|  |
| --- |
| **MAT301 / CMP304 Coursework**  **Project Report (50%)**  **CMP304 Assessment 1 – Finite State Machine vs Behaviour Tree** |
|  |
| **by: Max Clow - 1904985** |

|  |
| --- |
| **1. Introduction (5%)** |
| For this CMP304 coursework, the AI techniques chosen were a Finite State Machine and Behaviour Tree.  These techniques were selected due to ease of understanding the theory behind them, as well as a solid understanding of the practical applications of them. They also fit very well for the game that they were implemented in.  The game is a simple “Dodge the obstacles” game built in Unity. The AI implementation is as a cube, what would typically be a player-controlled object, and its objective is to avoid the obstacles while moving at a constant speed and make it to the end of the course.  Finite State Machines (FSM for short) are well suited for this task as avoiding obstacles can easily be translated into different states. For example:   * Move to the left * Move to the right * Jump * Duck   Transitions are also straight forward, with only the nearest obstacle type needing to be calculated. Typically, in other applications, states and transitions may not be so set in stone and “fuzzy”, and so FSM suffers due to its rigidity and requiring “crisp” values. The total amount of states for this application is 6, which is a very manageable amount and well suited for FSM, as too many states increases complexity and makes the program difficult to maintain and manage.  Behaviour Trees are quite similar to FSM’s, which immediately makes it a good idea to use in this same context. The intended aim of the project is to decide which technique suits this game best  **Background**  Basic Finite State Machines are possibly the simplest AI technique to implement into a program. States will have to be defined at the start of the program, which can be in either enums, ints or strings. The core setup of them involves using switch statements, which will allow specific code to be executed depending on what state the program is currently in. To switch states, the variable being used as the argument in the switch statement’s value must be changed to reflect the state change (eg. if the state needs to be changed to Jump, currentState variable would be changed to match the state name, “Jump”). The code inside of each state will have implement a way to switch states, so it doesn’t become stuck in the one state, like running a function. For this reports example, this is running a “check the nearest obstacle” function.  As stated previously, Behaviour Trees are very similar to FSM’s. The main difference is that instead of the “Behaviour” of the AI being controlled by what state it is in, it is more akin to a set of tasks. As the name suggests, this AI technique is depicted as a tree. Like real trees, they have branches and leaves. These leaves are where the actual commands of the program are executed. To get to them, branches (aka composite nodes) are traversed, and these can take on multiple different types. All leaves and branches are described as “nodes”.  Nodes can either be running or return a success or failure. This applies to both branches and leaves. It is these returns which then determine the flow of the tree, based on what types of composite nodes are being used.  Some common composite nodes include Selectors, that will always return success if 1 of its child nodes is successful. Sequences will return failure if any of their child nodes fails. There are also Random Selectors/Sequences that will randomly pick what child node to run first, second etc. as normal selectors/sequences always go in order. Behaviour Trees are more complex than Finite State Machines, but also scale easier with project size and scope and can be a lot more powerful. |
| **2. Methodology (15%)** |
| **Finite State Machine**  The Finite State Machine developed has 6 core states. Move Left, Move Right, Move Middle, Jump, Crouch and Uncrouch. These are setup at the start of the main script as enums. A starting state is applied to a generic state variable, which will be used in the switch statement.  In the Update() function the switch statement is written. Cases for each state are implemented. Inside all of them, the movement is applied to the object based on what state it is in (In middle state, the AI object should move to middle). Then, the ObstacleCheck function is called, as this ensures the program won’t become stuck in a state and is constantly checking if any obstacles have become closer.  **ObstacleCheck**  This function goes through the list of all game objects/obstacles in the level and checks their distance. It is akin to a MIN algorithm, checking for the smallest distanced obstacle from the AI object. Once this obstacle is found, there are multiple If statements checking the tag of this closest obstacle. Once the appropriate If statement has been entered, the state of the program is changed to match the state needed to avoid the obstacle.  **Behaviour Tree**  A diagram was created to help visualize and solidify the structure of the tree to be implemented into the program. Below is a picture of this diagram: |
| This diagram is then translated into the program. All the following nodes are in their own scripts. First, the Root Node which contains 3 states, Running, Success and Failure. It then has a getter for the current state it is in and reference to an Evaluate function. This class is the best place to implement the closest obstacle check function to avoid repetition of code as half the child nodes will need this function. This is the base class that all other nodes will inherit from.  **Composite Nodes**  Selector and Sequence nodes are similar, both inheriting from root node, both containing a list to store all its child nodes and their own implementation of the Evaluate function. This evaluate function has a for loop to go through every child node, and inside of that a switch statement depending on what is returned from the child nodes after they are evaluated.  **Selector**  If a child returns running or success, the selector will return this state. If a child returns failure, nothing happens as failures are allowed in Selectors. If the end of the for loop is reached, we know that all children must have returned failure, otherwise the loop would’ve been exited.  **Sequence**  If a child returns running, a Boolean is set to true. If success is returned, nothing happens and keeps iterating through the children. Finally, if a failure is returned, the sequence returns failure and no more children are ran. After this for loop, a check is performed to see if any children are still running. If they aren’t, they must have all succeeded and so we return a success.  **Children/Leafs**  For Child nodes, there are two for each sequence. A “check” and a “move”. All inherit from Node and all checks make use of the ObstacleCheck function  **ObstacleCheck function**  A list with all game objects(obstacles) and reference to the AI object is passed to the Node class. It takes these two values and iterates over each game object, calculating its distance from the AI. If a new shortest distance is recorded, the closest object is set to that and the variable for checking if a distance is below a certain threshold is set to this new distance. After the loop, the closest object is returned  **Check Nodes**  After calling the ObstacleCheck function, there is an If statement to determine if the tag of that closest obstacle is the one the node is checking for (eg IsLeftNearest node checks if object tag is equal to something like “FreeLeft”). The distance is then checked to make sure we are an appropriate distance away before returning a NodeState success. If the tag does not equal that which the class requires, failure is returned. The program then goes on to the next sequence  **Move Nodes**  These nodes contain the meat of the code in respect to actual Unity functions. If these nodes are reached, the program knows what object is closest and this node then enacts this object dodge. It is as simple as obtaining current position, setting desired position for certain moves, checking the object isn’t already in that position, performing the move and then returning success. These should never fail.  **Implementing into program**  Once all nodes are setup, the BT must be initialized and constructed into the main script. In Unity, this is done in a Start() function. Instances are created of all child nodes first, passing through appropriate parameters. Sequences are next, passing through a list of Nodes containing the children for their appropriate sequences. Finally, the Selector is set up with a list of nodes containing all sequences.  **Update()**  In the Update() function of the main script, all is needed now is to call the Evaluate function of the Selector Node. Every frame, the BT will be evaluated and checked. |
| **3. Results (10%)** |
| When running both FSM and BT programs, the immediate results are very similar. Neither program “fails” to avoid obstacles, and each make it to the end in the same time, as the speeds are the same. This shows they are both capable of doing the task.  To find differences, a timer can be implemented into both AI techniques. The job of this timer is to calculate the decision-making time of each technique.  For BT, the timer should be started each time the root node (in this applications case, the Selector node) runs it’s Evaluate function. The timer will then be stopped once the function returns a node state success, as this means a decision has been made and executed. The reason this timer doesn’t stop at just a decision is for the Behaviour Tree to be “complete” and reset, the “action” part must have run successfully too.  For FSM, the timer is started at the beginning of each possible state. Once a state reaches its check obstacle function, has determined what obstacle is closest and has checked if immediate action needs to be taken (if distance to obstacle is getting close), then the timer stops, as this is when the decision has been made.  Lists were constructed in both applications to store values and then calculate the averages. This method produced very accurate results as hundreds of values were being stored and used. See code for implementation.  This was repeated 10 times for each level and technique. L1 is “Level 1” and L2 is “Level 2”      **Results – Discussion**  As seen from the results above, the FSM takes less than 0.0000001 milliseconds to make its decision (C#’s Time command couldn’t track a difference below this, hence the 0ms results). BT on the other hand has a significantly higher amount of time compared to FSM to reach its decision (although still small).  The reason for this is because the decision-making part of this Finite State Machine (transition event) is simply the check closest obstacle function. In contrast, the Behaviour Tree has the chance to run this function 6 times depending on what obstacle is nearest before completing its decision. For example, if the AI needs to uncrouch, it will have to go through checking all other possible behaviours first.  For a Behaviour Tree to “complete” its iteration and reset the tree, it not only depends on what the closest obstacle is, but also relies on the “movement” nodes to be successful too. This adds some extra time to the timer calculation. The decision cannot be considered a “success” just on the basis it has checked for the nearest obstacle correctly. To time this in a way similar to FSM where the timer stops after the check obstacle function is run would not be a proper way to compare the decision-making times.  The Behaviour Tree times rely heavily on the layout of obstacles on the course. If all obstacles on the level were with a gap on the left, the times could be very similar to FSM. This is because a left-hand gap is what the BT checks for first.  It can be seen from the results that increasing the number of obstacles and reducing distances between has very little bearing on the decision-making time. Level 2 has double the obstacles as Level 1 and yet decision making time is only slightly increased (+0.000136ms) for BT’s. Increasing the number of obstacles by a lot more may increase times, but it would not really show anything about the AI’s decision-making time and is just an inflation of time due to the number of obstacles being looked over in the function.  The first test result in BT L1 is an outlier. This could have been for several reasons, most likely another process on the computer that took resources away from the application. |
| **4. Conclusion (10%)** |
| There is a general concept of modularity vs reactivity when comparing FSM’s and BT’s. FSM’s are far better suited to a reactive environment, where it changes states based on what is going on around it (ideal for a dodge the obstacles game).  However, if the application was to be expanded upon and some more complex behaviors added in, like grabbing a score bubble before dodging the obstacle, Behaviour Tree’s would become better suited as they have far more scalability when adding to the application (modularity) and are easier to adapt to different requirements and change. Adding these complex behaviours to an FSM would result in a lot more mess and less structure, as well as being a lot harder to implement, possibly having to create Finite State Machines inside Finite State Machines, which is not ideal from both a coding standpoint and in terms of running the game.  Behaviour Trees are also great for handling recovery when encountering failures. In a large and complex application, the chance of this happening is far greater, so it is best to make use of an AI suited to deal with this.  The Finite State Machine implemented for this application could have been made using classes but doing so felt like adding unnecessary complication to the code. The Behaviour Tree Uncrouch sequence could also have been modified to be more efficient. Instead of creating a new sequence for uncrouching, another selector could have been added in place of the Crouch leaf node to check for safe distance between the obstacle or not which would aid in deciding whether it was time to crouch or uncrouch.  Based on the results and methodology of implementing these techniques, it can be concluded that, for this specific application, Finite State Machines are better suited than Behaviour Trees. This is due to the ease of implementation and faster decision-making times. When dealing with such rigid and concrete conditions and states like this application does, FSMs are far better suited than BT’s. |
| **5. References (5%)** |
| Castro, S. (2021) Introduction to behavior trees. Available at:  <https://robohub.org/introduction-to-behavior-trees/> (accessed last 25th March 2022)  Colledanchise, M., Ogren, P., (2020) Behavior Trees in Robotics and AI:  <https://arxiv.org/pdf/1709.00084.pdf>  Simpson, C. (2014) Behavior trees for AI: How they work. Available at:  <https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work> (accessed last 25th March 2022)  Opsive (no date) Behavior Designer. Available at:  <https://opsive.com/support/documentation/behavior-designer/behavior-trees-or-finite-state-machines/> (accessed last 25th March 2022)  University of Rhode Island (2010) Lecture notes:  <https://homepage.cs.uri.edu/faculty/hamel/courses/2010/spring2010/csc481/lecture-notes/ln011a.pdf> |

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