

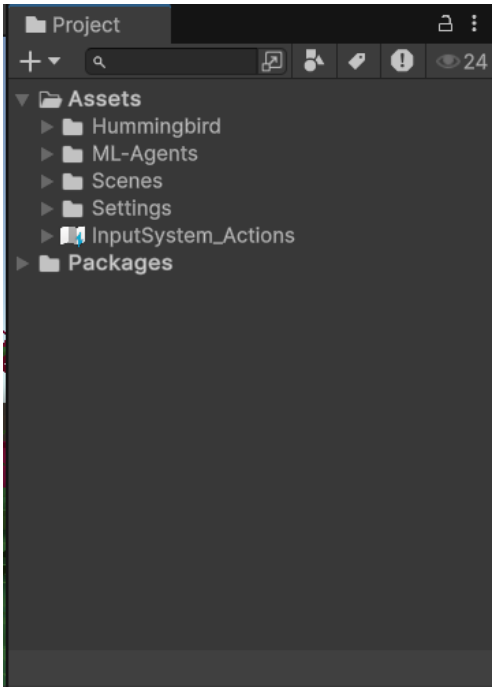
- **MLagents:** The library we use to train our RL agnets in unity
- **Torch:** An open-source machine learning library used to train deep learning models.
- **Onnx:** A tool to import, export, manipulate and represent machine learning models.
- **Protobuf:** A tool used to serialize large structured data.

3. Import ML-Agents Package in Unity

- Open Unity
- Go to Window -> Package Manager and a window should appear
- In the searchbar wrtie ML Agents and you should see the package and click “Install”.

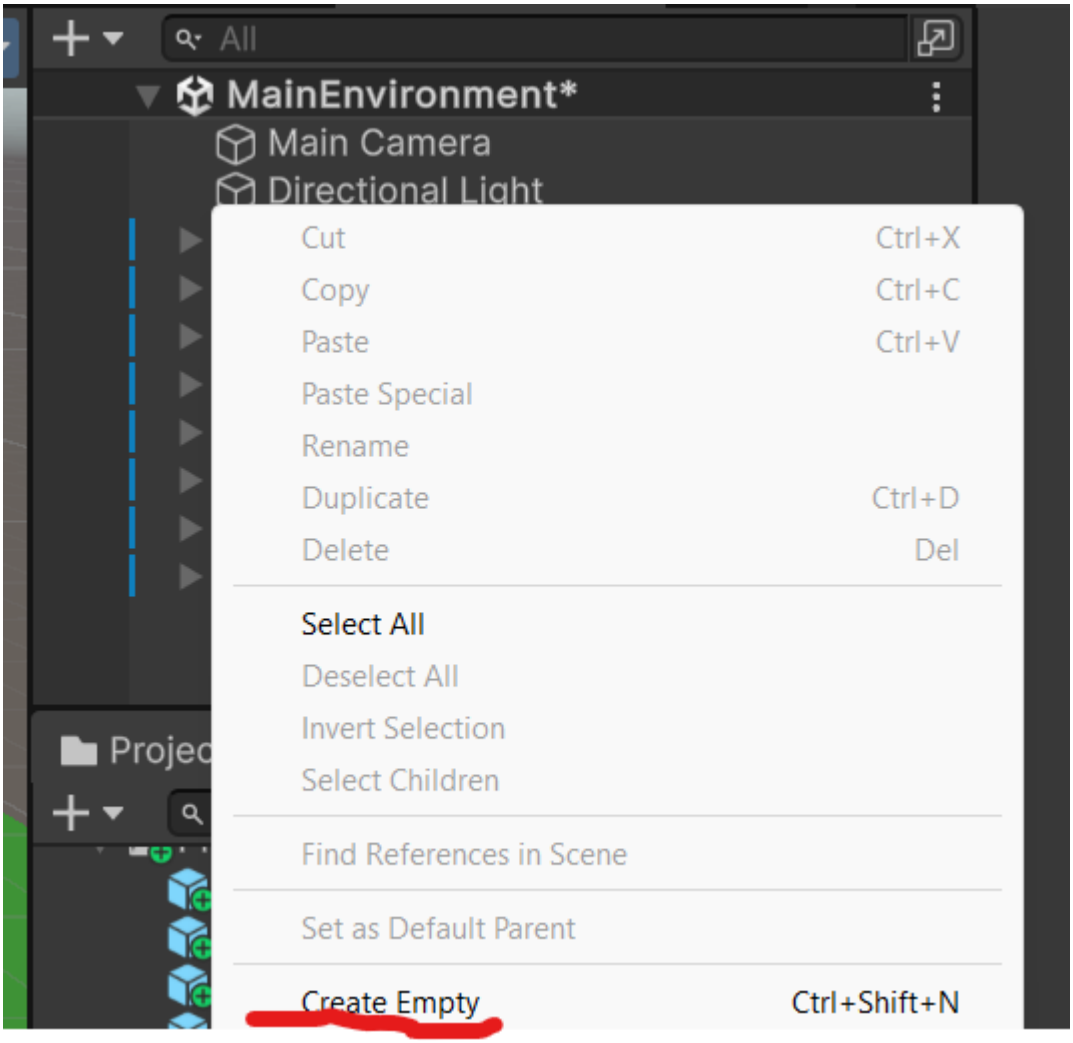


- When that is done you should see a new “ML-Agents” folder appear in your “Project” folder in Unity.

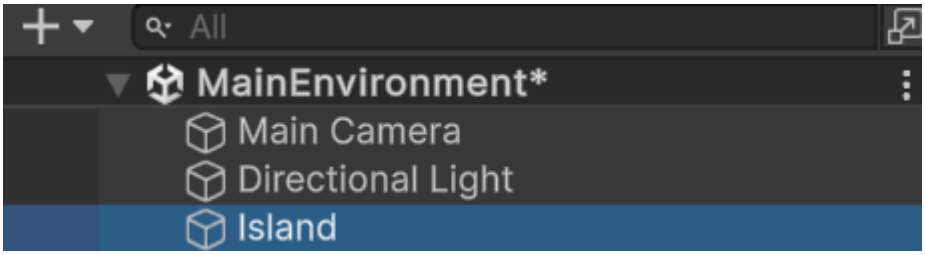


Scene Setup

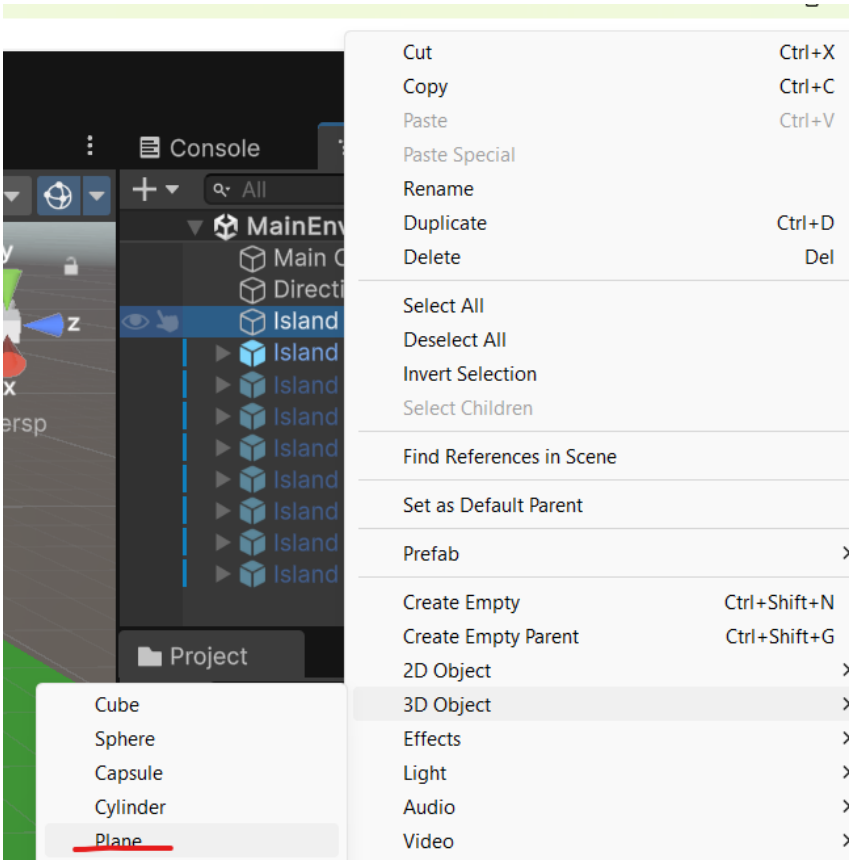
For our agent to train we need to have a training environment. This environment should also be very simple graphically, since while we train we will be running thins envrionment multiple times at the same time. So in the main scene we can right click and click “CreateEmpty”.



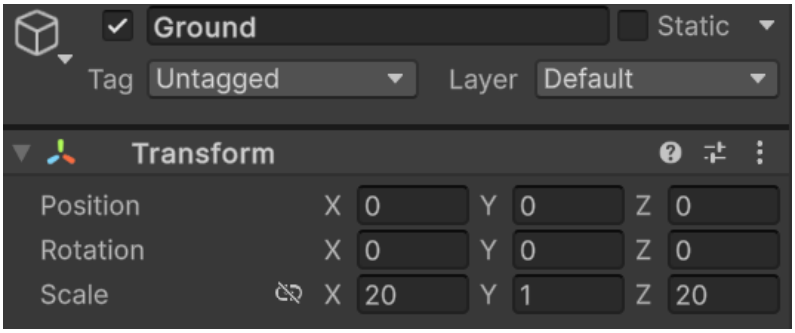
We then name that empty game object as “Island”.



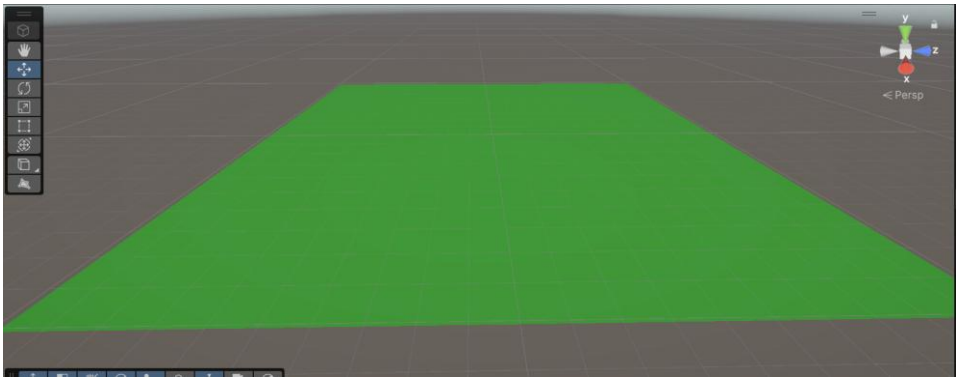
In there we can add our actual “island” by creating a 3D plane.



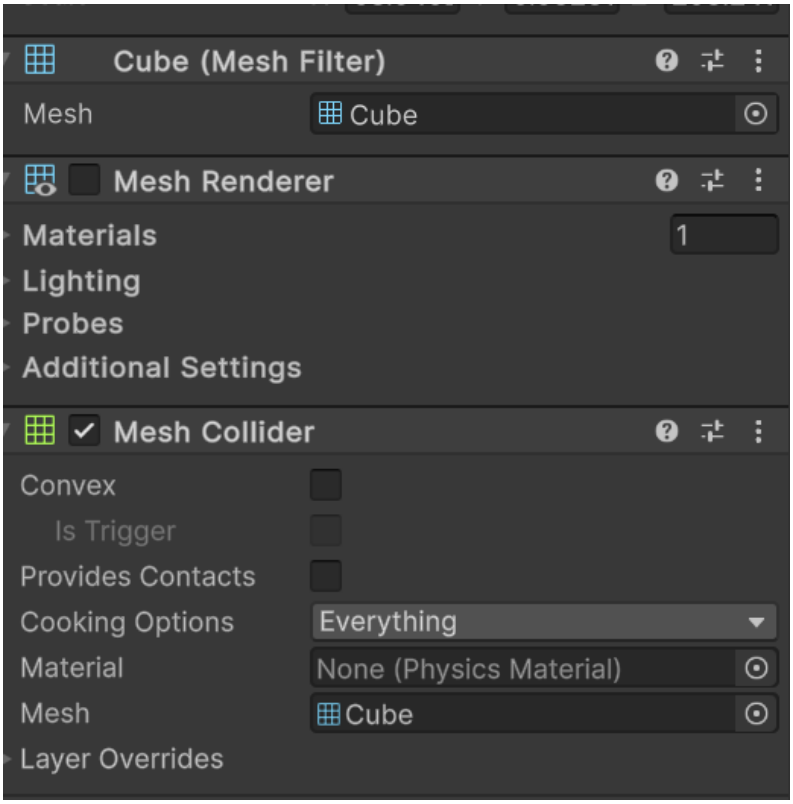
You can then set the dimensions, mainly the scale, however you like in the Transform section here:



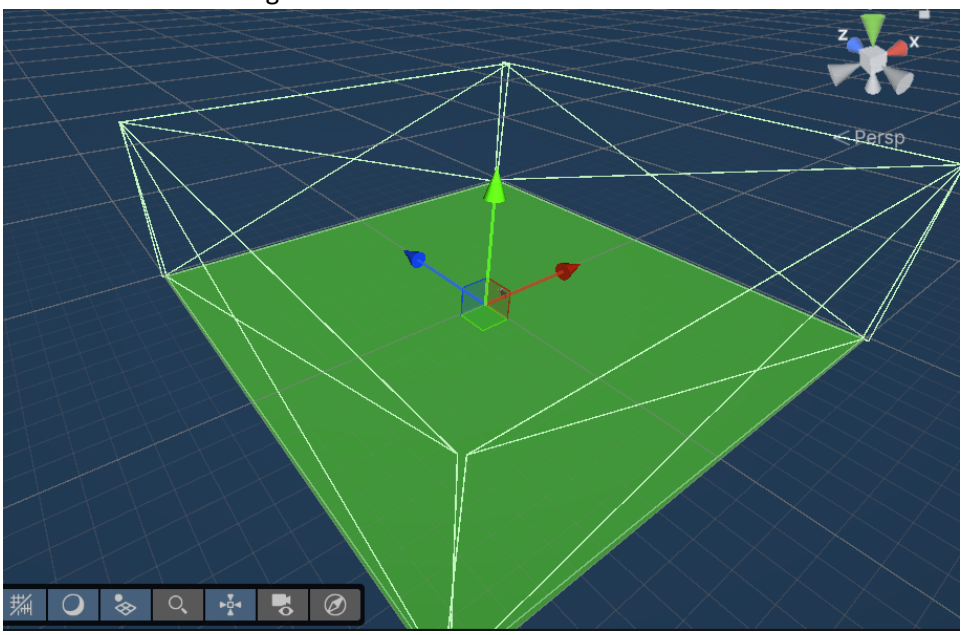
At the end you should have something like this:



After that we have to create the invisible walls. We need them so that our agent does not fall from the ground and also so that the agent can learn not to bump into them. We create another empty object in the “Island”. We name it “IslandBoundaries”. Then in there we add another 3D plane. Resize it to the size we want, utick the “Mesh Renderer” so that it is invisible.

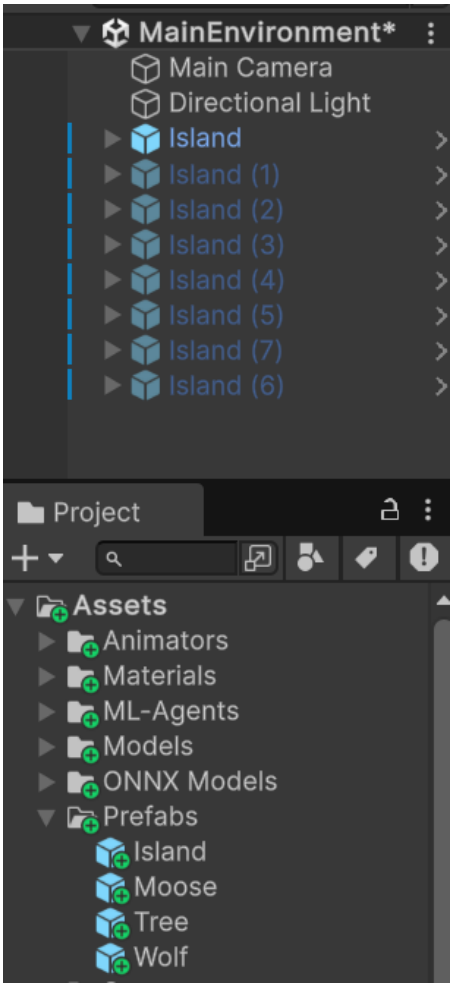


Then rotate it to be as a wall to the environment. Copy paste the wall, rotate it and you should have something like this.



The green outlines are the walls.

After we did that we should make our “Island” into a prefab. A prefab is like a template of a game object. After you make a game object into a prefab you can use it or make a copy of it wherever you like, with all its changes instead of making it all again. This is how we do it. We create a Prefab folder in our project structure, then just drag the “Island” object into the prefab folder and done. You have a prefab. And the game object box should turn blue.

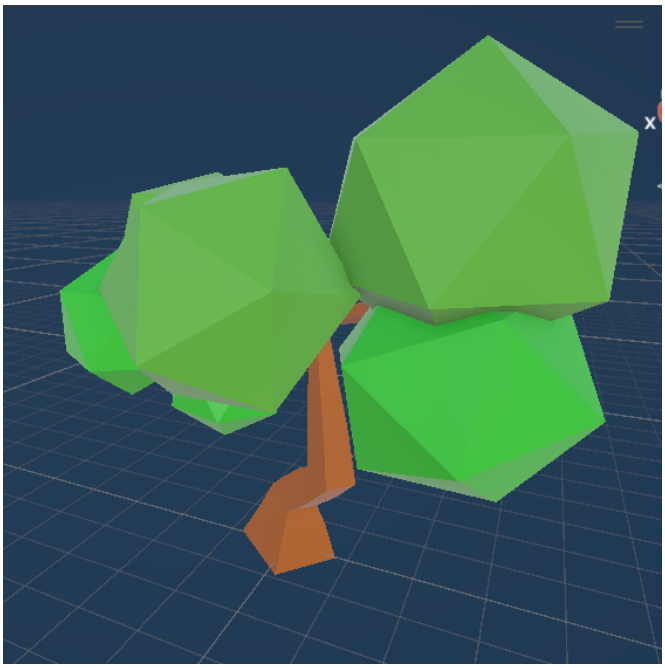


We have our training environment now. Or atleast “physically”. We now want to populate this environment with the trees, moose and wolves.

4. Making models in Blender

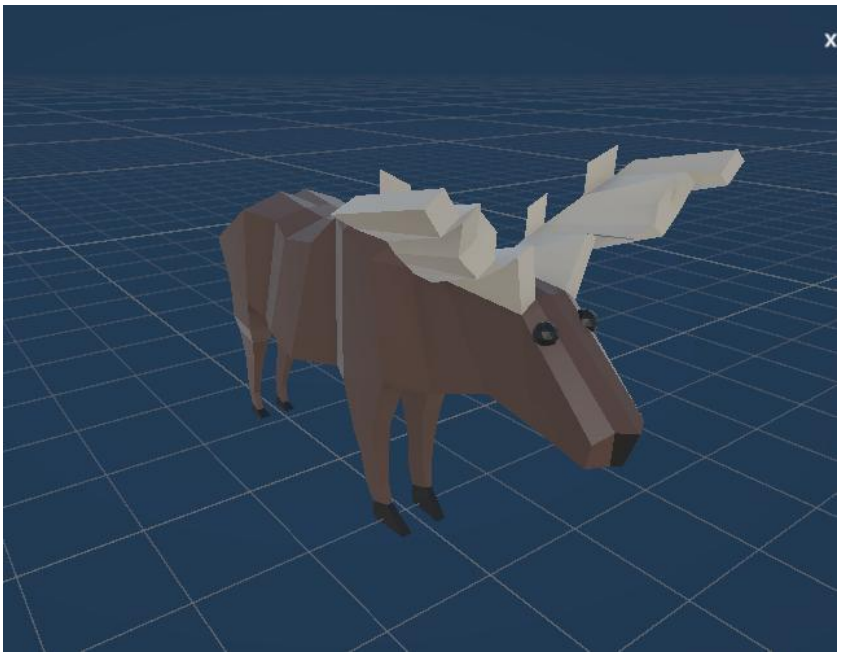
Here I had a problem. I had no idea how to create models to use in unity and since I did not find any models of these animals online that I liked and that were consistent with eachother I decided to learn how to create my own. I will not go into depth about this but I will share the sources that helped me learn how to do it.

Tree



The picture above is of the tree model. This picture was taken in unity. I created this model by following this video: https://www.youtube.com/watch?v=hvxoAX_poI0

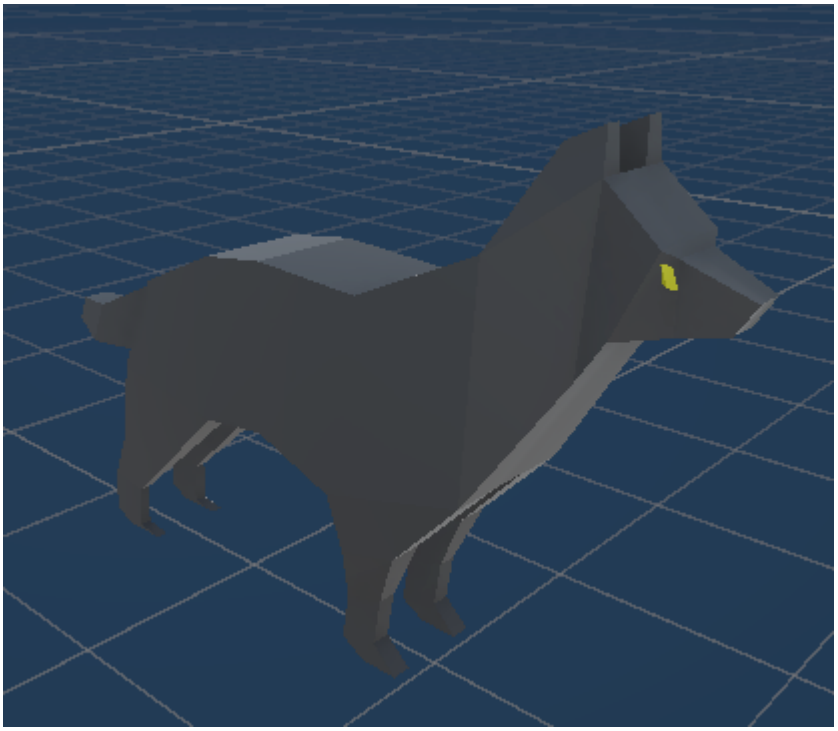
Moose



The picture above is of the moose model. This picture was taken in unity. I created the model by following this video: https://www.youtube.com/watch?v=6mT4XFJYq-4

I also created animations for it. I learned how to do so by following this video: http://youtube.com/watch?v=Sz0FhxaTAvE&t=271s and also this video: http://youtube.com/watch?v=iMar3keWaUo&t=766s

Wolf



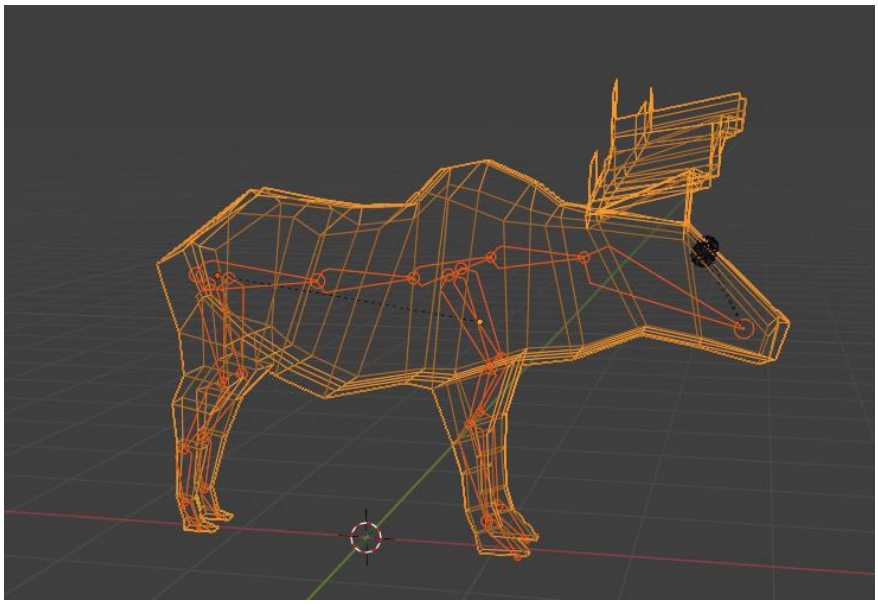
The picture above is of the moose model. This picture was taken in unity. I created the model by following this video: https://www.youtube.com/watch?v=6mT4XFJYq-4

I also created animations for it. I learned how to do so by following this video: http://youtube.com/watch?v=Sz0FhxaTAvE&t=271s and also this video: http://youtube.com/watch?v=iMar3keWaUo&t=766s

How to export

Now that the models are created, they have to be exported in a format that allows us to use them in unity. So in Blender do the following:

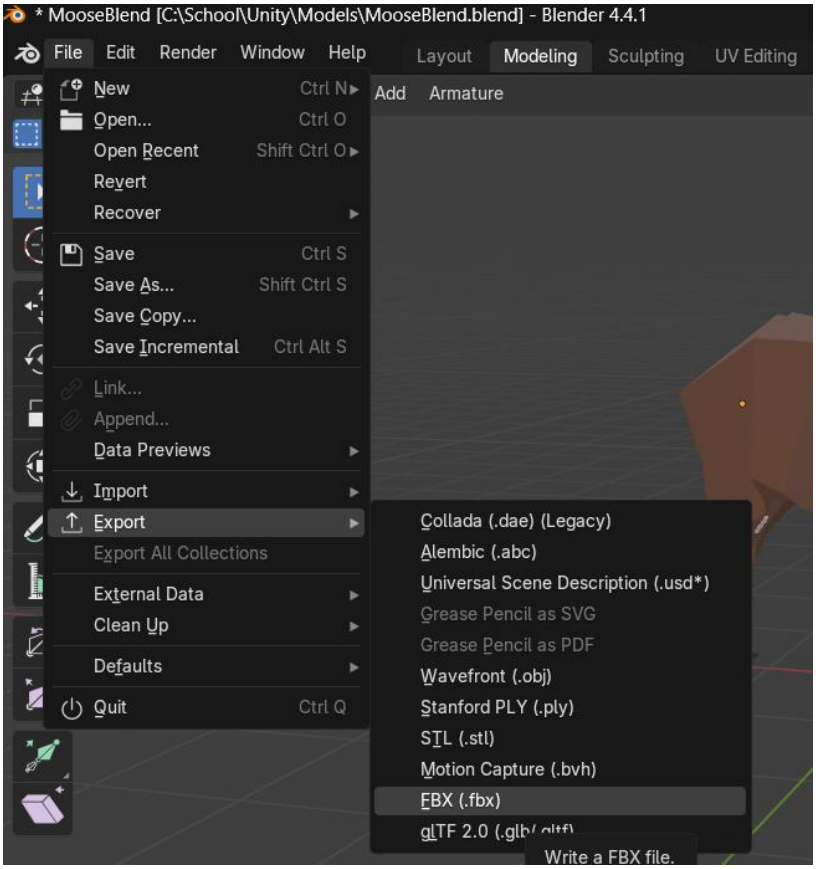
In “Object Mode” select both the model and the armature. The model is the model of the moose itself, and the armature is its “skeleton” which by moving it allows us to animate the model.



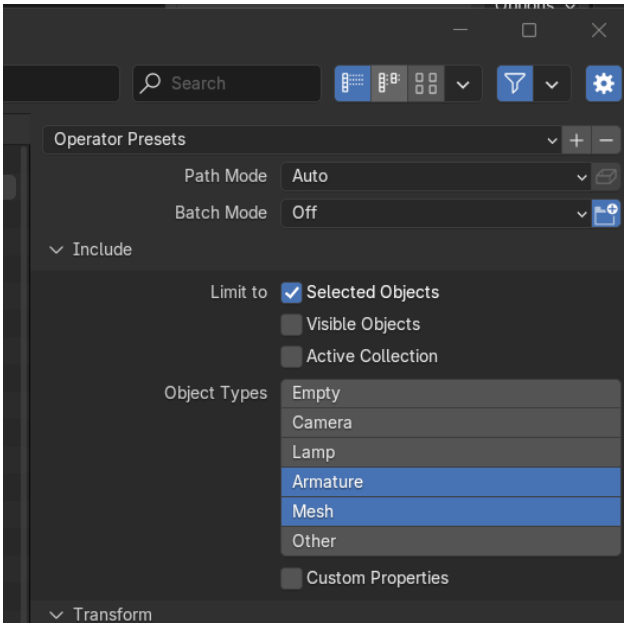
You can see they are both selected by the model being orange and the armature being darker orange.

Then you need to go to:

File -> Export -> FBX

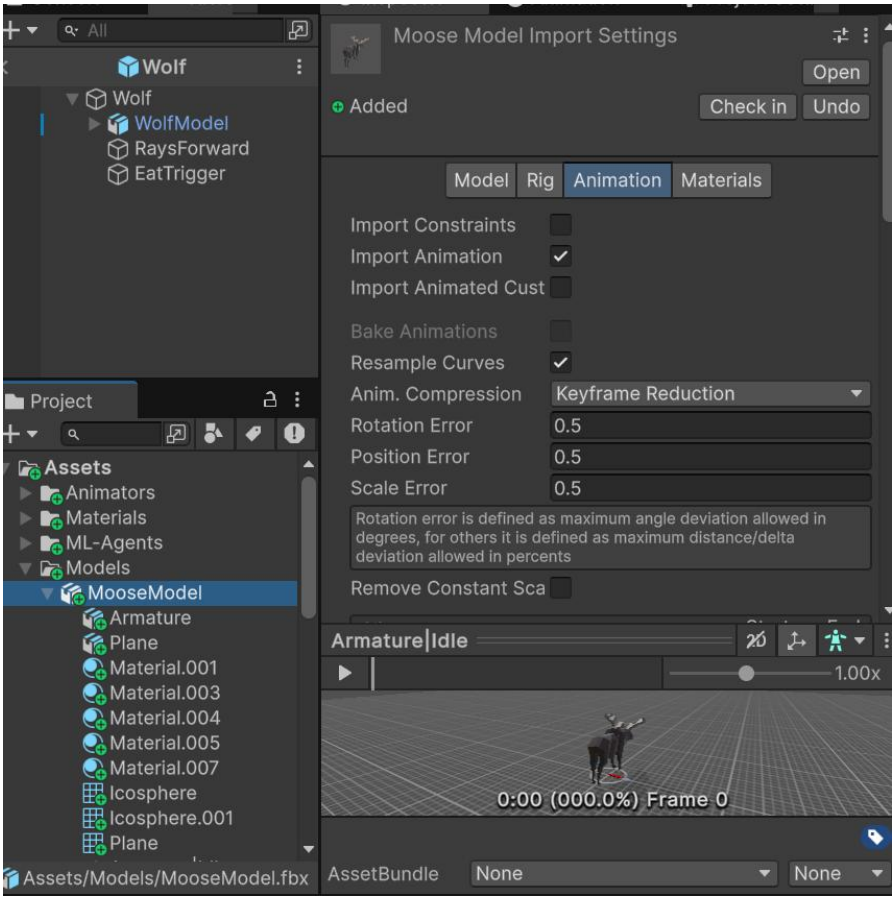


Then you want to have your settings like this:



You only want to export the selected objects, and only armature and mesh should be selected as object types.

Then you hit export and in the folder you selected you should now have an FBX file. After that you can drag that fbx file in a folder in your unity project sructure and you should have something like this:



5. Scripts

Now that we have our island, animals and trees we can start writing code for them. Below you will see the explanation of each script and how to apply it properly. All the scripts should be in a separate folder called scripts in your project structure.

Tree Area Script

This script is responsible for spawning trees on the island.

Variable setup:

```
Unity Script (1 asset reference) | 4 references
public class TreeArea : MonoBehaviour
{
    [Tooltip("The tree prefab to spawn")]
    public GameObject treePrefab;

    [Tooltip("How many trees to spawn")]
    public int treeCount = 10;

    [Tooltip("The radius in which the trees will be spawned.")]
    public float spawnRadius = 100f;

    [Tooltip("Minimum distance between spawned trees.")]
    public float minTreeSpacing = 3f;

    private List<GameObject> spawnedTrees = new List<GameObject>();
```

Here we have our variables setup.

- **treePrefab:** The prefab we want to spawn which is the tree. So you will want to make a prefab of the tree aswell by making an empty game object and then add the tree model to it.
- **treeCount:** The amount of trees we want this script to spawn. This can be later edited in unity. Each variable set to public can be edited in the GUI later.
- **spawnRadius:** The radius in which the trees can spawn
- **minTreeSpacing:** The minimum distance between each tree.
- **spawnedTrees:** A list of all trees spawned which we will use later.

Reset Area method:

This method is responsible for destroying old trees and spawning new ones.

```
public void ResetArea()
{
    if (treePrefab == null)
    {
        Debug.LogError("TreeArea: treePrefab is not assigned in the Inspector.");
        return; // Early exit to prevent crash
    }

    // If we haven't spawned trees yet, do it now
    if (spawnedTrees.Count < treeCount)
    {
        for (int i = 0; i < treeCount; i++)
        {
            GameObject newTree = Instantiate(treePrefab);
            newTree.tag = "Tree";
            newTree.transform.parent = transform;
            spawnedTrees.Add(newTree);
        }
    }

    int tries = 0;
    int placed = 0;

    foreach (var tree in spawnedTrees)
    {
        tree.SetActive(false); // deactivate before repositioning
    }
```

First we check if a prefab is set to be spawned. If not we just retrurn so we avoid any crashes.

Then we check if trees have been spawned. If not we spawn them now. There is a for loop which goes on as many times as we have indicated in the treeCount variable. We then

create a new tree and tag it that it is a tree (this is important for the agent). After that we add it to the spawnedTrees list.

After we have all of our trees in the list we now want to place them in different positions since right now they all just go in one position. So we prepare the variables “tries” and “placed”. We have these since we will be trying to place the trees but not every try might be susccessful. It maybe could collide with another tree so it would have to try again. When a tree gets placed the placed counter goes up.

Then we deactivate all trees so that they are not visible while trying to place them.

```
        }

        while (placed < treeCount && tries < treeCount * 10)
        {
            Vector3 pos = GetRandomPositionInArea();
            if (IsPositionFree(pos))
            {
                GameObject tree = spawnedTrees[placed];
                tree.transform.position = pos;
                tree.transform.rotation = Quaternion.Euler(0f, Random.Range(0f, 360f), 0f);

                Tree treeComponent = tree.GetComponent<Tree>();
                treeComponent?.ResetTree();

                tree.SetActive(true);
                placed++;
            }
            tries++;
        }
        Debug.Log("TREE AREA RESET");
    }
```

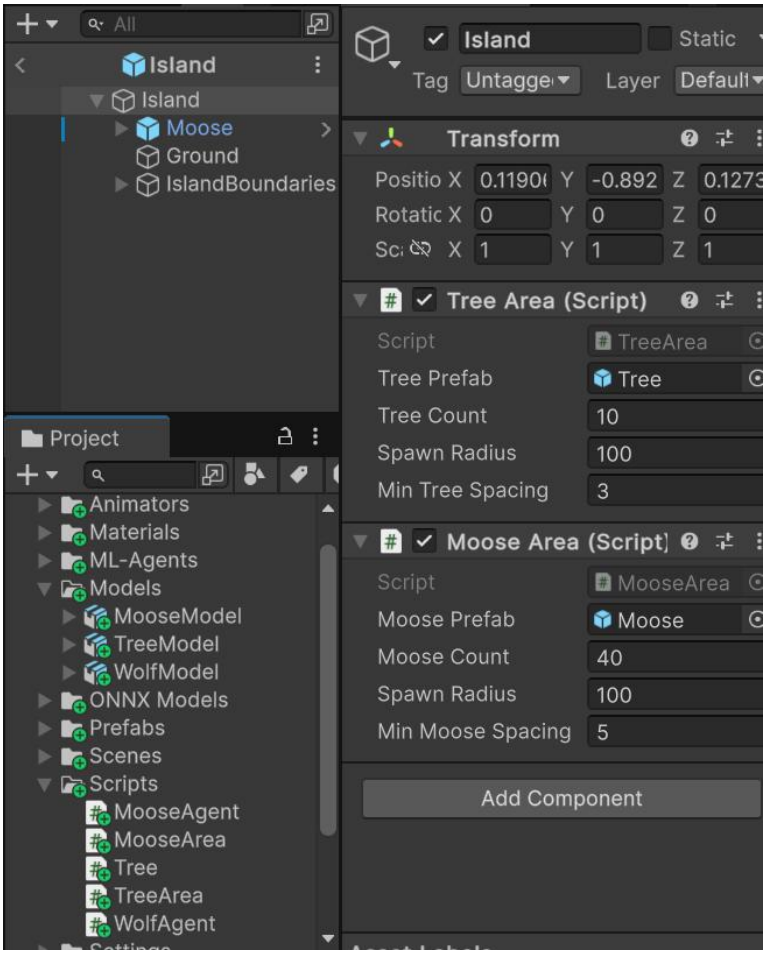
Then while the placed count is not as much as the tree count, we try and place a tree. We have a method named “GetRandomPositionInArea” which gives us backa position on the map. Then if that position is free we get the tree from the list, give it the chosen positon and a random rotation. After that the tree I set as active and the placed counter goes up.

Below you can see the methods that determine the position and if it is free:

```
1 reference
private Vector3 GetRandomPositionInArea()
{
    Vector2 circle = Random.insideUnitSphere * spawnRadius;
    Vector3 position = new Vector3(circle.x, 0f, circle.y) + transform.position;

    position.y = transform.position.y;

    return position;
}

1 reference
private bool IsPositionFree(Vector3 position)
{
    //Check if any spawned trees are next to spawning point
    foreach (GameObject t in spawnedTrees)
    {
        //If tree is destroyed then skip to next tree
        if (t == null)
        {
            continue;
        }
        if (Vector3.Distance(position, t.transform.position) < minTreeSpacing)
            return false;
    }
    return true;
}
```

Applying the script:

Now we have to apply this script to our Island object. To do so go in Unity and open your island prefab. Then drag the TreeArea script to the island, the empty object holding the gournd and walls together. And now you should have something like this:



Now you can see that in the “Tree Area” you can manipulate the public variables we already set. You also have to add the tree prefab to the prefab secition.

Tree Script

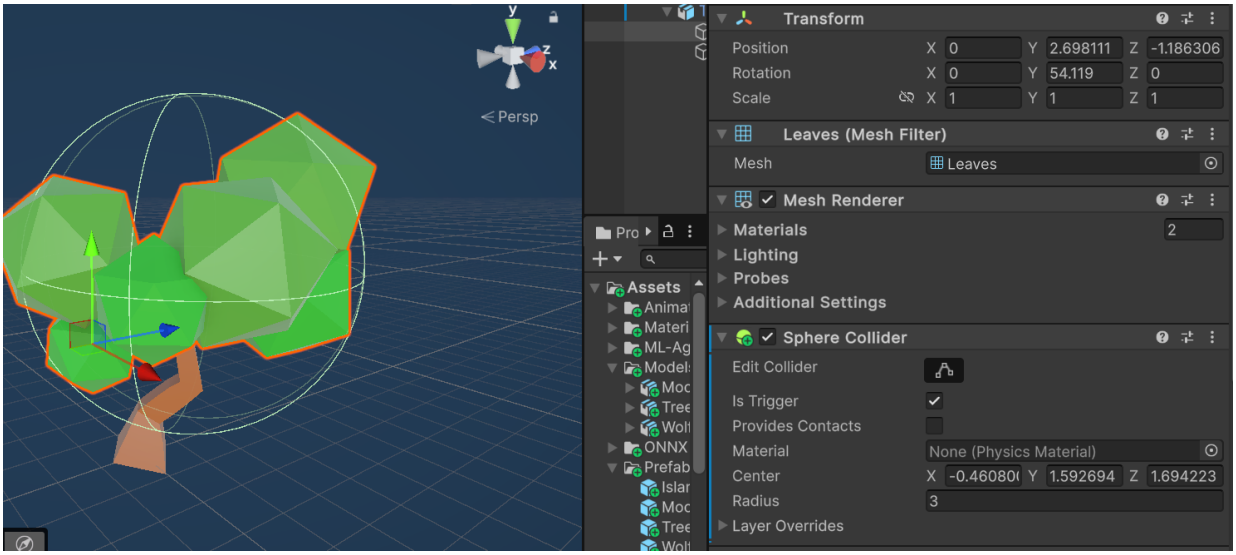
This is the script for the behaviour of the tree.

Variable setup:

```
Unity Script (1 asset reference) | 7 references
public class Tree : MonoBehaviour
{
    [Tooltip("The trigger collider on the leaves")]
    public Collider leafCollider;

    [Tooltip("The trunk mesh collider")]
    public Collider trunkCollider;

    [Tooltip("Destroy delay after being eaten.")]
    public float destroyDelay = 0f;

    6 references
    public bool HasLeaves { get; private set; } = true;

    private Renderer[] renderers;
```

First we have our variable setup again. We have a true or false that if the tree has leaves or not. This just means is it eaten or not. We have a destroy delay for visual effect if we want the tree dissappear a bit later. Now we have two “collider” variables. One for the trunk and one for the leaves. These colliders tell us if a moose collides with the leaves or not. Then we know if we want to trigger the eating or not. These colliders should be setup in Unity. This is how:

Go to Unity into the tree prefab. I have exported the tree in scuh a way that it has two objects. Leaves and Trunk. First select the leaves. In the inspector on the side click Add Component. Then you should add a sphere collider:



You should then see this big sphere. You can resize it and put it in another position if needed. Then it is very important to click “isTrigger”. This will make our collider not physicall. Also it will serve as a trigger for the eating action. If the moose goes to the tree then the trigger can tell us that and we can trigger the appropriate action.

We can then do the same for the trunk. But there DO NOT click the isTrigger. We want that to be an actualy physical collider so that our animals can bump into it. You can add a Mesh Collider. The collider will take the form of the tree trunk.

Now that we have the colliders setup we can go back to the code.

Awake Method:

```
Unity Message | 0 references
private void Awake()
{
    // Get colliders automatically if not assigned
    if (leafCollider == null)
    {
        Transform leaves = transform.Find("Leaves");
        if (leaves != null)
            leafCollider = leaves.GetComponent<Collider>();
    }

    if (trunkCollider == null)
    {
        Transform trunk = transform.Find("Trunk");
        if (trunk != null)
            trunkCollider = trunk.GetComponent<Collider>();
    }

    renderers = GetComponentsInChildren<Renderer>();
}
```

The awake method triggers the moment the tree is set to active, or when the simulation starts initially. Here we find the colliders of the leaves and trunks. We also find the renderer of the tree. This will help us make it invisible later.

Consume method:

```
public void Consume()
{

    if (!HasLeaves) return;

    HasLeaves = false;

    // Disable colliders
    if (leafCollider != null) leafCollider.enabled = false;
    if (trunkCollider != null) trunkCollider.enabled = false;

    // Hide visuals
    foreach (Renderer r in renderers)
    {
        r.enabled = false;
    }

    // Deactivate the tree for pooling instead of destroying
    gameObject.SetActive(false);

}
```

The consume method is used to despawn the tree. It can be called from other classes such as the Moose script to consume the tree. When this happens we deactivate the tree. We ddo not delete it so that we can respawn it again later. This is mainly because of using the tree in a training environment for the moose and wolf.

ResetTree method:

```
public void ResetTree()
{
    HasLeaves = true;

    if (leafCollider != null) leafCollider.enabled = true;
    if (trunkCollider != null) trunkCollider.enabled = true;

    foreach (Renderer renderer in renderers)
    {
        renderer.enabled = true;
    }
}
```

This method is used for activating the tree again.

Moose Script

Here is where the interesting part is. This script has the reward dessign for the moose, the logic for executing its actions and how to interact with the environment.

Variable setup

```
Unity Script (1 asset reference) | 7 references
public class MooseAgent : Agent
{
    [Header("Movement")]
    public float moveForce = 5f;
    public float turnSpeed = 100f;
    private float smoothTurnChange = 0f;

    [Header("Hunger")]
    public float hunger = 1f;
    public float hungerDrainRate = 0.005f;
    public float hungerThreshold = 0.5f;
    private float starvationTime = 0f;
    public float maxStarvationTime = 50f;

    private Rigidbody rb;
    private TreeArea treeArea;
    private Tree nearestTree;
    public bool trainingMode;

    private Animator animator;
    private Collider collider;
    private Renderer[] renderers;
```

First in the code above, we intialize the variables and objects we will be using.

- **moveForce:** How fast the bird will be moving.
- **turnSpeed:** How fast the bird will turn left or right.
- **smoothTurnChange:** This will be used to smoothen the turns from left to right.
- **hunger:** The initial hunger level. 1 is full 0 is starving.
- **hungerDrainRate:** how much the hunger level goes down constantly
- **hungerThreshold:** if the hunger is below the threshold the animal should start searching for food
- **starvationTime:** how long the level of the hunger bar has been 0
- **maxStarvationTime:** the max amount of time the hunger bar can be 0 before the animal dying
- **rb:** The rigid body of the moose.
- **treeArea:** The area in which the agent is.
- **nearestTree:** We use this to find the nearest tree to the bird.
- **trainingMode:** Based on this, if its true or not the environment logic will be different.
- **Animator:** This helps us trigger appropriate animations
- **Collider:** The collider of the moose
- **Renderer:** The render of the moose.

You can also see that this scripts inherits from the class *"Agent"*. This class is from the Ml-Agents library. We inherit because we will be overriding some methods.

Intialize and OnEpisodeBegin methods

```
0 references
public override void Initialize()
{
    rb = GetComponent<Rigidbody>();
    treeArea = GetComponentInParent<TreeArea>();
    animator = GetComponentInChildren<Animator>();
    collider = GetComponent<Collider>();

    renderers = GetComponentsInChildren<Renderer>();


    if (!trainingMode) MaxStep = 0;

}

0 references
public override void OnEpisodeBegin()
{
    rb.linearVelocity = Vector3.zero;
    rb.angularVelocity = Vector3.zero;

    MoveToSafeRandomPosition();

    transform.rotation = Quaternion.Euler(0f, Random.Range(0f, 360f), 0f);

    if (trainingMode) { treeArea.ResetArea(); } //figure out how to spawn trees at start of actual sim.
    hunger = trainingMode ? Random.Range(0.2f, 1f) : 1f;

    animator.Rebind();
}
```

On initialization we get the body of the moose and the island its on. We also get its collider and animator. If we are not in training mode we set an infinite amount of timesteps available for the agent.

OnEpisodeBegin is a method from the Agent class. This activates when a new episode begins. If we are in training mode we reset the trees to different positions. We reset the velocity of the agent so it doesn't keep walking in the next episode in the direction it was walking. We stop any velocity that the moose has. We also moove the moose to a safe position.

MoveToSafeRandomPosition method:

```
1 reference
private void MoveToSafeRandomPosition()
{
    int attempts = 100;
    while (attempts-- > 0)
    {
        Vector3 pos = new Vector3(
            Random.Range(-treeArea.spawnRadius, treeArea.spawnRadius),
            transform.position.y,
            Random.Range(-treeArea.spawnRadius, treeArea.spawnRadius)
        );

        // Use a small sphere to check overlap with obstacles
        if (Physics.OverlapSphere(pos, 0.5f).Length == 0)
        {
            transform.localPosition = pos;
            transform.rotation = Quaternion.Euler(0f, Random.Range(0f, 360f), 0f);
            return;
        }
    }

    Debug.LogWarning("Could not find a safe spawn position!");
}
```

In this method we move the moose to a random safe position in the area its in. So it doesn't collide with any other objects in the scene.

OnActionRecieved:

```
public override void OnActionReceived(ActionBuffers actions)
{
    //Movement
    float moveInput = Mathf.Clamp(actions.ContinuousActions[0], -1f, 1f);
    float turnInput = Mathf.Clamp(actions.ContinuousActions[1], -1f, 1f);

    Vector3 force = transform.forward * moveInput * moveForce;
    rb.linearVelocity = transform.forward * moveInput * moveForce + new Vector3(0f, rb.linearVelocity.y, 0f)

    if (Mathf.Approximately(moveInput, 0f))
    {
        rb.linearVelocity = new Vector3(0f, rb.linearVelocity.y, 0f);
    }

    // Smooth turn
    smoothTurnChange = Mathf.MoveTowards(smoothTurnChange, turnInput, 2f * Time.fixedDeltaTime);

    float currentYaw = transform.rotation.eulerAngles.y;
    float yaw = currentYaw + smoothTurnChange * Time.fixedDeltaTime * turnSpeed;

    transform.rotation = Quaternion.Euler(0f, yaw, 0f);

    // Animate walking
    bool isWalking = Mathf.Abs(moveInput) > 0.1f;
    animator.SetBool("isWalking", isWalking);

    //Drain Hunger
    hunger -= hungerDrainRate * Time.fixedDeltaTime;
    hunger = Mathf.Clamp01(hunger);
```

OnActionRecieved is again a method from the Agent class. So on every action we interpret the input actions that the ML agent is sending us. The two values in actions.ContinuousActions

represent the direction in which the agent wants to move: X (left/right) and Z (forward/backward). We combine them into a movement vector and apply a force to the agent's rigidbody to move it physically in that direction. The **"force"** variable scales how strong this movement is.

Then we start rotating the agent based on the turn input. Same as the move input.

We also trigger the walk animation if the move force is a bit more than 0.1. Also on each step we drain the moose's hunger.

```
if (hunger <= 0f)
{
    AddReward(-0.0005f);
    starvationTime += Time.fixedDeltaTime;

    if (starvationTime >= maxStarvationTime)
    {
        AddReward(-1f); // Big penalty for dying
        StartCoroutine(DespawnMoose());

    }
}
else
{
    starvationTime = 0f; // Reset if not starving
}
```

If the moose is starving we start giving the moose a negative reward constantly and up the stavation time. If the starvation time reaches its threshold we despawn the animal and give a big negative reward.

```
if (nearestTree != null && nearestTree.HasLeaves)
{
    if (hunger <= hungerThreshold)
    {
        Vector3 toTree = nearestTree.transform.position - transform.position;

        float alignment = Vector3.Dot(transform.forward.normalized, toTree.normalized);
        AddReward(0.001f * alignment);

        float dist = toTree.magnitude;
        float closeness = 1f - Mathf.Clamp01(dist/treeArea.spawnRadius);

        AddReward(0.001f * closeness);
    }

}
```

Finally, we calculate rewards based on how well the agent is moving towards a tree if its hungry.

- **toTree** is the direction vector from the agent's face to the tree.

- **alignment** measures how well the agent is pointing toward the tree — a higher dot product means better alignment.
- **dist:** is how far the face of the agent is from the flower
- **closeness:** gives a higher value when the agent is closer to the flower.

This is how we reward the moose for the more alligned and closer it is to the tree.

CollectObservations

```
0 references
public override void CollectObservations(VectorSensor sensor)
{
    if(nearestTree == null)
    {
        sensor.AddObservation(new float[13]);
        return;
    }

    //Observe agnets local rotation
    sensor.AddObservation(transform.localRotation.normalized);
    //Observe agents velocity. Use for running from wolves.
    sensor.AddObservation(rb.linearVelocity.normalized);
    //Observe agents hunger.
    sensor.AddObservation(hunger);

    Vector3 toTree = nearestTree.transform.position - transform.position;

    // Direction to tree (normalized 3 floats)
    sensor.AddObservation(toTree.normalized);

    // Distance to tree (1 float, normalized)
    float dist = toTree.magnitude / treeArea.spawnRadius;
    sensor.AddObservation(dist);

    // Alignment (1 float)
    float alignment = Vector3.Dot(transform.forward.normalized, toTree.normalized);
    sensor.AddObservation(alignment);

}
```

CollectObservations is also a method from the Agent class. It is called every time the agent needs to collect information from the environment. These observations help the neural network decide what actions to take next.

If there are no trees nearby we still need to send something since the neural network expects 13 values every time. The amount of observations required from the neural network is set in unity. We have set it to 13 since that is the total number of observations we have found usefull.

If there is a tree nearby, we begin gathering useful info about the world around the agent.

An observation for the agent's own rotation, normalized so all values are between -1 and 1. This tells the neural network what direction the agent is facing.

An observation for the agent's veolcity to know its speed. I have added this since it could be usefull for when running from wolves so it knows how to speed up.

An observation for it's hunger level so it knows when it should start searching for food.

We calculate and observe the direction to the flower, normalized. This will be used to givee the agent a clue about which direction it should go to reach the tree, no matter how far away it is.

At the end we observe how far away the tree is. The value is normalized by dividing by the area's diameter, so the agent always gets values between 0 and 1 regardless of how big the area is.

So in totatal that is 13 observations:

- 4 values for the rotation quaternion.
- 3 values for the velocity
- 1 value for the hunger
- 1 value for the direction to the tree.
- 3 value for the dist to tree.
- 1 value for the alignment to the tree.

Heuristic method

```
public override void Heuristic(in ActionBuffers actionsOut)
{
    float move = 0f;
    float yaw = 0f;

    if (Input.GetKey(KeyCode.W)) move = 1f;
    else if (Input.GetKey(KeyCode.S)) move = -1f;

    if (Input.GetKey(KeyCode.A)) yaw = -1f;
    else if (Input.GetKey(KeyCode.D)) yaw = 1f;

    var c = actionsOut.ContinuousActions;
    c[0] = move;
    c[1] = yaw;
}
```

Heuristic is a method that lets us control the agent manually, usually for testing purposes. This is super useful when we want to debug or just see how the agent behaves when controlled by a person. The idea is to take player input (like key presses) and convert that into the same kind of data the ML model would generate.

We start by creating placeholder variables for movement forward and turning sideways. They start with initial value 0 so no movement at start.

Then we just listen for input keys. W and S is used for moving forwards or backwards. D and E are used for turning left or right;

And finally, we assign the values to the output actions.

So now instead of the neural network deciding which action to use we do. But it works just the same way as in a simulation. This way we can test our environment.

TriggerEatAttempt

```
2 references
private void TriggerEatAttempt(Collider other)
{
    if (other.CompareTag("Leaves"))
    {
        Tree tree = other.GetComponentInParent<Tree>();
        if (tree != null && tree.HasLeaves)
        {
            if (hunger <= hungerThreshold)
            {
                animator.SetTrigger("eat"); // Play eating animation

                tree.Consume();
                hunger = 1f;
                AddReward(1f); // reward for eating properly

            }
            else
            {
                AddReward(-0.2f); // penalty for eating when not hungry
            }
        }
    }
}
```

TriggerEatAttempt is a method that checks if the agent is collinding with a “leaves” collider.

If the agent is colliding with the leaves collider we check which tree does this collider belong to. Then after that is known the agent consumes the tree, refills hunger and gets a big reward.

If the moose eats while its not hunry it gets a negative reward.

OnCollisionEnter

```
Unity Message | 0 references
private void OnCollisionEnter(Collision collision)
{
    if (trainingMode && collision.collider.CompareTag("Boundary"))
    {
        //Collided with the area boundary, give negative reward
        AddReward(-.5f);
    }
}
```

This method detects collisons. If the collision is of type boundary then it gets a minus reward.

Other scripts:

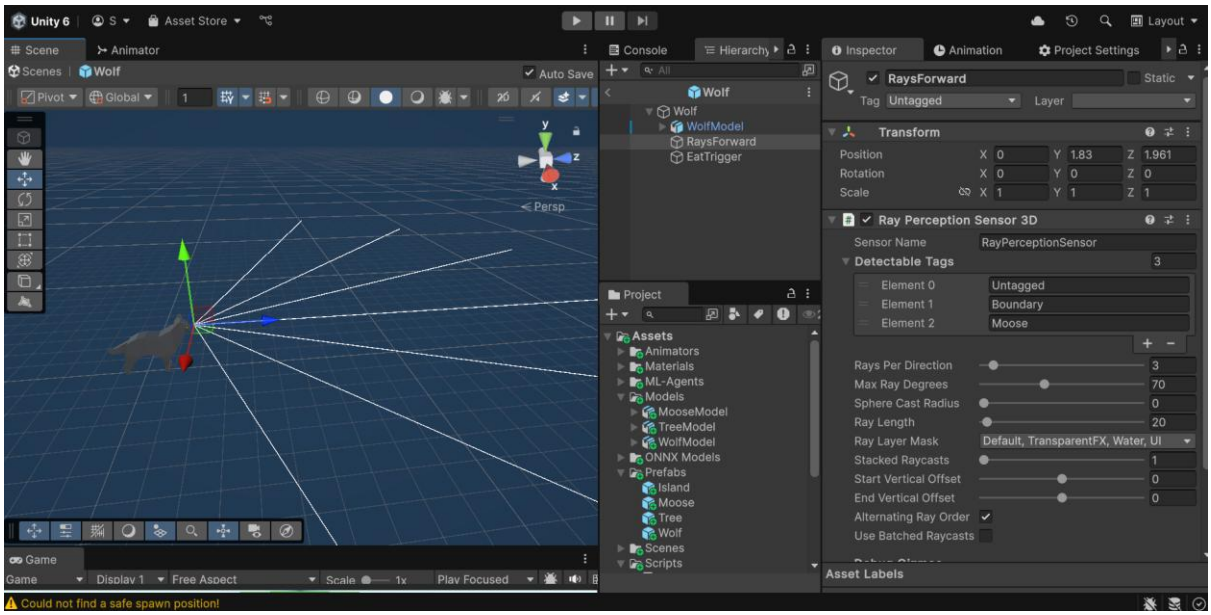
- **Moose Area:** Almost same script as tree area but spawns moose instead of trees so that the wolf can train.
- **Wolf:** Almost same script as moose with some minor differences in observations and penalties for bumping into trees. Also has some stamina features which are not implemented fully.

6. Agent config in unity

After creating all the scripts necessary, we need to do some stuff to our moose and wolf model.

Ray Perception Sensors

In the moose or wolf prefab, we need to add another empty game object INSIDE the moose or wolf game object. We will name it “RaysFoward”. To that object add a component and choose “Ray Perception Sensor 3D”. Then move empty game object to the front of the moose or wolf. This will serve as the animal’s vision. You should have something like this:



The rays can detected tags. The tags are mainly used for detecting untagged and boundary objects so it knows not to bump into them. These observations will also be fed to the neural network.

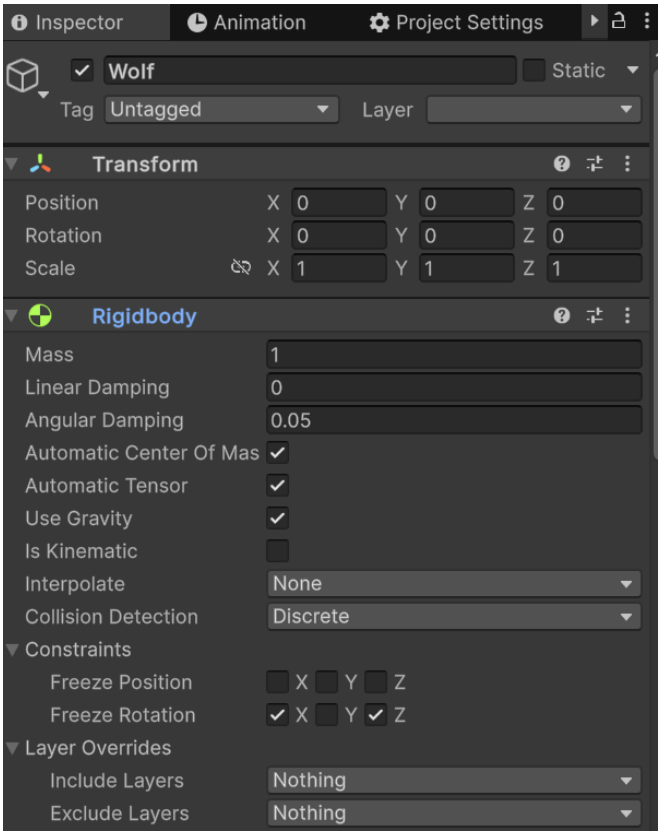
Adding components to the animal

Again on our wolf or moose we have to edit the prefab. Go to the empty game object that holds everything. In the inspector add the following components.

- Rigid Body
- Box Collider
- Behavior Parameters
- Decision Requester
- Wolf/Moose script if not added already

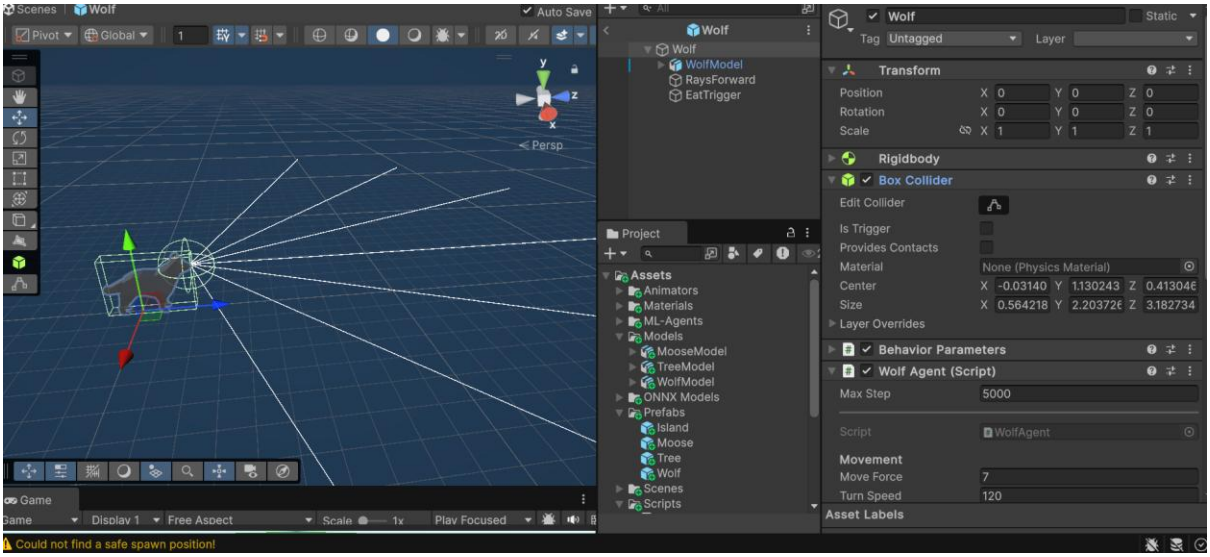
Rigid Body

The rigid body is what gives physics to our object. Once added you can leave it as is it should look like this:



Box Collider

The box collider is what will serve as a collider for our animal. It can just go around it roughly. You can edit it however you want to fit the animal. Should have something like this:



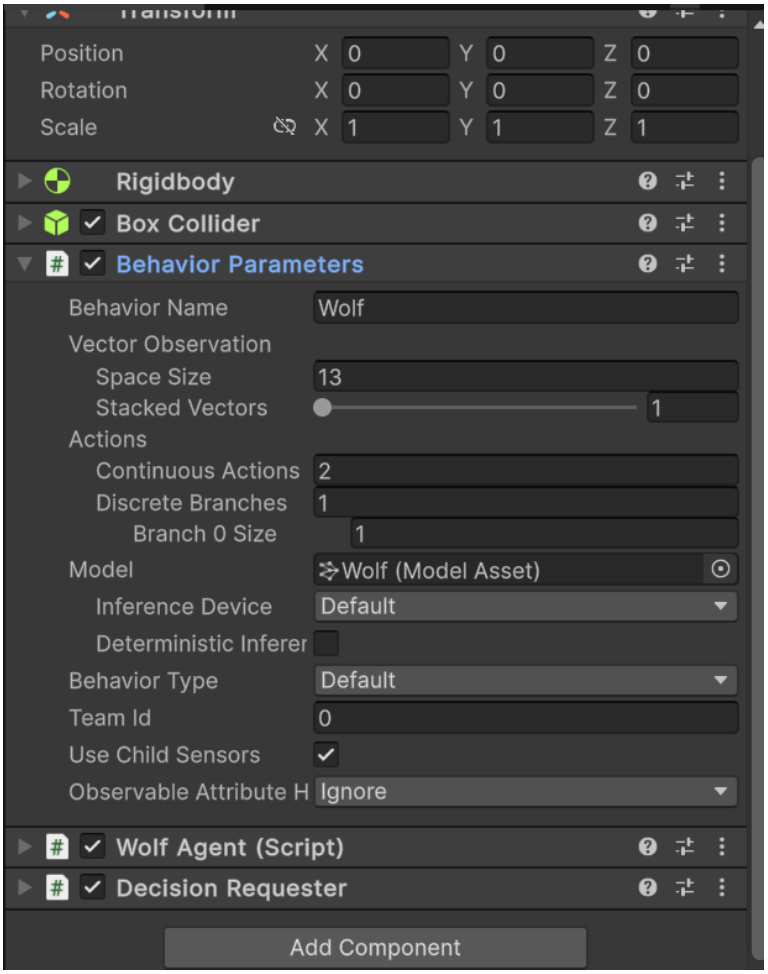
Behaviour Parameters

This should already be there if the script has been attached already. Since the script is of class Agent, it automatically also brings the behaviour parameters.

We have to set the following things:

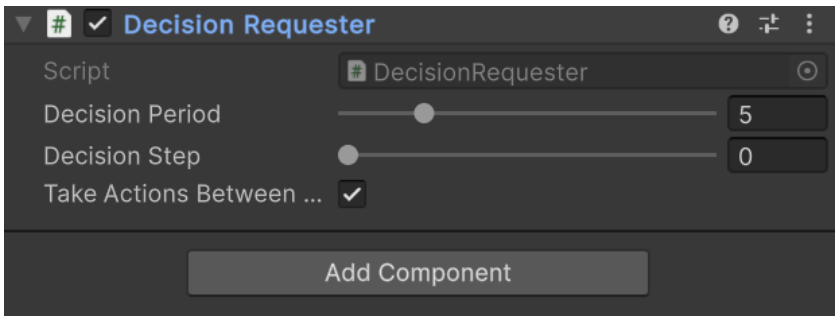
- **Behaviour Name:** This can be set to whatever but is important to remember the name since it will be used somewhere else aswell later on.
- **VectorObservation:** Here we have to set the space size to 13, since that is how many observations we are getting from the script.
- **Behaviour Type:** For training purposes this should be set to Default. If you would like to test out the environment then you can set it to heuristic. Meaning you can control the agent yourself.
- **Use child sensors:** This should be checked. This will enable the agent to use our sensors that we added to the wolf/moose model before.

At the end you should have something like this but instead of “Wolf” it should say “Default”:



Decision Requester:

We also need to add a decision requester. Without this our neural network will not make any decisions. Once added leave as is. It should look like this:



7. Training

Config File

Again, you could follow the video guide from which I learned how to do these things: https://learn.unity.com/tutorial/trainer-config-yaml?uv=2019.3&courseId=5e470160edbc2a15578b13d7&projectId=5ec83171edbc2a44309bf22e#

However if you decide to follow that guide the config will not work so I will provide you with the fixes down below.

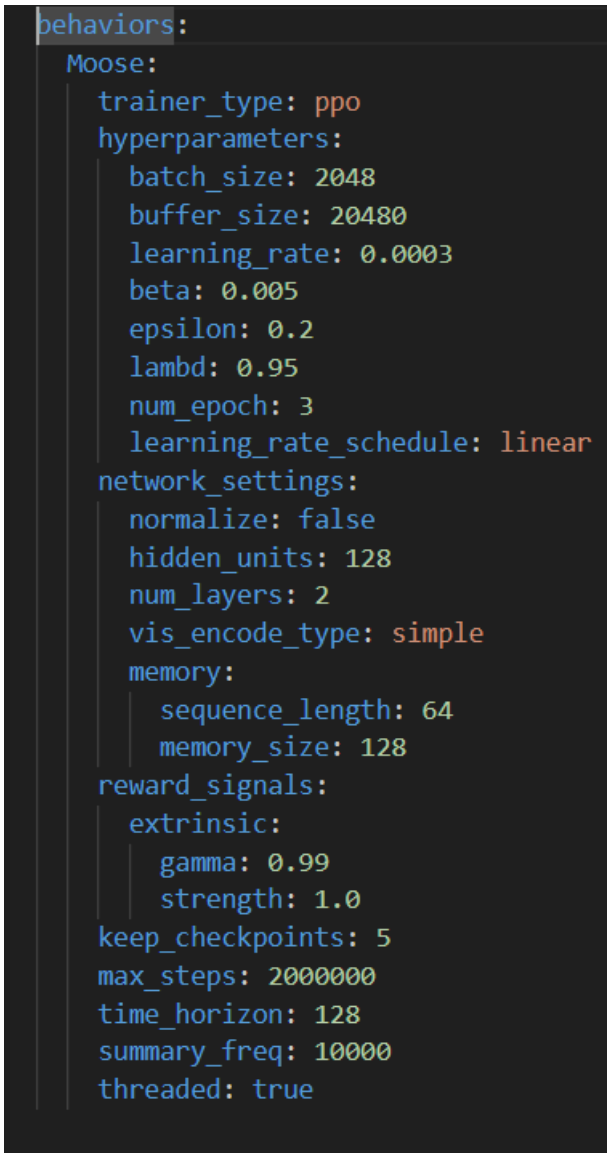
If following video guide:
After going through the community comments I found that the syntax of how you should setup the config file has changed throughout the time. This person posted the right layout:

> AS others have mentioned as well, the trainer config has changed since the release of this course. There is no such a thing as default and custom config profiles.

```yaml
behaviors:
  Hummingbird:
    trainer_type: ppo
    hyperparameters:
      batch_size: 2048
      buffer_size: 20480
      learning_rate: 0.0003
      beta: 0.005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: false
      hidden_units: 256 #128
      num_layers: 2
      vis_encode_type: simple
      memory:
        sequence_length: 64
        memory_size: 128
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 2000000
    time_horizon: 128
    summary_freq: 10000
    threaded: true
```

My guide:

We first need to create a yaml config file which will hold all the parameters needed for the agent to function.



```yaml
behaviors:
  Moose:
    trainer_type: ppo
    hyperparameters:
      batch_size: 2048
      buffer_size: 20480
      learning_rate: 0.0003
      beta: 0.005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: false
      hidden_units: 128
      num_layers: 2
      vis_encode_type: simple
      memory:
        sequence_length: 64
        memory_size: 128
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 2000000
    time_horizon: 128
    summary_freq: 10000
    threaded: true
```

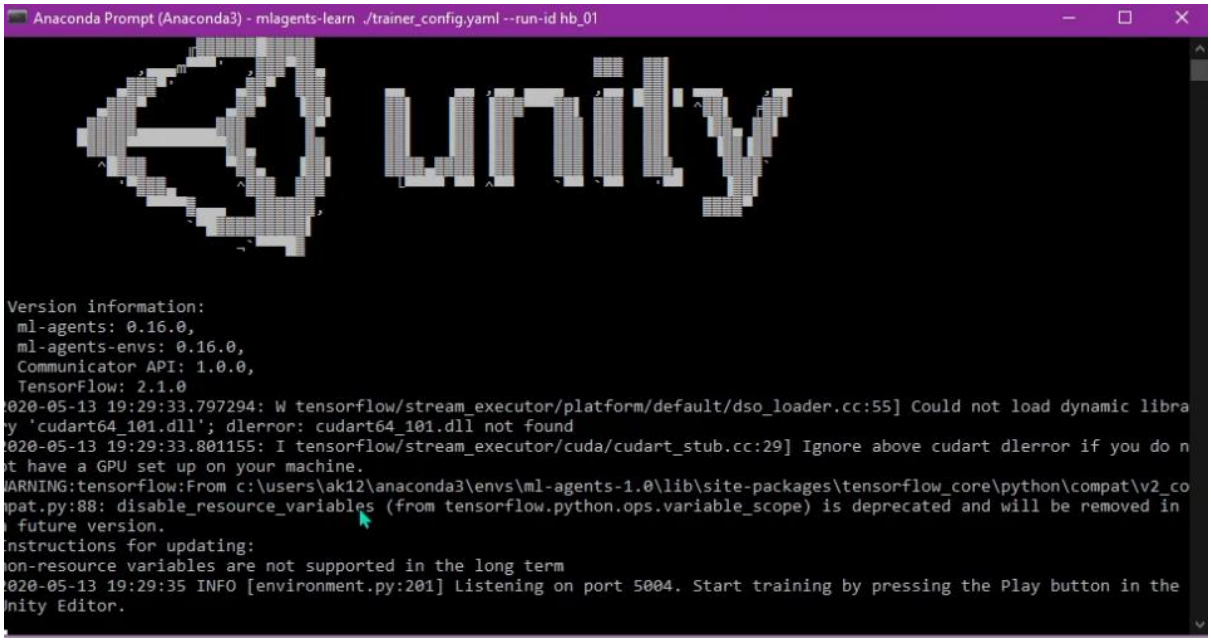
- **behaviors:** This is the top level section. We are defining a behaviour named Wolf/Moose. This is the name we set in unity.
- **trainer_type:** We set this to ppo. This means we are using something called Proximal Policy Optimization. A reinforcement learning algorithm that is commonly used. PPO basically means that our agent will try and find what actions to take to get a higher reward. It tries our actions, remembers what it did and how high it scored and then updates its brain policy.
- **Batch_size:** After collecting 2048 actions/observations/rewards, the agent will use that data to perform a learning update.
- **Buffer_size:** This is how much experience is collected before each policy update. It should usually be much larger than the batch size.
- **Learning_rate:** This controls how quickly the model learns. Lower values = slower but more stable learning. Higher = faster but more risky.

- **Beta:** Controls entropy regularization, which encourages exploration. Higher = more randomness/exploration.
- **epsilon**: Controls how much the new policy can deviate from the old one when updating.
- **Lambd:** Balances bias vs variance in calculating future rewards. Basically a higher lambda means the agent will think more about long-term rewards and if its lower it will think more short-term rewards.
- **Num_epoch:** How many times the agent learns from the same collected experience.
- **Learning_rate_schedule:** If set to linear this means the learning rate will slowly decrease over time to stabilize training.
- **Reward _signals:** This tells the agent how it's rewarded. Extrinsic means that rewards come from the environment.
- **Gamma:** This controls discounting of future rewards. A value close to 1 means the agent cares more about long-term goals.
- **Max_steps:** Training will automatically stop at this many steps.

Start training

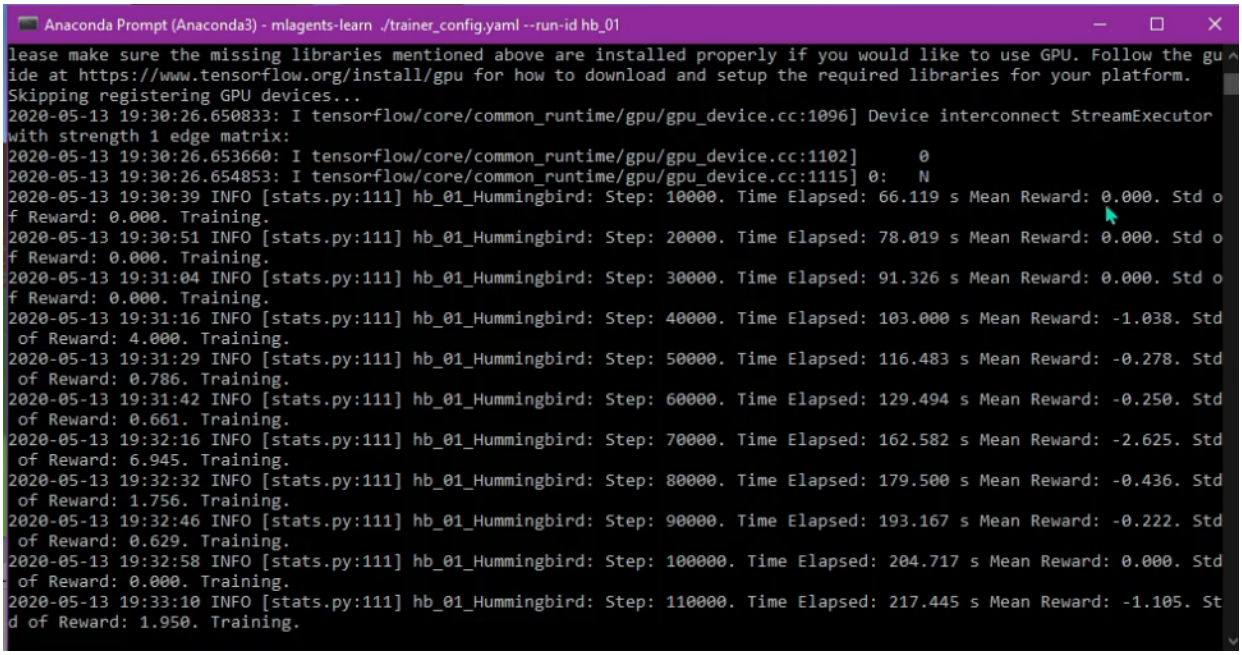
Now that everything is setup all is left to do is start training. In our anaconda prompt, in our virtual environment we need to navigate to the folder that holds our config file. Then we need to type this command:
mlagents-learn ./(name of config file).yaml ---run-id (name of run)

When you run this you should see a screen like this:



When you see this then go to Unity and click the play button. The the training will start and you can see the moose/wolf is moving on its own! Trying to figure out what do and in maybe an hour or two you will begin to see it eating trees or moose.

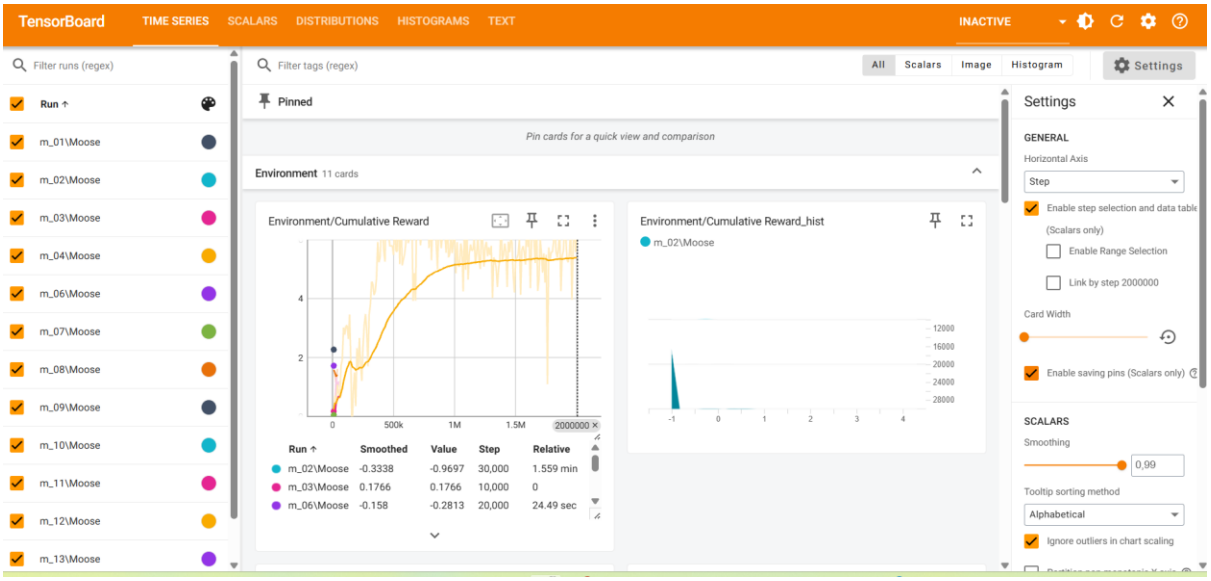
You can also see how the agent is doing in the anaconda prompt:



This is how the agent progress looks like in the console. But if we want a more visual approach we can open another prompt and run tensorboard to see our agents progress. In the new prompt we have to activate again our venv. Then navigate to our folder containing the config. Then run the following command:

tensorboard --logdir results

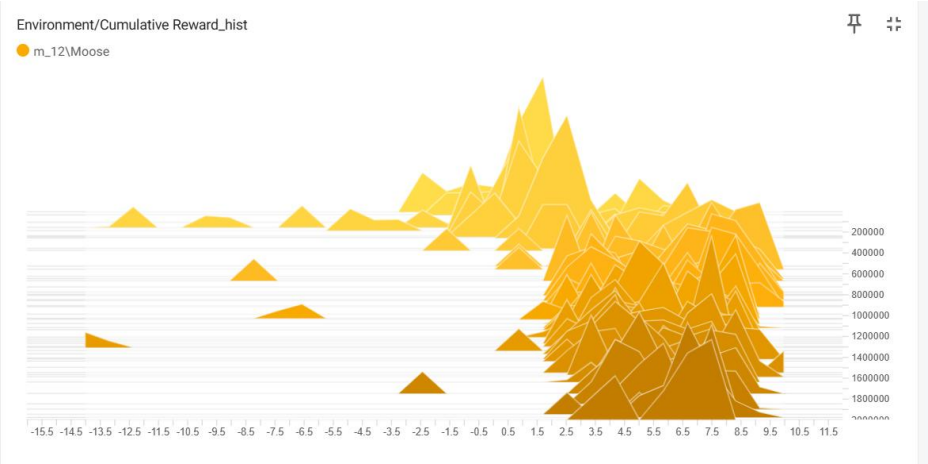
Then on localhost:6006 you can see all your runs and results represented in graphs.



You can see that I have around 14 runs and you can see one of them is quite better than others. Most of the first ones were test runs to see if everything worked. After leaving one run to train for longer we started getting quite good score. As you can see the moose

started to get a cumulitve reward of 6. Meaning it ate around 5,6,7 trees in one run before the episode ending.

We also get a nice historgram:



The x axis shows the cumulative reward per episode ranging from -15 to 11 in this case.
The y axis shows us the number of training steps. Ranging from 0 to 2 million.
The height of the spikes shows how many episodes fell into each reward bin. So from this histogram we can tell:

Early training:

The rewards were mostly on the left side meaning from -15 to 0. Meaning the moose has been mainly failing and starving or getting big penalties for something.

Mid training:

The distribution shifts right, with more episodes achieving positive rewards. This shows that the moose is starting to learn how to get a higher score.

Late training:
Th peak reward is around 5. We can also see now most of the episodes are on the right side. So we can see how it gets better and better the longer it runs.

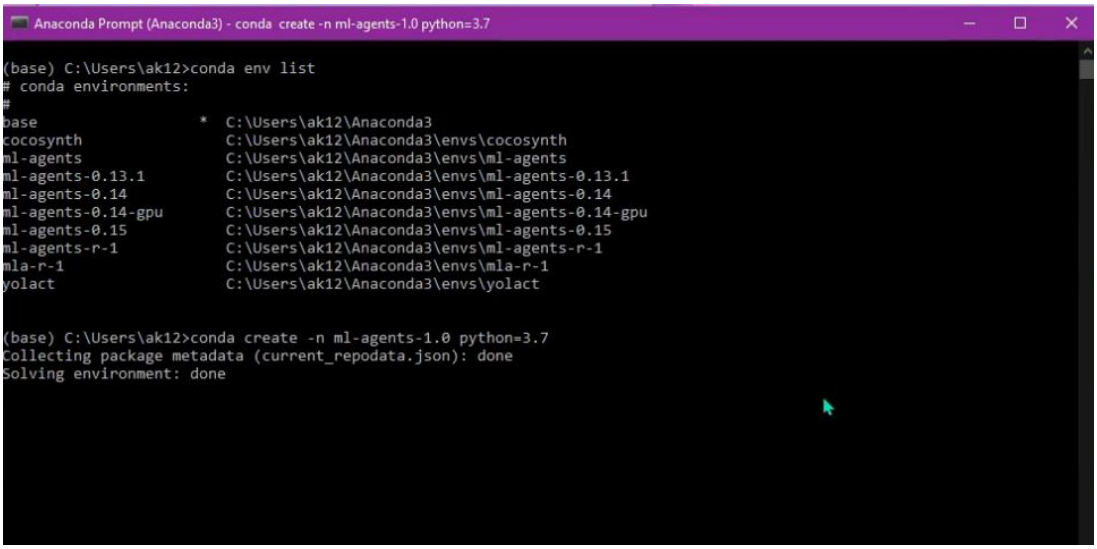
8. Problems I faced and how I tackled them

Originally, I learned how to do all these things through video course when I was doing my Proof Of Concept project. Here I will highlight all problems I faced while following the guide and also working on my own project.

While following the guide:

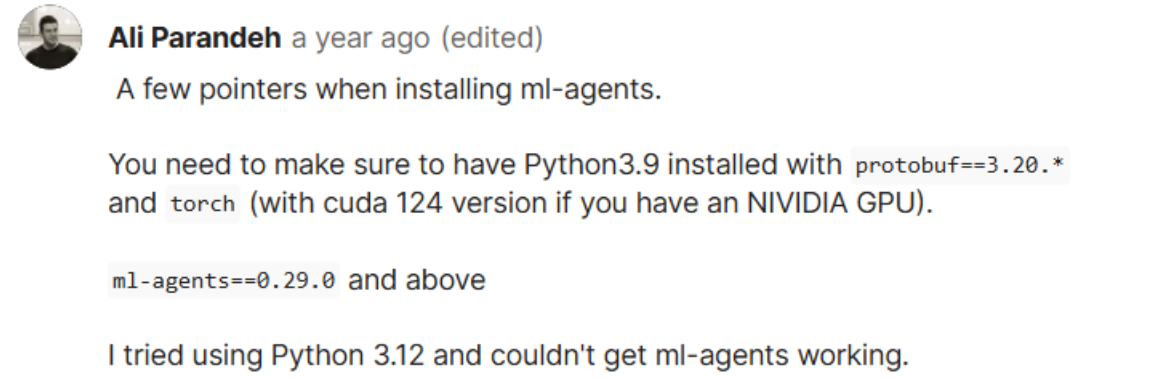
In the video guide I followed for my PoC there were a couple of problems that arose since the guide is very outdated and the library being used was updated.

- **Venv setup:**
  In the video guide they create the venv as such:



They use python 3.7 and install only the ml-agents package.

If you want to use the latest stable version then you have to use another version of python and need install some other necessary packages. Luckly I could look through the comments of the community about this guide and there I found a fix:



Ali Parandeh a year ago (edited)

A few pointers when installing ml-agents.

You need to make sure to have Python3.9 installed with `protobuf==3.20.*` and `torch` (with cuda 124 version if you have an NIVIDIA GPU).

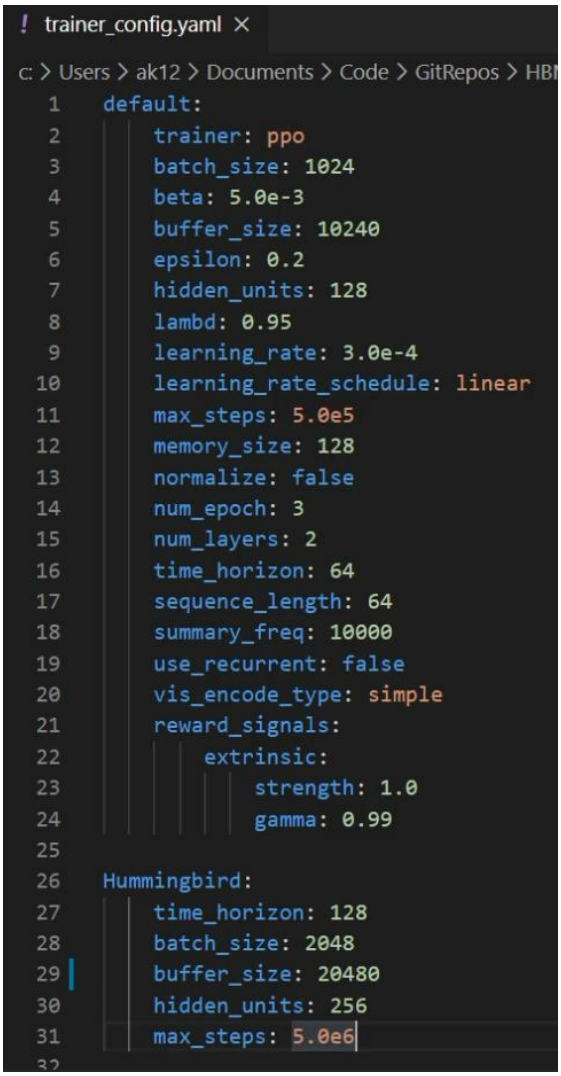
`ml-agents==0.29.0` and above

I tried using Python 3.12 and couldn't get ml-agents working.

Also you will be prompted to install onnx. Install onnx 1.16.1

- **Config file configuration**

  In the vdeo guide that I followed for the PoC the config layout was outdated. It was as such:



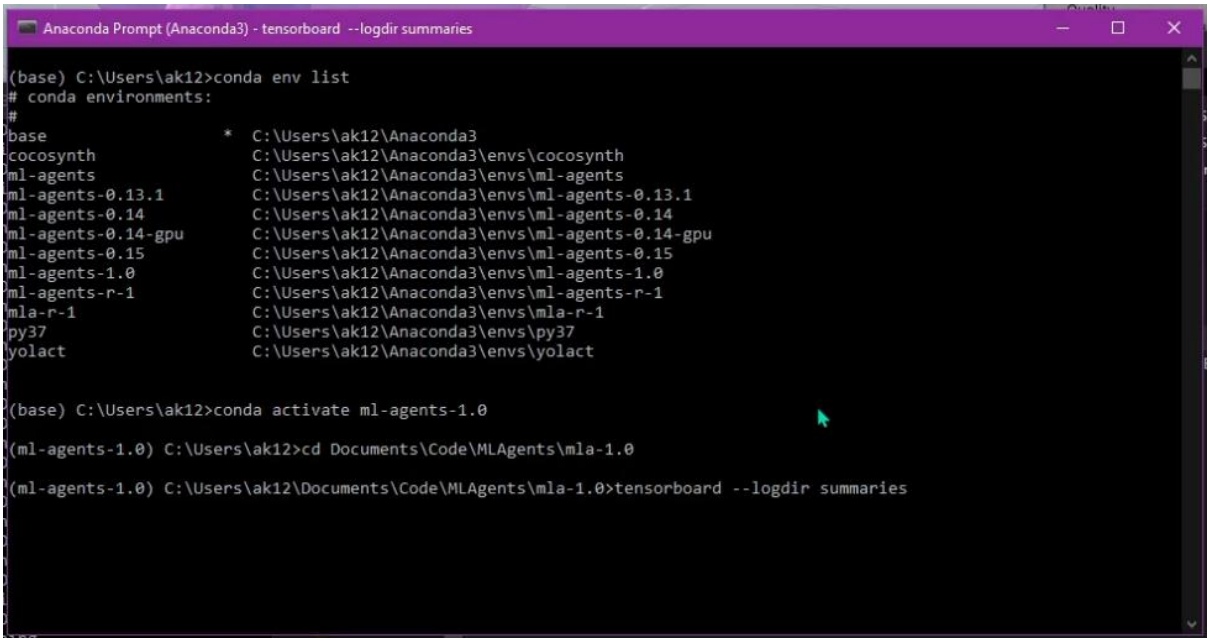
Again thanks to looking through the comments of the unity community I found a fix for the layout:

AS others have mentioned as well, the trainer config has changed since the release of this course. There is no such a thing as default and custom config profiles.

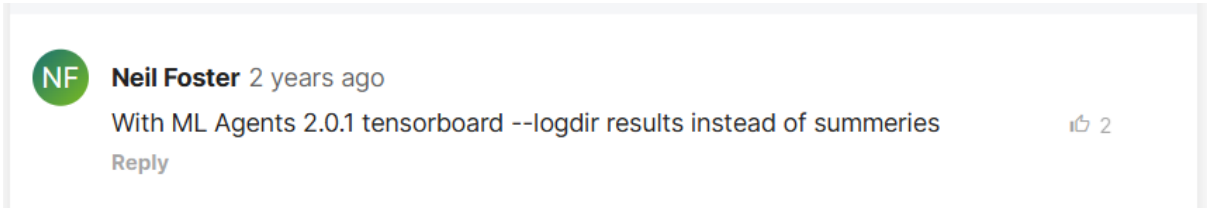
```yaml
behaviors:
  Hummingbird:
    trainer_type: ppo
    hyperparameters:
      batch_size: 2048
      buffer_size: 20480
      learning_rate: 0.0003
      beta: 0.005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: false
      hidden_units: 256 #128
      num_layers: 2
      vis_encode_type: simple
      memory:
        sequence_length: 64
        memory_size: 128
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 2000000
    time_horizon: 128
    summary_freq: 10000
    threaded: true
```

I also found another fix. You can use a command that can migrate your old format file into a new one: https://github.com/Unity-Technologies/ml-agents/blob/release_3_docs/docs/Migrating.md

- **Visualising results**



The tensorboard command being used is old. I found the correct command again looking through the comments of the unity community:



- **Reward System:**
The reward system I made for my PoC did not work at first. The agent was only getting rewards ithey hit a small reward (you can read more about this in my PoC document). I decided to try to give agents rewards for getting close to a flower instead of only getting a reward if hitting the flower. This made my agent actually start orienting itself but still would only hover over the flower.
Then in my actual project I decided to use the same reward shaping idea, but I made sure to add a way bigger reward on actually eating the tree. I also did some research on how to properly shape the reward function for my agents, so that they mimic animal behavior. You can read more about that in my Applied Research document.

9. Steps forward

What I want to try and implement next

- A graph showing how the population has been affected throughout the whole simulation. **(Implemented after the writing of this document)**
- Another graph for comparison based on the Lotka Volterra model **(Implemented after the writing of this document)**
- Reproduction behaviour
- (Optional) Water drinking behaviour and thirst.
- Create a better map

If I would continue this project after this semester I would try

- To see if I can optimize the training by adjusting some RL parameters. Get the same or better result for less training time.
- Implement better penalty for eating when not hungry
- Better camera movement
- Train another type of animal
- Add natural factors that can affect the behavior of the animal. Drought for example.