|  |
| --- |
| **MAT301 / CMP304 Coursework**  **Project Report (50%)**  **Finite State Machines and Behaviour Trees - Report** |
|  |
| **By: Rowan Ruthven - 1802152** |

|  |
| --- |
| **1. Introduction (5%)** |
| This AI project simulates a guard, similar to those seen in stealth games, patrolling the streets and responding to alerts. The simulation is built in Unity, using C# scripts, to compare the Finite State Machine and behaviour tree AI structures.   * REASONS FOR USING AI TYPES HERE   In the scenario, both the red guard and the blue guard have the same goal: to find the yellow targets in the scene and destroy them. Their standard patrol takes them around the scene in a repeating loop until either they detect a target or an alert is created in the scene (created by the user with mouse button inputs).  My theory as to how these AI will perform in relation to each other is that, for this scenario, the Finite State Machine will perform more efficiently than the behaviour tree due to the simple nature of the project. |
| **2. Methodology (15%)** |
| Finite State Machines    Behaviour Trees  Behaviour trees are widely used in games to create more realistic behaviour patterns for non-player characters, usually enemies. Well known examples of games using behaviour trees for their AI design include Halo, Alien: Isolation, and Spore.  Halo 2 Covenant Enemy Behaviour Tree  halo ai    A behaviour tree is a hierarchical node system that controls the decision making of an AI. Every branch in a behaviour tree is some form of a utility node that dictates the path an AI takes towards deciding on an action, controlling the flow of behaviour. The most common control nodes are the sequence and selector nodes, which both execute their child nodes in order from first to last (left to right in diagram).  Sequences act as an AND gate, requiring *all* of its children to return a success to return a success itself and stopping the sequence whenever a child node returns a failure. Selectors, in contrast, act as an OR gate and will return a success and break whenever its first child returns a success.  Parallel nodes are less commonly used control nodes that differ slightly from sequence and selector nodes. A parallel node executes all of its children at the same time. It will then return a success if some are all of its children return successes, similar to selector nodes.  Decorator Nodes are a sub-type of control node that exclusively have one child node, usually a leaf node. A decorator can come in one of many forms, but they primarily exist to modify the returned status of its child before returning it to the parent. The simplest type of decorator node is the inverter which acts as a NOT gate, returning the opposite of whichever status it returns. A succeeder decorator will ignore the returned status of its child node and always return a success. This can also be reversed to get a failer decorator which will always return a failure regardless of its child. A repeater decorator reprocesses its child when the child returns a status and can also be made to run its child a set number of times before moving on to the next node. A “repeat until fail” decorator is an expansion on the repeater that runs its child until the child returns a failure, which is when a success is returned to the parent node.  At the very end of every branch is a leaf node, also known as an execution node, which is incapable of having children but is the most powerful of the behaviour tree’s node types. While the control nodes dictate the flow of operations for the AI, the leaf nodes are the operations that execute a specific action. Leaf nodes are wholly defined by the person writing the AI program, making them the most diverse nodes in a behaviour tress.  More advanced behaviour trees can also include leaf nodes that call another behaviour tree and pass all of the current tree’s existing data to the newly called tree. These can be vital in larger games that involve a lot of similar AI that might mimic somewhat similar behaviours, since this lets some behaviour trees be reused, which is significantly more resource efficient than building many very similar behaviour trees for very similar function. |
|  |
| **3. Results (10%)** |
|  |
| However, I believe the Behaviour Tree would perform better if the project became more complex as Finite State Machines are far simpler, making them more limited |
| **4. Conclusion (10%)** |
|  |
| **5. References (5%)** |
| <https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867>  <https://towardsdatascience.com/hierarchical-finite-state-machine-for-ai-acting-engine-9b24efc66f2>  <https://towardsdatascience.com/designing-ai-agents-behaviors-with-behavior-trees-b28aa1c3cf8a>  <https://www.techopedia.com/how-is-a-finite-state-machine-used-in-artificial-intelligence/7/33998>  <https://studyofai.com/game-ai-theory/halo-ai/>  <https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work> |

Structure, style, formatting, spelling, grammar, coherence (5%)