Interactive Agents Evaluation/Write Up

**Finite State Machine Implementation**

For the State Machine implementation, I went for a 3 State entity that makes used of class-based states/machine. A possible implementation could have been to use Switch statements that constantly checks, and switches based on the result of the checks, however this is a pretty naïve approach and can become incredibly complex/confusing to continue to implement and maintain. *(Buckland, 2004).* I initially started with by creating a base class node, defining the functions/variables that is required for a state to work. States can then be easily be made by inheriting this base class. Each state has a Entry, Run, Exit and Change State function – Entry to be called on the first change to the state, Run to execute every tick, Change State to run every tick after the Run function to check if any state change needs to be made and an Exit function to be called before the Entry function of the next state. All these functions help the runtime of a state including being able to handle variables as required.

There is also a State Machine class which does the overall handling of the State Machine, this initiate the set up of the states as well as controlling/calling the Run and Change State functions of the state every main loop tick.

**Steering Behaviours Implementation**

I implemented 2 types of steering behaviours for the “Skeleton Entity”: Wander and Pursue/Seek. Both we’re implemented taking inspiration and advice from Mat Buckland’s Book: “Programming Game AI by Example” *(Buckland, 2004)*. This book provides a very in-depth explanation of how steering behaviours should/could be implemented in a way that also makes it look natural. Both make use of driving forces that smoothly guide the entity to a certain point.

The Wander behaviour works by making use of a circle being projected in front of the entity and steer it towards a target constrained on the perimeter of the circle. *(Buckland, 2004)*. Every update will then change the random point on the perimeter and guide the entity smoothly and randomly around the environment. This approach is slightly different to the one proposed in the book – Buckland’s approach takes the last generated point on the circumference and generates a new point within a certain displacement range of the previous point. This is a much more appropriate way of doing it and creates more smoother movement however my approach still works and just needed adjustment on variables of the circle (distance from entity and size) to create the same effect. I did intend on doing the way proposed above however I couldn’t quite get it working in time.

The Pursue/Seek behaviour kicks in once the Skeleton is close enough to the Pirate entity. The approach I took for this behaviour is a slight Hybrid of the separate Pursue and Seeking steering behaviours. I again took a proposed solution from Mat Buckland’s book and initially started with the pursuing part by checking the pirate entity and the velocity it is heading and by defining a value to look ahead in time to see where it would be after a set time. This creates the illusion of intelligence within the AI by predicting where the evader is going to be instead of seeking directly at their position. *(Buckland, 2004)*. The more direct seeking-like behaviour then activates when again the skeleton is even more close enough to the pirate and heads directly into it to “Attack it” and remove a treasure.

**Pathfinding Implementation**

I have implemented 2 methods of Pathfinding for this demonstration, A\* and Breadth First. With most inspiration/research taken from Red Blob Games: Introduction for A\* blog post. *(Redblobgames.com, 2019)*. This website goes through and explains Pathfinding really well from basic implementation to full explanations of advanced searching.

Breadth First search was developed first as this is a simple implementation to do and I could test and demonstrate whether my entire grid system worked. Breadth first searches equally in all directions and so the approach to take is to take a node, loop through all its neighbours and add them to a list of nodes that are to be checked on the next time round and rinse and repeat for the nodes to check. Because of this, the search gradually radiates outwards until the target is found.

The A\* algorithm was developed after and is a modification to a different Pathfinding called Dijkstra’s Algorithm. Both make use of prioritising the routes it searches through the use of a cost to travel for each node. Higher costs can be assigned to discourage certain routes for reasons such as danger/inefficiency - they prioritize paths that seem to be leading it closer to the goal. *(Redblobgames.com, 2019).* Starting from the start node, its neighbours are checked to see if the cost to traverse in that direction is cost effective – if so then it’s priority to be checked is calculated using the cost and the heuristic of the current node and the target node. Dijkstra’s and A\* uses the heuristic to determine how close currently the search is to the close and expand towards the goal more than other directions. The priority queue storing the currently searched nodes with their weighting is re-inputted into the function and the search continues until the goal is reached.

For both searches, when the target is reached the path is highlighted and determined by looking back through the parent node of where the search came from – working from the target node back to the starting position. The path is constructed and the Pirate entity will follow this.

# **Bibliography / References**

Buckland, M. (2004). *Programming Game AI by Example*. Plano, Tex.: Wordware, pp. 48-68, 90-99.

Redblobgames.com. (2019). *Red Blob Games: Introduction to A\**. [online] Available at: https://www.redblobgames.com/pathfinding/a-star/introduction.html [Accessed 23 Apr. 2019].