At the heart of every genetic algorithm lies the gene sequence. As we saw in the coloured people project, it is an array of numbers that we keep that forms parent DNA that is then “stitched” together to produce offspring that receive a different sequence of these numbers. In our previous project, we used them to represent the colour of a character, now we will use the same technique to represent movement.

To begin, create new 3D project in Unity and bring in the Ethan Walker unity package that your teacher will make available to you. Add the WalkStraight scene to the hierarchy view:

A screenshot of a computer

Description automatically generated

In the Game and Scene windows, we will see a long, narrow red plane with a white plane underneath.

A screenshot of a computer

Description automatically generated

The whole purpose of this is to create a whole population of bots that start out on one end and learn to make their way to the other end. The bots will select from certain movements that they can perform while standing on the plane. If they fall off, then they will be classified them as dead (or unfit) so that only the ones that stay on the plane for the longest time be bred into the next population.

For our agent, we will use the in-built Ethan prefab Unity provides – if you drag him to the Hierarchy pane, he will appear on one end of the red plane:

A screenshot of a video game

Description automatically generated

If all is functioning as it should, you can remove Ethan from the Hierarchy pane. Ethan has been pre-prepared for this project, so briefly familiarise yourself with his inspector view. You should notice a few things; he comes with an Animator, Rigidbody, Capsule Collider, and a Third Person Character Script. We will make use of these properties in our DNA and Brain programs.

Now let us add a Population Manager GameObject to the Hierarchy. Next, create a new C# script named DNA and remove the contents except for the Namespaces as we are using a standard class (rather than a monobehaviour). This standard class DNA is a helper class that will be attached to a brain class we will create shortly:

public class DNA { }

Inside this method, we will need to create a list of genes that the DNA has, these will be like values we used in previous lessons such as the RGB colour channels. The main difference this time is that in previous lessons we hard coded the values for those colours. This time, we will make our program more flexible so that so that any sort of length of chromosome with genes can be created. Begin then, by having a DNA length value that we will keep track of along with DNA max values:

List<int> genes = new List<int>();

int dnaLength = 0;

int maxValues = 0;

These maxValues are used to set up the random values in the gene once it has been initialised. For example, let us say we create a gene and need its maxValues to be 10; when the object is created, a random function will only allow it to assign a value at as high as 10. Notice we are working with integer values? Each set of genes can have an integer value in any of the positions and we will see shortly how they are to be employed.

Next, we have a constructor for our DNA where the length of the strand of a gene is assigned (int l) and then a maximum value for that strand (int v):

public DNA(int l, int v)

{

dnaLength = l;

maxValues = v;

SetRandom();

}

Finally, a SetRandom() will run the method below to clear the genes and re-initialise them for the next iteration. So, the program loops through the list and sets them to random value between 0 and a max value:

public void SetRandom()

{

genes.Clear();

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

{

genes.Add(Random.Range(0, maxValues));

}

}

This method will become clearer shortly, once we see what it is exactly doing in Unity. Next, we have a SetInt that will set the int value at a particular position for the gene sequence:

public void SetInt(int pos, int value)

{

genes[pos] = value;

}

Following this we will create a Combine method:

public void Combine(DNA d1, DNA d2)

{

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

{

if(i < dnaLength / 2.0)

{

int c = d1.genes[i];

genes[i] = c;

}

else

{

int c = d2.genes[i];

genes[i] = c;

}

}

}

This method splits the sequence from the parent and then puts them back together. So, there are two parents that are going to contribute their DNA, half from each parent, so that the offspring are comprised of half from each parent. In our previous algorithm, we used colour DNA combined from both parents by randomly assigning it, here we split the DNA sequence and contribute half from each parent to an offspring making the new offspring’s gene.

Next, we need a method to control mutations:

public void Mutate()

{

genes[Random.Range(0, dnaLength)] = Random.Range(0, maxValues);

}

public int GetGene(int pos)

{

return genes[pos];

}

Here, mutations are set by a random value at a random gene position in that sequence. The public int GetGene allows us to can get any gene in that sequence at a certain position and then returns their value back.

This is the entirety of the DNA code so make sure to save your work (and save regularly along the way!), and we will begin our Brain program. Again, create another C# script in the Assets folder called Brain, open the script, and add two new Namespaces and a bit of code in square brackets that will give us access to the script already attached to Ethan:

using UnityEngine;

using UnityStandardAssets.Characters.ThirdPerson;

[RequireComponent(typeof (ThirdPersonCharacter))]

The Brain controls Ethan, and “sits” between Ethan and his DNA. It reads the DNA to determines what to do and then tells the actual character itself what to do. So, it functions like a real brain!

Next, create a public class method for the Brain:

public class Brain : MonoBehaviour { }

Inside this method we will create public variables:

public int DNALength = 1;

public float timeAlive;

public DNA dna;

First, we have an int for DNALength. The code is already in place to increase the complexity of the DNA but for the beginning of this exercise, so we can see what we can do with one simple value, we will start with a length equal to 1. Next, we have a timeAlive value and we need to create our sequence of DNA (that will at this stage be a length of 1).

Following this, we need something to grab hold of our ThirdPersonCharacter script, as well as a Vector and a Boolean value to help us move and jump respectively:

private ThirdPersonCharacter m\_Character;

private Vector3 m\_Move;

private bool m\_Jump;

bool alive = true;

Ethan is a pretty cool character, he has a wide range of complex emotions he can perform so we will use this to “hook in” to tell Ethan how to move and animate. Finally, create a boolean value alive = true since when he fails we will stop updating our timeAlive.

Next, we will need to record how long a character lived since we will use this as our measure of fitness, so create an on collision enter:

private void OnCollisionEnter(Collision obj)

{

if(obj.gameObject.tag == "dead")

{

alive = false;

}

}

Remember that the Ethan prefab has a Capsule Collider attached? It is important that this script and that collider are on the same gameObject so the script fires when Ethan in this case hits something called or tagged with "dead". From there, that character will no longer be considered alive (alive = false). This will stop his time alive from being updated.

So, which object has the dead tag? If you go back to Unity, when a character falls off the red plane hey will hit a white plane object called DeadGround. This GameObject has the tag “dead”.

A close-up of a computer screen

Description automatically generated

So, when the Capsule Collider comes into contact with the plane labelled “dead”, the collision is triggered that then sets alive = false.

Next, we have an initialisation (Init) that is called when we create a GameObject from our Ethan prefab:

public void Init()

{

// Initialise DNA

// 0 Forward

// 1 Back

// 2 Left

// 3 Right

// 4 Jump

// 5 Crouch

dna = new DNA(DNALength, 6);

m\_Character = GetComponent<ThirdPersonCharacter>();

timeAlive = 0;

alive = true;

}

So, within the DNA we are storing one integer—the gene is one int long—but within that int we will store 6 different values. A 0 in that value in the gene will command Ethan to move forward, a 1 will cause him to move back, 2 will move him left while 3 means right, 4 will make Ethan jump, and finally 5 will make him crouch. So, the DNA is initialised with a length of 1, but has a max value of up to six in it (from 0 to 5).

Next, we are using a Fixed Update (rather than the usual Update you might be used to). A Fixed Update is called in sync with the physics replacing the character controller that comes on the Ethan third person character:

private void FixedUpdate()

{

// Read DNA

float h = 0;

float v = 0;

bool crouch = false;

if(dna.GetGene(0) == 0) v = 1;

else if(dna.GetGene(0) == 1) v = -1;

else if(dna.GetGene(0) == 2) h = -1;

else if(dna.GetGene(0) == 3) h = 1;

else if(dna.GetGene(0) == 4) m\_Jump = true;

else if(dna.GetGene(0) == 5) crouch = true;

m\_Move = v \* Vector3.forward + h\* Vector3.right;

m\_Character.Move(m\_Move, crouch, m\_Jump);

m\_Jump = false;

if(alive)

timeAlive += Time.deltaTime;

}

}

The normal prefab of Ethan comes with a character controller that will allow us to move him with the WSAD keys. We will replace that with this bit of Brain programming that will automatically send through the arrow key movements to the character. All of this occurs in a bit of code m\_Character.Move. This call will send through vertical and horizontal movements, as well as crouching and jumping animations, all controlled by a gene. The program works like this:

If our solo integer stored in the DNA at position zero is a zero, then we move forward, which is a vertical movement of one. If that value is a one, then move backwards (a vertical movement of -1). If two, go to the left. If a three, go to the right. If a four, then jump. And if there's a five, we want him to crouch. We are converting what is stored in the DNA into movements. Then after calling the move, jump is set to false.

*Note: it may seem odd to randomly set the jump to false but without it, the program begins to behave unintendedly. This move only wants to jump once and then if the gene changes in the example, we do not need to keep jumping. Because of how the script attached to Ethan that we are going to automatically control is set up, setting m \_Jump = false; was found to be necessary. If you have created your own character with its own movement then this code is likely to be a lot simpler and you will not need to correct this unintended behaviour.*

Finally, if we are alive the last segment of the program updates our timeAlive. And that completes the brain program so make sure to save your work!

Back in Unity, Add Component to the Bot Prefab (*i.e.* Ethan) by dragging and dropping the Brain script onto the Bot. Select Ethan and ensure the inspector values are as follows:

A screenshot of a computer

Description automatically generated

We can see the DNA has a length of 1, and the alive time is initialised to 0. Next, we will create the population manager.