Things I’ve learnt!

I learned to use 'Invoke Repeating' to create a function that is called after a specific amount of time has passed. This method allows me to reference my function by string and set both the initial time for the function to be called and the duration between subsequent calls. I found this technique in the Unity Documentation!

Raycasting involves a point where the ray hits an object collider, allowing me to manipulate the collider's behaviour upon raycast contact. This was a new technique for me, but it worked! I learned how to use raycasts by watching a Unity tutorial and exploring the Unity documentation.

I've successfully learned to create an interface, which enables the implementation of methods without bodies across different classes. This capability allows for the sharing of common behaviours among multiple classes. Essentially, an interface defines a set of protocols that different classes follow consistently. This was a completely new concept for me, taught through a Unity tutorial!

I recently learned how to create a Navmesh and configure an agent within it to follow the player across the Navmeshed area. This was a completely new concept to me until I delved into Unity documentation and YouTube tutorials on creating AI that tracks the player's movements. Through this process, I discovered the utility of 'Navmesh.SamplePosition,' a method crucial for agent navigation within the Navmesh area. This method helped me identify points on the Navmesh where my target could navigate.

Moreover, I explored the 'Random.onUnitSphere' function in Unity, which generates a random direction in a 3D space using Vector3. This function allowed the target AI to select a random point to move towards, enhancing its behavior.

Another valuable function I encountered was 'Navmesh.AllAreas,' which considers the entire Navmesh surface area of your map. Understanding and utilizing this function became integral to leveraging the full potential of the Navmesh for my AI navigation. Enums, I never knew you could make a list of different functions until I read up on it on a forum, this allowed me to group different functions my character can have and then creating a case statement to define what each function in the enum list can do!

Fixing the canvas to fit all screen sizes, just had to anchor the points of the UI elements as well as set the canvas to fit with screen size rather than pixel by pixel.

By using interfaces, I grasped the concept of utilizing lists that adhere to the protocols set by the interface I defined. This enabled me to manage various game objects that followed the interface rules. I gained this understanding from Unity documentation and forums. Using the '.add' and '.remove' functions, I can dynamically add or remove items from the list. Furthermore, accessing different objects within the list by their index allows me to manipulate them based on their position in the list.

I discovered the importance of using Null checks to prevent program crashes by verifying whether something is initialized. This practice allows me to perform null checks, ensuring that variables are not empty. Initially, I wasn't familiar with the concept of Null, which essentially means 'empty.' Therefore, when I use '!= null,' it signifies 'is not empty.

I discovered the capability to create a sphere within a designated area that could gather information about its potential placement on a game map, including position and size. This revelation came to me after establishing a boundary for game object placement. Specifically, I utilized 'Physics.OverlapSphere' to define this boundary, enabling the regulation of distances between instantiated game objects in the game world.

I have also learnt to use the modulo function which allowed me to get a whole number of values as well as it is allowing me to cycle through an array, in my context, if I had a set of 4 prefabs in my array and wanted to spawn each one of them into my scene, I would iterate through the list using modulo by a variable which would consider how many had spawned by the length of my array. In the first cycle, the variable will be at 0 and will undergo the modulo function (%) against the length of my array (4) which would end the result in being 0. This would then mean the 0th value in my array will be selected. In the next iteration, the variable will increment to 1, which would lead to the following calculation: 4/1 which gets the value 1 which means the object in the array with the index 1 is selected. This essentially helped me iterate through my list.

In one of my tutorials, I used tags as a way to manipulate enemies by searching through tags, I didn’t realise I can also use layers which does the exact same thing, as well as this, the problem I was having was when I instantiating an object, (the enemies) the flashlight would destroy the tagged enemy original prefab that I have used to clone in my scene to destroy as well as randomly destroying tagged enemies. So instead of using tags, I had to use layers as well as initialise a variable which will hold the targeted object I need to destroy. Afterwards I had to rewrite my raycast protocols, which considered the enemy layer the raycast can affect as well as initialise the raycast’s hit variable to the target enemy I created above my class. This allowed me to assign which target is currently being pointed at depending on where the variable “hit” is facing. Therefore, allowing the correct gameobject to be manipulated by the raycast instead of the random targets from earlier. I learnt this from a friend who helped me out on the bugs I had which helped me understand a better way to create the same logic as well as fix the logic!

Another New skill I learnt through this coding journey, is that you can create gizmos that basically draw your raycasts, so that there is a visual representation of the rays so that you can see the direction your raycast would be facing. The method “OnDrawGizmos”, you can use the “Gizmos.Draw” function to draw the ray you are using. This is extremely helpful in debugging as well as having a representation of the rays in your scene. I learnt this from various unity forums and seen an exemplar on YouTube on how it works.

Something I have never tried before was using inheritance from different scripts, I never knew I can access my methods by using “protected override void” allowing me to inherit and modify functionality from a base script to create specialized behaviour within a derived script. I even learnt that I could override essential methods such as update and start, allowing me to create unique behaviour in my scripts. “Base.Start()” allowed me to access and execute the start function from the base class I am inheriting from. The last thing I learnt is that I can also access privatised variables in a class I had inherited by adding “protected” allowing it to be privatised – yet accessible to the new class I have created that is inheriting from the base class!