In this next part, we will create a tribe of Ethans that are going to learn to walk along our red plane. For this, we will need a way to manage a whole population of Ethans! In the assets space, create a new C# script and call it PopulationManager. Open this script and try to recall our PopulationManager in the previous project since this is almost identical!

Beginning with the Namespaces:

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using System.Linq;

Recall that using System.Linq allows us a neat way to sort our lists. Inside a Population Manager class:

public class PopulationManager : MonoBehaviour { }

We need to assign a prefab to populate our multiple Ethans—we have called it botPrefab as a reference to our Unity object:

public GameObject botPrefab;

public int populationSize = 50;

List<GameObject> population = new List<GameObject>();

public static float elapsed = 0;

public float trialTime = 5;

int generation = 1;

Here, our populationSize = 50 since there is no need to click on them as in the previous exercise. The higher our population, the better the training will be with genetic algorithms, however, you can have too many! Too many bots populating the scene will slow down the processing so keep it at 50 for now. Next, we create our population List to put our GameObjects in when they’re instantiated. Also, we have an elapsed time to keep track of a trialTime that will track how long each GameObject is alive. Here, 5 seconds should be enough time to fall off the red plane if a bot goes that way. Finally, we will keep track of which generation we are at as we cycle through.

Like before, we will create a GUIStyle:

GUIStyle guiStyle = new GUIStyle();

void OnGUI()

{

guiStyle.fontSize = 25;

guiStyle.normal.textColor = Color.white;

GUI.BeginGroup (new Rect (10, 10, 250, 150));

GUI.Box (new Rect (0,0,140,140), "Stats", guiStyle);

GUI.Label(new Rect (10,25,200,30), "Gen: " + generation, guiStyle);

GUI.Label(new Rect (10,50,200,30), string.Format("Time: {0:0.00}",elapsed), guiStyle);

GUI.Label(new Rect (10,75,200,30), "Population: " + population.Count, guiStyle);

GUI.EndGroup ();

}

Recall that this GUIStyle simply produces a box that will display some of the data we want to keep track of such as the population.Count. As to the GUIStyle, Unity have produced plugins based on this program called Canvas. You can simplify this program by referring to canvas instead—The GUIStyle

System is useful to know since it forms the basis of the updated canvas system used in editor plugins and packages. Understanding what lies behind many of the inbuilt Unity features is useful to know, so we will continue for now with building our own GUI.[[1]](#footnote-1)

The void Start will create our initial population, so inside this method we will add a for loop that will loop around a population size and each time instantiate a new bot (*i.e.* a new version of Ethan) from our bot prefab:

void Start () {

for(int i = 0; i < populationSize; i++)

{

Vector3 startingPos = new Vector3(this.transform.position.x + Random.Range(-2,2), this.transform.position.y,

this.transform.position.z + Random.Range(-2,2));

GameObject b = Instantiate(botPrefab, startingPos, this.transform.rotation);

b.GetComponent<Brain>().Init();

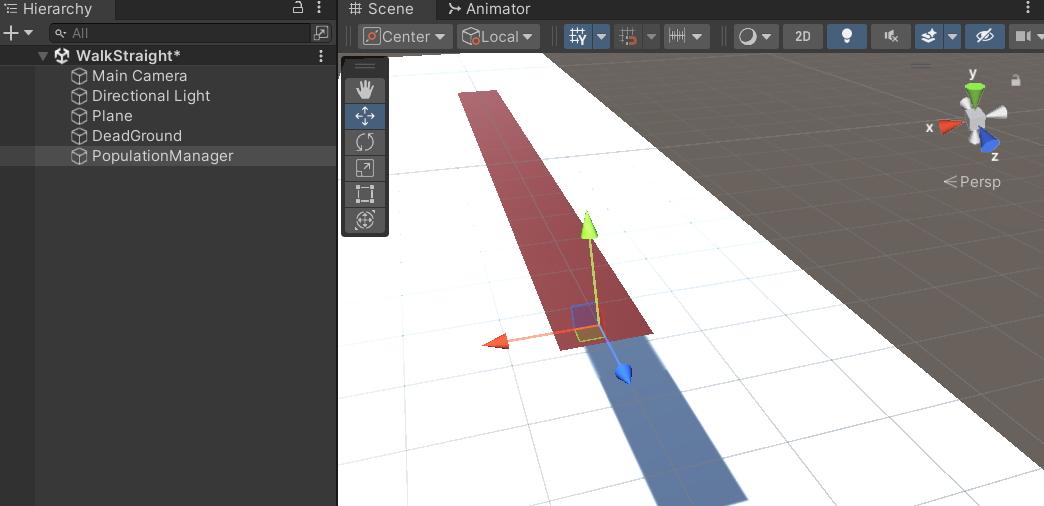
population.Add(b);

}

}

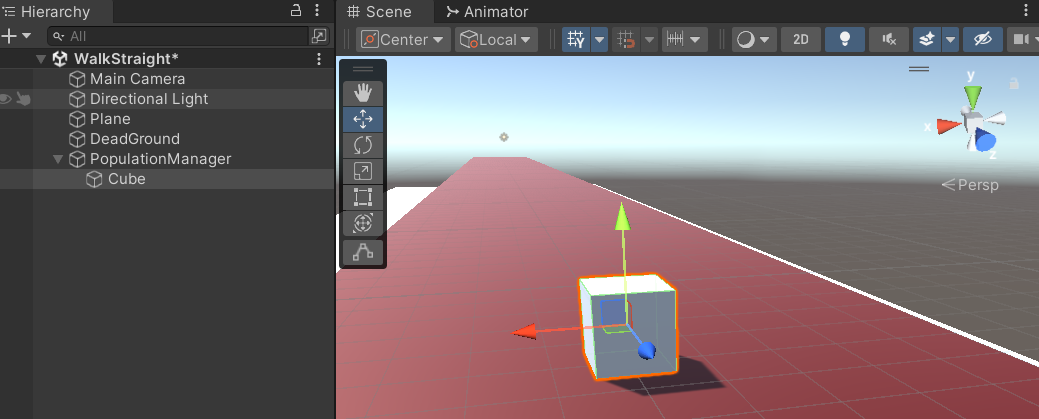
The rest of the program provides a position that a particular bot will be instantiated at, with some randomness in the X and Z directions. This is to avoid the bots all piling up on top of one another.

Before we proceed, ensure that when we select our population manager from the Hierarchy that the position on the Z-axis is approximately at the start of the red plane (z = 65 should be achieve this):



Right-click on the Population Manager and add a cube as a child that has a position of 0, 0, 0. This will show us exactly where the bots are going to spawn. Move the Population Manager up a tad if it sits in the middle of the plane. Your cube and PopulationManager should look like this:

A screenshot of a computer

Description automatically generated

You can turn off the mesh renderer, so that the cube disappears. Also, turn off the Box Collider since we do not want that cube interfering with the Colliders on our Ethans. While we are dealing with the collisions, we should make sure that Ethan cannot collide with other Ethans—colliding with one another will interfere with the training of the genetic algorithm since the physics system will report movements caused by external Ethans as movements coming from a particular Ethan. So, select the bot prefab in the assets window and in the inspector add a layer to Ethan:

A screen shot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

Using Layer 8 (some of the other Layers are reserved for internal Unity functions), name the Layer “Ethans” as pictured above. Now we need to set up the layers so they will not interact with one another. In the Edit menu, choose Project Settings > Physics > Layer Collision Matrix (scroll down to see the menu item). There is no neat way to say this, simply, we need to turn off Ethan being able to collide with Ethan:

A screenshot of a computer

Description automatically generated

So, we uncheck the box that enables Ethans/Ethans interaction. Back to our code, we have a starting position, and we are instantiating our bot prefab at that starting position at the same rotation the PopulationManager has. In the previous project, we placed a quaternion.identity here so we had a zero rotation for each prefab. In this situation, we do not need to rely on quaternions since if we need them to start facing away from us, then we adjust the blue axis of the PopulationManager to face the same way as he red plane.

The last bit of code in wrote in the void Start initialises each of our bots that we create by running an Init method that is attached to the Brain. This gives them a random value for their single gene, and then adds them to the population. Next, we build the Breed program that is similar to the previous project:

GameObject Breed(GameObject parent1, GameObject parent2)

{

Vector3 startingPos = new Vector3(this.transform.position.x + Random.Range(-2,2),

this.transform.position.y,

this.transform.position.z + Random.Range(-2,2));

GameObject offspring = Instantiate(botPrefab, startingPos, this.transform.rotation);

Brain b = offspring.GetComponent<Brain>();

if(Random.Range(0,100) == 1) //mutate 1 in 100

{

b.Init();

b.dna.Mutate();

}

else

{

b.Init();

b.dna.Combine(parent1.GetComponent<Brain>().

dna,parent2.GetComponent<Brain>().dna);

}

return offspring;

}

As before, we take genes from both parents and Combine them to create an offspring as well as a starting position for the offspring. The offspring is instantiated in the same way as the parents. We then access the Brain through Brain b = offspring.GetComponent<Brain. Next, we have a program that will control the mutations. We want to mutate 1 percent of the time, and when a mutation occurs, it will be initialised by the Brain and will have added to its DNA a random value. If it is not a mutation, then we simply initialise it! Finally, we Combine the parent’s DNA to make the DNA of the offspring and return that offspring so it can be placed into the new population.

Now, the second last bit of code for our we need will sort the population list to begin with, and using System.Linq commands, we can order the list by the time that the GameObject is alive:

void BreedNewPopulation()

{

List<GameObject> sortedList = population.OrderBy(o => o.GetComponent<Brain>().timeAlive).ToList();

population.Clear();

for(int i = (int)(sortedList.Count/2.0f) - 1; i < sortedList.Count-1; i++)

{

population.Add(Breed(sortedList[i], sortedList[i + 1]));

population.Add(Breed(sortedList[i + 1], sortedList[i]));

}

for(int i = 0; i < sortedList.Count; i++)

{

Destroy(sortedList[i]);

}

generation++;

}

For fuller comments on the function of this program, see the previous lessons. Here, we are simply creating a copy of the prefab from the spliced genes. And the final bit of code again resembles the Update in previous lessons:

void Update () {

elapsed += Time.deltaTime;

if(elapsed >= trialTime)

{

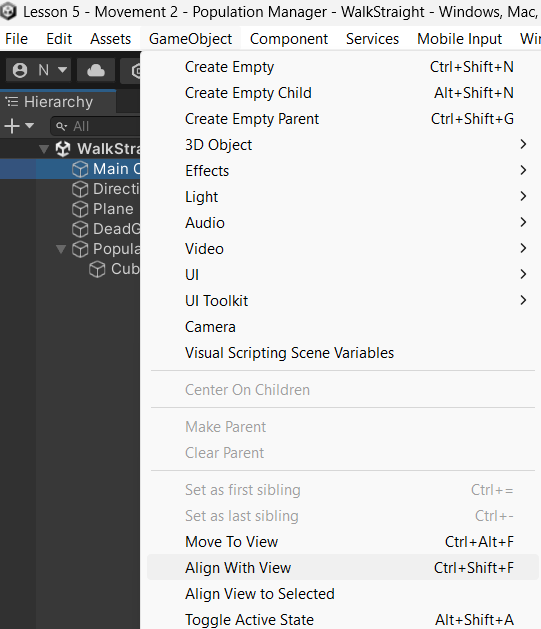
BreedNewPopulation();

elapsed = 0;

}

}

That is the complete PopulationManager! Switch back to Unity and take the PopulationManager script and attach it to the Population Manager GameObject. Choose the Bot prefab and before pressing play, align the main camera with GameObject by selecting the Main Camera in the Hierarchy view, then choose GameObject > Align With View:



A screenshot of a computer

Description automatically generated

Pressing Play you should see a population of Ethans created! As we can see, we get 50 Ethans created, and they all are given random commands. So, their gene is set to a value between 0 and 5. We can see that we have some jumping, some running, some crouching, and others falling off the edge. As our generations increase, we should eventually breed at least one Ethan that has learnt to survive. What we will find is, those that have learnt to live the longest have done so by either running in the “right” direction, or by crouching. Both good strategies to survive (even jumping is a legitimate strategy to survive)! Those that are running will eventually fall off the red plane, except that we are not allowing them to live long enough to figure that out.

Now it is time to experiment! Start by shortening the red plane or changing the population size. Then see what changes you think you might make. We should note that we cannot get a great deal of complex behaviour to evolve out of one gene. The power of genetic algorithms becomes more apparent when we begin to mix survival strategies such as colours, size, and other traits that we might utilise in a fight for survival. So, as we move forward with this project, we will build systems that have longer gene sequences in them, with more complex results from their combinations and different strategies that emerge from the different values of those genes together in one system.

It will therefore become increasingly important to review the explanations for the programs we will produce. For now, we have had a bit of fun with genetic algorithms. So, let us finish this off with a challenge!

1. See the document guide for guidance when using GUI: https://docs.unity3d.com/Manual/GUIScriptingGuide.html [↑](#footnote-ref-1)