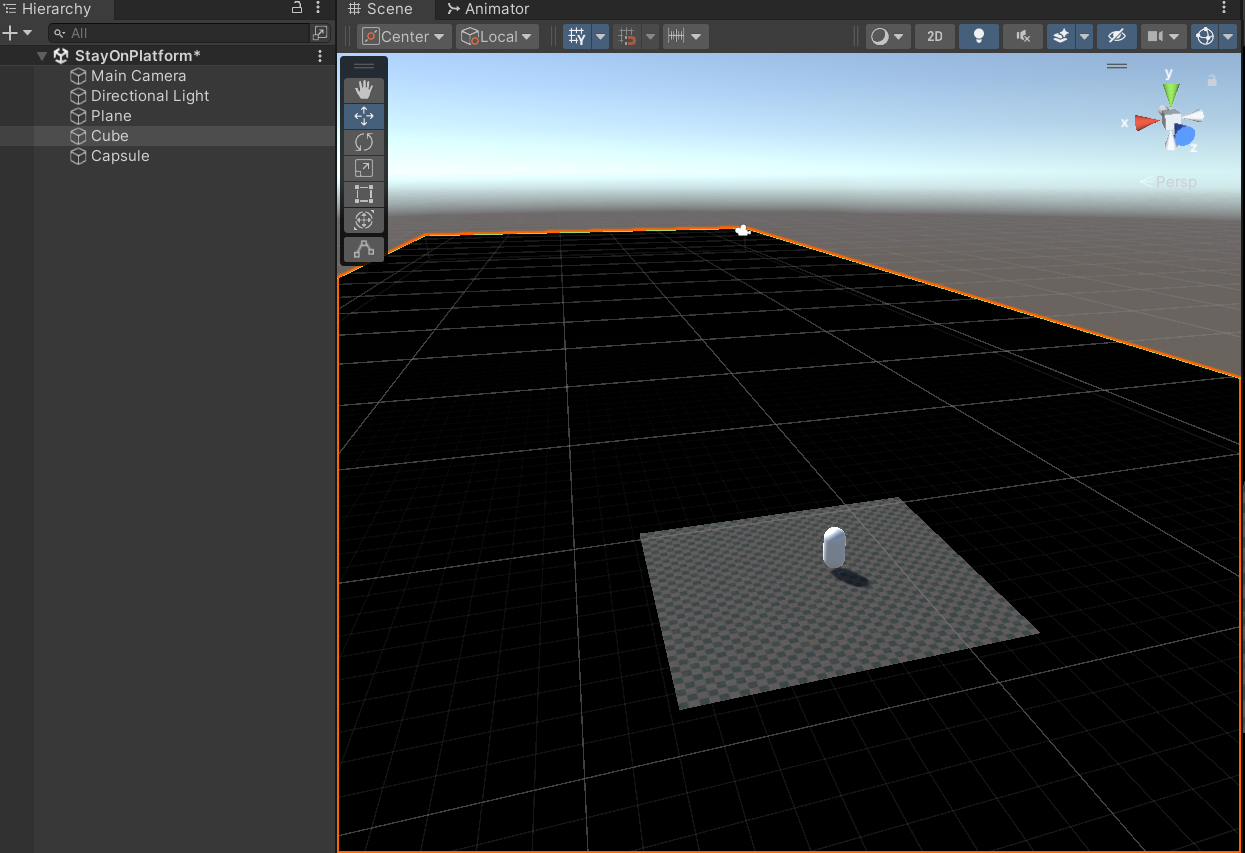
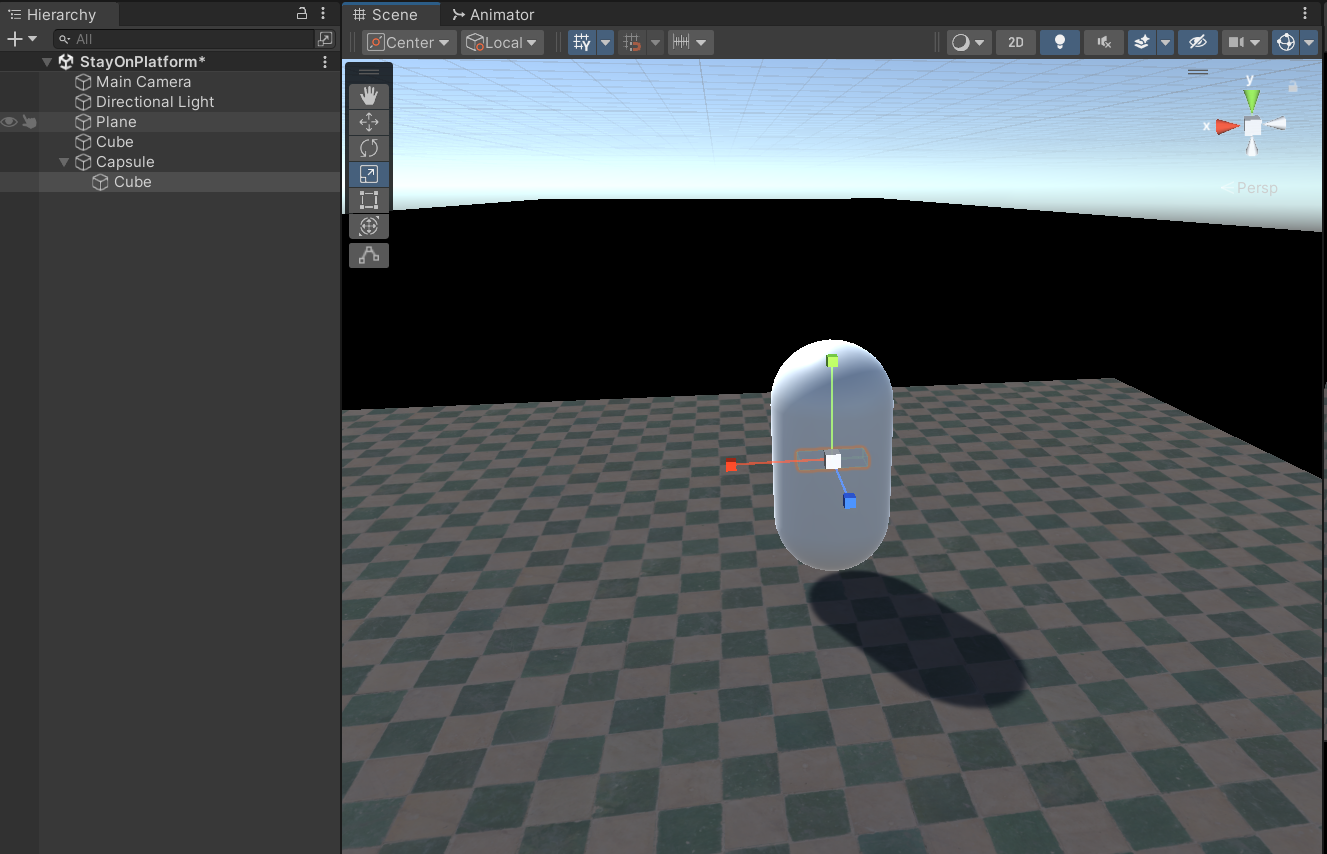
In this lesson, an agent will learn to stay on the platform using senses. From here, since we have done this a few times, I will describe what is to be done rather than supplying screenshots unless it is a menu we have not encountered. Create a new 3D Unity project and your teacher will make available to you a Unity package named “StayOnPlatformStarter2022” with presets to help us get started. Drag onto the Hierarchy, the scene named StayOnPlatform and ignore any warnings that may pop up from our Standard Assets. The Standard Assets gives us access to Ethan when we need him a little later in this lesson.

Here we will see a plane with a checkerboard pattern on where we will train our genetic algorithm to stay by giving our agents senses so that it knows where the edges of the platform are and to turn away if it gets too close. Besides this plane, there is a cube GameObject that will be our dead space around the platform so that if an agent falls off the edge, we can designate them as “dead.” Now, clicking on the plane, make sure you see it is tagged as “platform” as we will use this in the code to determine where our agent is and what is can “sense.” The cube is tagged ad “dead.”

Next, let us create our AI robot. To start, create a 3D capsule in the hierarchy window with a unit of 1 (or 0.1) depending on your default:



Let us give our AI some “eyes.” Select the capsule, right-click and create an empty cube as a child. This cube is going to act as the eye’s location and direction facing so select the cube and hit the “R” key and make the “eyes” narrower:



We will turn this off eventually, however, it is good to visualise the project this way. The eyes ought to be at the front of his body so hit the “W” key—this will show us where the front facing direction is, it is always our blue axis, that is, our blue facing transform. So adjust the position of the eyes so that it appears roughly where eyes should be:

A computer screen shot of a computer

Description automatically generated

The important aspect to these eyes are where they are looking—and since we are trying to detect if our bot is close to the edge, then they need to be facing down slightly. Hit the “E” key and this will allow us to rotate them down—ensure they are only rotated around the x-axis. We can also change the rotation in the Inspector view—about 47 degrees should be fine, and we can always adjust this if the bots are not scanning far enough ahead of themselves:

A white pill with colorful circles around it

Description automatically generated

Now, when our bot does detect the edge, that should be enough time to turn around without falling off into the dead zone. Rename the Cube in the Hierarchy to eyes and we can move onto beginning our code by creating a new C# script named DNA. This script will resemble the DNA script we made in *Lesson 4 – Movement*, so, the code should be familiar now and I will attach solutions to the bottom of this Lesson.

Save your work! Next, create our Brain script that will be different to previous Brain scripts. This time, we will have a DNALength of 2:

int DNALength = 2;

The reason the length has doubled is that we now have two decisions to make. Our agent needs to make a decision when it detects the platform and what it should do when it cannot see the platform. We also need to add the following:

public float timeAlive;

public DNA dna;

public GameObject eyes;

bool alive = true;

bool seeGround = true;

Within our DNA, we will have three different values. We have our timeAlive that will keep track of since when our agent falls from the platform it will be classed as dead and our fitness function is going to be based on how long we can stay on the platform and stay alive. Next, we have a link to our DNA dna set up, and we will also need a link to our GameObject eyes since this is what we use to determine whether we are at a precipice. Finally, we have two bool values alive and seeGround both set to true so that our agent knows if it can see the ground or not.

Next we will create our OnCollisionEnter similar to before:

void OnCollisionEnter(Collision obj)

{

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

{

alive = false;

}

}

If our agent ends up hitting a zone labelled “dead,” that black part of our map, then the agent is dead. Remember, this is based on Collision rather than what our “eyes” detect. collides with that black cube. Next is the Init that is simpler than before:

// Initialise DNA

public void Init()

{

// 0 Forward

// 1 Left

// 2 Right

dna = new DNA(DNALength, 3);

m\_Character = GetComponent<ThirdPersonCharacter>();

timeAlive = 0;

alive = true;

}

This is where we set up the DNA with one of three possible values. A 0 will, then it means move forward. A 1 means turn left, and a 2 means turn right. Now we can move into the update function. Get this code down, and I will explain its functioning:

private void Update()

{

if (!alive) return;

Debug.DrawRay(eyes.transform.position, eyes.transform.forward \* 10, Color.red, 10);

seeGround = false;

RaycastHit hit;

if (Physics.Raycast(eyes.transform.position, eyes.transform.forward \* 10, out hit))

{

if (hit.collider.gameObject.tag == "platform")

{

seeGround = true;

}

}

timeAlive = PopulationManager.elapsed;

// Read DNA

float h = 0;

float v = 0;

if (seeGround)

{

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

else if (dna.GetGene(0) == 1) h = -90;

else if (dna.GetGene(0) == 2) h = 90;

}

else

{

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

else if (dna.GetGene(1) == 1) h = -90;

else if (dna.GetGene(1) == 2) h = 90;

}

this.transform.Translate(0, 0, v \* 0.1f);

this.transform.Rotate(0, h, 0);

}

The first part of the Update function says that if (!alive) then return; out of the function. So, if our bot is dead, there is no need to continue processing the remainder of the function. Next, by using Debug.DrawRay we can see where the bot is looking as it will draw a red line form the eyes in the forward direction along the forward vector. This will allow us to see where the bot is aiming its gaze. To get this to work we need to pass four arguments— (eyes.transform.position, eyes.transform.forward \* 10, Color.red, 10)—including a colour and a length. After this, we set our bool seeGround = false. So, we create a Raycast and a Physics.Raycast using the eye position. The eye has a forward length of ten and then places the result into the RaycastHit variable, which is called hit. This means that if the thing that we do hit is the platform we are walking around on, (since we have the hit.collider tagged as the "platform") then we say seeGround = true. This is how our agent knows it is on the platform. Next, now that we know whether we can see the ground or not, we can then use our DNA to determine what to do.

The remainder of the code is as we have produced before so see my explanations in the previous lessons. From here, we will move into building a population manager.

// DNA Script

public class DNA

{

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

int dnaLength = 0;

int maxValues = 0;

public DNA(int l, int v)

{

dnaLength = l;

maxValues = v;

SetRandom();

}

public void SetRandom()

{

genes.Clear();

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

{

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

}

}

public void SetInt(int pos, int value)

{

genes[pos] = value;

}

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;

}

}

}

public void Mutate()

{

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

}

public int GetGene(int pos)

{

return genes[pos];

}

}