|  |
| --- |
| **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 the Unity engine, using C# scripts, to compare the Finite State Machine and behaviour tree AI structures.  Finite state machines and behaviour trees are both widely used in the games industry, especially in a scenario similar to what I have designed. As these systems are so widely adopted in the games industry, it is paramount to know how they work.  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  A finite state machine is a computational model that is defined by a list of set states, usually an enumerator, that an AI picks to define its behaviour. Finite state machines are one of the simplest forms of AI that see use in the games industry, most notably in RPGs for non-playable characters, due to being inherently simple, predictable, and very easy to build.  A finite state machine effectively has a “start” state (s0), usually a lack of a defined state, before the system is initialised and the first state is called. There are a few ways to organize the function of the states in a finite state machine. The simplest method, and the method that is used in this project, to implement a finite state machine is to use a “switch” statement which acts according to which state is currently active. The stack data structure can also be used alongside the state enumerator to keep track of the order that each state is called in to allow the finite state machine to change states based on which states were called before it.  A more advanced method of building a finite state machine involves making classes for each state. This is done by initially creating a base state class that each state can inherit from. This allows each state to have an execution (or run) function with a shared name between the states by overriding the base state class with its own unique execution function, allowing to efficiently build a script that controls the execution of each state.  An even more advanced form of finite state machine is a hierarchical finite state machine. This system contains smaller finite state machines with sub-states within the main finite state machine. This can allow a single action state call (e.g. a “Move” state) to be more nuanced in execution and provide more fluid movement (e.g. Walk, Run, Jump, Roll sub-states).  Behaviour Trees  Behaviour trees are widely used in games to create more realistic behaviour patterns for non-playable 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.  The project I built compares the efficiency of behaviour trees with finite state machines at a rudimentary level. A base level finite state machine is, though very rigid and basic, quite efficient at a lower level, whereas a behaviour tree can start off rather bulky, having many essential classes even in a base level program. These systems are compared by the time they take to come to a decision.  The finite state machine version of the AI uses a switch statement to change between states (Patrol, Alert, Search, Attack), which are held in an enumerator. The “Patrol” state tells the AI to move towards one of four waypoints that are at each corner of the area and can transition to the “Alert” and “Attack” states. The pathfinding used in the AI is A\* pathfinding which allows for two-dimensional pathfinding in the Unity engine. The “Alert” state tells the AI to move towards the location of the alert so that it can begin searching and can only transition to the “Search” state. The “Search” state is the simplest state as it just increases the enemy detection radius over five seconds from 10 units to 15 and can transition back to the “Patrol” state after the time expires and the “Attack” state if it detects a target. The “Attack” state tells the AI to move to the nearest target and destroy it.     * Root node is selector * Selector moves between patrol and attack * Alert sequence moves AI to alert position, returns success when reaches position * Search function is meant to stop the AI for 5 seconds while searching, but does not work after many iterations |
|  |
| **3. Results (10%)** |
| * **Behaviour tree makes more regular and more accurate decisions** * **Finite State Machine only changes decision with a state change** |
|  |
| **4. Conclusion (10%)** |
| * Overall, behaviour trees are more resource efficient * Finite State Machines can be valid substitute for more basic AI as they are simpler to implement and faster to create * The gap in efficiency between both systems would widen as an AI became more complex as Finite State Machines are more computationally expensive * FSM could be hierarchical * Behaviour Tree could be more complex, having more realistic search behaviour |
| **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://medium.com/dotcrossdot/hierarchical-finite-state-machine-c9e3f4ce0d9e>  <https://arongranberg.com/astar/>  <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%)