***Assignment 2 Writeup***

Implementation of the guard

# In order to meet and brief specification I created an easily adaptable behaviour tree which would allow me to easily add and remove any new behaviour into the guard. As Tommy Thompson explains in “Behaviour Trees: The Cornerstone of Modern Game AI | AI 101”, “A behaviour tree is a data structure in which we set the rules of how certain behaviours can occur and the order in which they would execute”(AI and Games, 2019), which makes them ideal for a guard AI as a real guard would have a strict set of rules which determine their actions when on patrol. In order to create such a structure, researched the different types of nodes present in a behaviour tree. As explained in and video there are 4 main types of node in a behaviour tree and I chose to use three of these in my implementation.

# The first main node is the root, which simply acts as the start point for the tree and is typically a selector or sequence node, this is the only node in a behaviour tree which has no parent as it is the node which begins the tree and therefore cannot be triggered by another node.

# The second main node is a leaf node, which has only a parent node (The node which triggers it) and has no children (Nodes it can move to). Leaf nodes are “where we put AI behaviours” (AI and Games, 2019), which means that they hold no logic for deciding behaviours and instead contain the actions of that behaviour.

# The final type of node used in my implementation is a composite node, the video defines this as a node “which executes a child based on some logic in and world” and contains a parent and children nodes. Specifically, in my implementation I used a type of composite node known as a “Selector” node, which executes one of its children depending on the condition, to then

# There is another type of node not used in my implementation known as “Decorator nodes”, AltDevBlog’s “Introduction to Behaviour Trees” defines these as “typically have only one child and are used to enforce a certain return state or to implement timers to restrict how often the child will run in a given amount of time or how often it can be executed without a pause.”(AltDevBlog, 2011), which makes them useful for agents which need to either repeat actions on timers or limit the number of times an action such as an idle action can be selected within a timeframe.

Visualisation of the guard’s behaviour tree

# For the structure of my Node classes I used the diagram of my behaviour tree to determine that the types of node my selectors should link. This led to each selector having another as a child and then a sequence which tests a left node for its criteria.

# In my implementation, each guard has its own instance of the same behaviour tree structure and uses and same set of rules based on other aspects of the scene such as the spy’s location which could be achieved through a blackboard structure similar to the one found in Unreal Engine 4. This blackboard structure allows for agents to run individual versions of the same behaviour tree while sharing select information across all versions of the tree.(Epic Games, Date Unknown) Other rules include if the spy is hiding, if the guard finds an open door and the guard’s hunger in order to determine the behaviour it should be performing each frame. The priority of each behaviour is based on its height in the tree, with the top rules being the highest in priority such as if the spy is visible. This also makes the tree highly adaptable as adding new nodes and behaviours is done by replacing, adding or moving nodes in an existing tree, which can be made easier with a visual representation of the current tree in order to identify where the behaviours would fit in.

# Once the basic outline of the nodes was in place for the guards, the tree then needed a priority for these rules to fall into. Without a coherent priority to the tree, the way in which the AI acts would be incorrect, as it could prioritise getting food over chasing the spy, an action a security guard would never take. This priority was eventually built around the idea that capturing the spy will always be the priority. With this the check for if the spy is in the guard’s field of view became the top check in the tree. Checking if the guard has a line of sight on the spy is a two-part process, the first is a visual cone sprite which has a collision box attached. When the spy overlaps with this box it triggers the second check to be carried out for as long as the spy is within the cone. The second part of the process is a raycast line between the front of the guard sprite and the spy, if the line is able to hit the spy (Identified by a GameObject tag) then a visual line of sight has been made and the guard can chase and spy, otherwise the visual cone has passed through a wall which and spy is on the other side of and as such the guard cannot actually see the spy.

# Next came the check for if the spy has recently been spotted falling next the allowing guards to move to the spy’s last known location, which was a simple Boolean if the spy leaves the cone of view while being pursued, which then sets another Boolean that keeps the guard moving to the location until it has arrived or spots the spy again.

# A guards next priority would be investigating any open doors they can see, which is performed through a method like that of the spy being spotted however it will trigger if an open door enters the guard’s cone of sight. In my implementation guards will always close any door they pass through while the spy will leave them open, meaning any door found open has been passed through by the spy.

# Finally, this was followed by a check of the guard’s hunger, which decreases every few seconds. If it is low enough, the guard will make the kitchen its new destination and move there unless another condition is met, whereupon its hunger will be reset, and it will move back to its patrol route. If none of these requirements are met then the guard will continue to patrol between its fixed set of patrol waypoints until something alters its behaviour.

# To drive the movement for the guard I decided to use an existing a\* pathfinding library for Unity. The library allowed for easy implementation of the guard’s patrol and chasing movement while also allowing the guards to avoid obstacles and walls, even whilst chasing the player. In addition to this, the pathfinding system also allowed for the guard’s speed to be altered depending on the situation. This allowed for the guards to have a slower speed while patrolling and an increased speed while chasing the spy.

Implementation of the spy

In comparison the spy implementation uses the basic premise of a finite state machine. Given the limited range of actions the spy needs to take this was the ideal solution as the spy has only 3 states to move between. These states were chosen by breaking down the actions a spy would need to take when attempting to steal a file. Assuming the spy knows where the file is stored their first priority would be to head in the file’s direction, only changing course if they are spotted by a guard in which case the spy would attempt to find somewhere to hide and not leave until safe to do so. This formed the base for the three states of the spy which I could then alter as needed. With this limited number of states, a state machine style of structure was preferred as is a simple and cheap way to move between the spy’s few behaviours, while also giving the ability to easily move between any states.

The spy in program has 3 main behaviours to cycle between, the first of which is fleeing the guards. If the spy is spotted by a guard it will scan the objects in the scene and find the nearest with the tag that means it can be hidden in. If the spy makes it to this spot it will remain there and enter its second behaviour which is hiding. While the spy is in this state it will compare this distance from it to each guard in the level (Which it has stored as an array of GameObject’s with the guard tag). Then when no guards are within a certain range before leaving the crate and resuming its movement towards the objective. This implementation prevents the spy from leaving and being spotted by another guard straight away, however it may wander into another guard’s line of sight and be forced back into the hiding spot. If neither of these criteria are met the spy will move towards its known objective, which spawns one of several random places each time the program runs.

References

AI and Games (2019)*Behaviour Trees: The Cornerstone of Modern Game AI | AI 101* [Online Video] Available at: <https://www.youtube.com/watch?v=6VBCXvfNlCM> [Accessed 29/03/2020]

AltDevBlog (2011) *Introduction to Behaviour Trees* [Web blog] Available at: <http://jahej.com/alt/2011_02_24_introduction-to-behavior-trees.html> [Accessed 30/03/2020]

EpicGames (Unknown) *Behaviour Tree Overview* [Website] Available at: <https://docs.unrealengine.com/en-US/Engine/ArtificialIntelligence/BehaviorTrees/BehaviorTreesOverview/index.html> [Accessed 30/03/2020]