Task 3 – Design and Implement a State Machine for an NPC

The environment being simulated here is very basic. There will be a player-controlled agent which moves about the node map by clicking of the left or right mouse button. The left mouse button uses the Dijkstra’s pathfinding algorithm to find a node and the right mouse button uses the A-Star pathfinding algorithm. I set this feature up to display both algorithms working for the first and second task of this assessment. I will include multiple NPC’s that will wander randomly around the map and either follow or flee from the player. This could represent a real-world scenario of patrolling enemies that will chase you down when you get close enough to them. Regarding the flee behaviour this could represent the player trying to catch an NPC. When the NPC realises the player is closing in on them, they run away to a safe distance and keep moving round the map.

While setting up this finite state machine I had to create a “Condition” class which is used for creating condition functions that need to be met so the NPC will transition to another state. For all my behaviours I only require a “DistanceCondition” function. Next, I need to be able to create States and add behaviours to the State variables. This is handled in the “State” class, the behaviours are stored in a vector array. This class also stores the transitions and stores them in the vector array for transitions. The transitions contain the condition to be met and the state to transition to. Finally, I need to create the “FiniteStateMachine” class. This class takes in the different states that the agent will use. It contains the condition and transition data and stores it in a vector array. It then updates by checking if the transition conditions are true and transitioning to the new state.

Wander behaviour – The wander behaviour enables the NPCs to roam around the map moving from one starter node to a random end node at a set speed and colour which I have dictated in the behaviour functionality. This behaviour is constantly checking if the agent has completed its current path. This is a function that checks if the agent’s current path is empty and returns a true or false value. If this function returns true the agent calls its “GoToRandom” function, this creates and stores a random node from the node map in a Node\* variable. It then tells the agent to go to this node. This algorithm was the easiest to implement. The functions to check if the path was complete and send the agent onto a random node were easy to implement with the existing code I had already implemented.

Follow behaviour – The follow behaviour enables the NPC to track down the player-controlled agent. When it gets within a certain distance of the player agent. I set the distance condition in the main so that if the distance is less than 5, the NPC changes to the follow state. It does this “DistanceCondition” function by checking the distance and returning a bool if the condition is met. The NPC agent will then change colour and increase speed and lock in on the player’s position. This function also checks if the player moves again (while in this state) and adjusts it target node to where the player is going next. Once the player exceeds the max distance the NPC returns to the wander state. This algorithm was medium/hard to implement. They had a decent code example in the tutorial and the other functions I had to make to facilitate this were manageable. Took a bit of tweaking as I didn’t realise the distance was being calculated in a completely different way from the example. Wasn’t too difficult once I realised that.

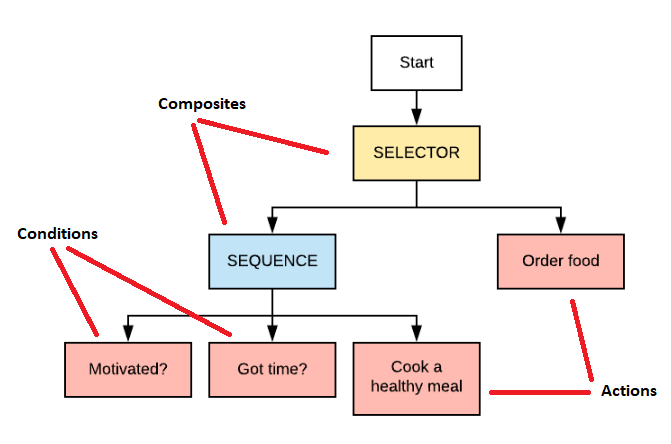
Flee behaviour – The flee behaviour enables the NPC to quickly run away from the player-controlled agent. When the NPC agent gets within the set distance of the player agent I call the same “DistanceCondition” function to change the state of the NPC to fleeing. I change the colour and increase the speed of the NPC agent so you can clearly tell it has changed states. The agent checks the distance of the player and then calls a “FleeToRandom” function I created which is basically just the “GoToRandom” function with an additional check to make sure the random node it goes to is over 5 units away. If it isn’t then then it chooses another random node to go to. This algorithm was the most difficult to implement as I didn’t have anything much in the way of an example to work from. Ideally, I would have liked to make the Flee behaviour operate so that the agent always runs away in the opposite (or most logical) direction from the player. I couldn’t think of a good solution to implement this and didn’t want to waste too much time trying so I went with the random node solution.

Task 4 – Design and Implement a Decision Tree for an NPC

Task 5 – Write an A.I. Game Strategy Report

Behaviour trees are a popular AI technique used in many games to model the behaviour of NPC characters. They are also used in computer science, control systems and robotics. Halo 2 was one of the first mainstream games to use behaviour trees and they started to become more popular after a detailed description of how they were used in Halo 2 was released. Ultimately, it is a tree of predefined node types aimed to represent how “something” behaves. Each node returns either Success, Failure or Running. Using just **Sequence** and **Selector**composites we can arrange custom **Actions** and **Conditions** to create almost any sort of A.I. decision system for our game's NPCs and even for our players. Player characters for games can be setup just like an A.I. controlled NPC, making use of a Behaviour Tree, but the decisions are derived from the player's input, e.g. Was X pressed? And are we jumping? Do air attack. Their main advantages are that they respond to interruptions, behaviours are re-usable, easy to understand and can be created using a visual editor like the behaviour designer tool in Unity.

The example below shows a behaviour tree for someone who is hungry and wants food. Behaviour trees work from top to bottom then left to right. So, it works its way down the tree to determine if the NPC wants to cook food or order food. To “cook a healthy meal” the conditions of “motivated” and “got time” must be met otherwise the sequence will fail and the NPC will have to order food.



* **Composites** are nodes that contain one or more children and dictate how they are run and when to stop. Two common types of composites are a selector and sequence. A **Selector** is a node that returns success if one of its child nodes returns success without executing its remaining child behaviours. If a child returns failure, then it executes the next child behaviour. If all child behaviours return failure, then the selector returns failure. Acts as an ‘OR’. A **Sequence** is a node that returns success if all its child nodes return success. If a child returned failure, then it would return failure and not execute the remaining child behaviours. All child behaviours must be a success for the sequence to return success. Acts as an ‘AND’.
* **Actions,** which are Leaf nodes in the tree. An Action behaviour is a behaviour that “does” something (e.g. move forward or perform this animation), generally always return success.
* **Conditions,** which are also Leaf nodes in the tree. A Condition Behaviour is a behaviour that “asks” something (e.g. is health empty or can see enemy), returns success or failure.
* **Decorator** nodes can only have a single child and are mostly used as utility nodes for more complex behaviour trees. They contain **Repeaters** (runs child node a number of times or indefinitely), **Inverters** (invert the result of the child node), **AlwaysSucceed** (failure becomes success) and **UntilFail** (runs the child node until it fails).

https://opsive.com/support/documentation/behavior-designer/what-is-a-behavior-tree/

https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work

<https://blog.zhaytam.com/2020/01/07/behavior-trees-introduction/>

https://en.wikipedia.org/wiki/Behavior\_tree\_(artificial\_intelligence,\_robotics\_and\_control)