Exercises - Game AI

For each of the below exercises, you can find an example implementation in the game-ai.unitypackage asset. You are encouraged to solve the exercises on your own without having to refer to the asset. However, if you are stuck or looking for inspiration, the asset is available. The asset only contains very basic implementations. You are welcome to extend the functionality with your own ideas.

NB: If you see a purple box when importing the asset, it is because you haven't yet imported the Probuildre package from the Unity package manager.

Exercise 1 – Following Enemy

For this exercise, you will create an enemy that follows the player character.

Import the Probuilder package and set up a basic environment in an empty unity scene. For now, the most important part is a ground/plane that the characters can navigate. You will be adding to this environment in later exercises.

Create an enemy object and a player object in your scene (capsules are fine). Implement basic movement (WASD) for the player (or reuse an existing character controller created earlier in the course). Now, create a script on the enemy that makes it follow the player wherever they go and, once within a certain range, attack the player (a simple "attack" print to the console is sufficient in this case). Make sure that the enemy is attempting to face the player (z being the forward-facing axis of the enemy). You can give your enemy some defining features to easily tell which direction it is facing.

Hint:

The MoveTowards, RotateTowards and Distance methods on the static Vector3 class might come in handy here as well as Quaternion.LookRotation when you have to perform the rotation of the enemy.

Exercise 2 – Ranged Enemy

In this exercise you will be creating a new type of enemy, a ranged enemy. This enemy should exhibit a retreating behaviour as soon as the player character gets too close. (that is, move away from the player if the player is within a certain distance). While moving away from the player, the enemy should face away from the player. While not retreating, the enemy should attempt to face the player and shoot towards them with whatever ranged weapon you can think of.

Hint:

The movement behaviour is very similar to the "following" behaviour. Modifying the speed and changing the distance check should be sufficient to get close to the result that you are looking for.

You can choose to implement the shooting behaviour with a simple timer in Update or as a coroutine.

Exercise 3 – Patrolling Enemy

In this exercise, you will create a patrolling enemy that moves between predefined positions in your game world. Make it so that the patrolling positions can be defined on the component in the editor. When the

enemy arrives at a position, they must wait for a little while. This duration must also be configurable on the component.

Hint:

The patrolling behaviour is a good fit for a coroutine.

"yield return null" yields execution of the coroutine until the next frame and can be used, together with a while loop, to model movement logic over time, similar to in an Update loop. If you want to wait for a specific amount of time, the WaitForSeconds class might come in handy.

Exercise 4 – Lookout Enemy

This enemy will be looking for the player by rotating around and "looking out" in the direction that it is currently facing. If it sees the player, it will print "player spotted" to the console.

Create a rotating enemy (either rotating constantly or, even better, rotating partly at random intervals).

The enemies "Line of Sight" should be visualized through a green colored line rendered from the eyes of the enemy and forwards with a length equal to the vision distance of the enemy. The vision distance should be configurable in the editor.

As an optional addition, when the enemy sees the player, it will alert its friends, spawning a horde of the "following" enemies to hunt down the player.

Hint:

Utilize Physics.Raycast to determine if the player has been spotted. You can use Debug.Drawline (works only in scene view) or a LineRenderer component to visualize the vision of the enemy.

Exercise 5 – Pathfinding Enemy

The behaviour of this enemy will be similar to the patrolling enemy in the sense that it will be moving around our scene towards either static or dynamic target positions. The big difference is that this AI behaviour will not take the direct route towards the target if there are obstacles in the way. In other words, it knows how to find paths in the environment.

To work with this AI, we will need to add multiple obstacles to the scene. Focus primarily on static obstacles that are part of the environment, but try also to include some dynamic objects through the NavMeshObstacle component.

Once you have created the environment in Probuilder, bake it to a NavMesh using the Navigation window. Feel free to tweak and toy around with various baking properties.

Now you can add the NavMeshAgent component to your enemy and configure various properties such as speed, radius and height of the agent.

Add a script to the enemy that makes it possible to assign a target Transform that the enemy must chase down the position of.

Turn the enemy into a prefab and spawn multiple of them in the game world.

Configure your enemies such that one agent can traverse areas that other agents cannot.

Exercise 6 – Your Own AI

Combine or tweak some of the principles you have seen in the previous exercises to create your own customized AI behaviour that you can use for your course project.

You can look up information and inspiration about even more AI behaviours here.