Task 3 and 4 – Design and Implement a State Machine/Utility AI for an NPC

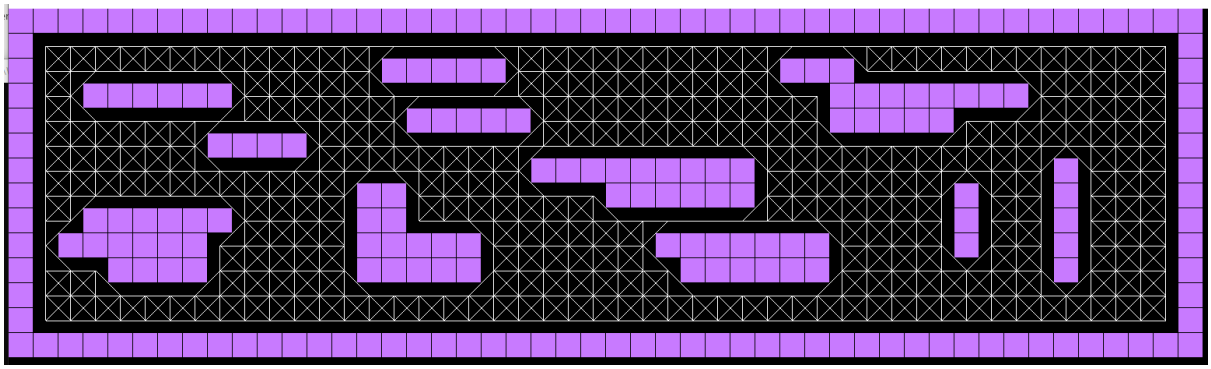
1 - A description of the real-world environment simulated, including any technical parameters

The project I am creating is a very basic simulation to demonstrate simple NPC behaviours I have learnt in class. The simulation will contain a map (design shown below) for the Player and NPCs to move around in. The player will be controlled by the mouse. The left mouse button will use the Dijkstra's pathfinding algorithm and the right mouse button uses the A-Star pathfinding algorithm. This feature will display both algorithms working as required for the first and second task of this assessment.

I will include multiple NPC's that will patrol around the map and either follow or flee from the player when they get close enough to them. The player can click around the map either running from NPC's or trying to catch them. If I have enough time, I will try implement some purpose to the task, so it becomes more of a game than a simulated demonstration. In the interest of making things easier for myself to complete all assignment tasks I will set up some NPCs to use the FSM and the others will use the Utility AI Strategy to carry out behaviours.

AI ELEMENTS – Mouse control, Patrol, Chase, Flee, A* Pathfinding, Dijkstra's Pathfinding, Finite State Machine and Utility AI

TECHNICAL INFORMATION – Coded in Visual Studio 2022, using C++ language, Top-down map



2 - A description of the underlying functionality of the A.I strategy specifying essential settings, states, conditions, and parameters

FINITE STATE MACHINE – To set up this finite state machine I need to create a "Condition" class which will be used for creating condition functions that need to be met so the NPC will transition to another state. All the behaviours will be dependent on a simple distance check condition.

Next, I need to create a "State" class which will allow me to add behaviours to the State variables, the behaviours need to be stored in a vector array. This class will also need to contain transitions and be able to store them in a vector array. The transitions will contain the condition to be met and the state to transition to.

Finally, I will need a "FiniteStateMachine" class. This class will take in the different states that an individual agent will use. It will need to contain all the data for the different states created and store it in a vector array. Then update by checking if the transition conditions are true and transition to a new state via enter and exit functions.

UTILITY AI – The utility AI will carry out the same behaviours I have already set up for the FSM, but do it based on evaluating which behaviour is required at the appropriate point in time. To do this I will need to add an "Evaluate" function to all of the behaviour classes and create an individual "UtilityAI" class. Inside this class I will need to be able to add behaviours and store them in a vector array. An update function will also be required to carry out the evaluation and switch behaviours at the appropriate time.

Each individual behaviour will need to return a float value from their evaluate function. This will be based on distance from player to the NPC. It iterates over each one and finds the one with the highest evaluation and sets that as the agent's current behaviour. Proceeding to change it through enter and exit functions like the FSM. The agent will only be able to switch between the specific behaviours added to the Utility AI variable.

3 - How the agents interact with the simulated environment

WANDER BEHAVIOUR – The wander behaviour will enable the NPC to roam around the map, moving from a starter node to a random end node. I will set the speed and colour in the behaviour functionality, so they are easily identifiable. This behaviour is constantly checking if the agent has completed its current path. If this function returns true, the agent will proceed to pick another random node on the map and head there.

FOLLOW BEHAVIOUR – The follow behaviour will enable the NPC to track down the player-controlled agent once it gets within a certain distance of the player agent. The NPC agent will change colour and increase speed to lock in on the player's position. The NPC will continue to follow the player until the player exceeds the max distance, then the NPC returns to the wander state.

FLEE BEHAVIOUR – The flee behaviour will enable the NPC to quickly run away from the player-controlled agent once it gets within the set distance of the player agent. I change the colour and increase the speed of the NPC agent so you can clearly tell it has changed states. The easiest way for me to do this would be to speed the agent up and send it to a random node with an additional check to make sure the random node it goes to is over 5 units away from the player. If it isn't then then it chooses another random node to go to. As soon as the NPC is far enough away from the player it will return to the wander state.

4 - Difficulty levels and their controls, as appropriate

The only difficulty levels and controls that will be adjustable within this project are the speeds of the player/NPCs, and the distance at which the NPCs starts to chase/flee from the player. These values could be easily scaled down to make it easier to catch or avoid the NPCs. They could also be scaled up to allow the NPC to recognise the player from further away and chase/flee faster. Although very basic this could be a starting foundation for different difficulty levels in the simulation.

5 - Evidence of discussion or review of the design against project requirements

I designed this project to meet the absolute minimum requirements of the assignment tasks. To make sure **tasks 1 and 2** are completed I gave my player agent the ability to use both Dijkstra's and A-Star algorithms. You can't necessarily tell it's doing anything different when playing apart from a colour change. But if you look at the code then you can see the left mouse button is using one algorithm and the right mouse button is using the other to move the player about the map. I wasn't sure how to implement a more graphic way to show the difference on screen. Might have been able to find a good solution if I had more time but wanted to prioritise finishing everything on time before doing any extra.

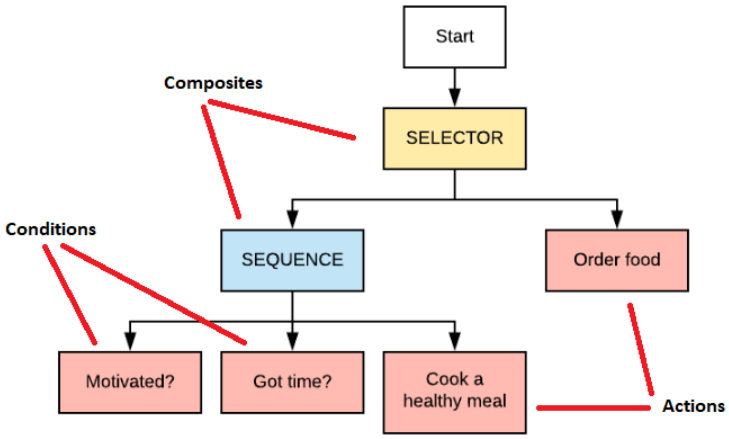
To make things easier for myself in **tasks 3 and 4** I implemented two different A.I strategies that use the same behaviours. I set them both up in the same project and made sure the NPCs were different sizes so you can clearly see both A.I strategies functioning correctly. If I had more time, I would have liked to make more behaviours that had some purpose e.g., being able to catch or kill the fleeing NPC's and punishing the player if the follow NPCs caught up to you. This would have made it more game-like and fun to play. Another thought was to implement completely different games for each A.I. strategy, this would have been even more time consuming though. To save myself from repetition I decided that instead of having two different design documents for these tasks, it would be easier to make one big design document (at least 2 pages for the whole thing). Both A.I strategies are doing the same thing just in a different way, so it seemed pointless to re-write a lot of this information.

For **task 5** I wanted to learn about an A.I. strategy I hadn't used, and behaviour trees seem to be popular in lots of big games, so I chose that one. I hit the 500-word mark and I believe it shows the basic design and use of the algorithm. There is plenty more detail I could have touched on for this strategy, but I didn't want to spend any more time on it while other parts of my assessment weren't completed. I would like to implement this strategy into one of my own projects in the future and play around with the Behaviour designer tool in unity so I can better understand and see it working.

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).