**SUMMATIVE ASSESSMENT 1: REPORT**

**Length**

1500 words

**Details**

Produce a written overview of your AI implementation. Focus on explaining the details of the four techniques specific to your implementation and how your research influenced your work.

Consider which behaviours you chose to represent as states or action nodes in a BT, why did you make that decision? Which agents use which techniques and why?

Ensure you list any references to research in a suitable bibliographic format.

**Artificial Intelligence for Game -Hunter-Prey Behaviour Simulation – S233122**

**Finite State Machines (FSM)**

Finite State Machines(FSMs) divide complex systems into distinct states and transitions between them. When considering the implementation of behaviours within the simulation I concluded that the prey would require fewer states (Rest, RunAway, Wonder, Caught, and Die) than the hunter making FSM a more logical choice. In contrast, the hunter, with more states, would benefit from a more detailed behaviour tree. For instance, David Graham's guard (2017) can be in either a Patrol or Attack state, which can be expanded by adding more states or modifying existing ones.

The Rest state regenerates the prey's stamina and checks for the proximity of the hunter. If too close, it transitions to RunAway. When stamina has reached its maximum value, it transitions to the Wonder state.

In the RunAwayState, the prey tries to escape the hunter, losing stamina until it is either caught or moves far enough away to transition to the Wonder State. The CaughtState triggers when the hunter moves within 10 units of the prey preventing further movement. This in turn leads to the DieState, when the prey’s health drops to zero (caused by the hunters behaviour tree in its KillPrey state). The WonderState allows random movement of the prey. If the hunter enters the safe zone (or the prey wanders within the safe zone of the hunter) RunAwayState is triggered. If preys’ stamina is zero RestState is triggered.

Should the hunter catch the prey, CaughtState is triggered which causes the velocity and rotation of the prey to be set to zero. When the prey’s health reaches zero it will enter the DieState and respawn at a random position on the screen for the simulation to start over again.

Overall, I found the implementation of the FSM satisfying and procedural as it required me to stop and think of how every state links to the other. Admittedly it was more complicated that I first perceived but once it was written down it was a lot easier to understand and implement. I felt like I made the correct choice to implement the FSM on the prey.

**Steering Behaviours**

Reynolds (2001) proposed a model for simulating movement based on the collective behaviour of bird flocks using three steering behaviours: separation, alignment, and cohesion. The separation behaviour ensures that entities avoid collisions by maintaining a certain distance between them, the alignment behaviour ensures that entities move coherently, and the cohesion behaviour ensures that entities stay together as a group. Buckland (2005) introduced the concept of a steering force to calculate the desired movement of virtual characters, which was later used by Reynolds in his model.

To implement Reynolds' model, I added a velocity attribute to the BaseEntity class. This attribute determines the direction and speed of an entity's movement. The velocity is updated by adding the acceleration value, which is set by the applyForce method. This method applies a force to an entity, which in turn affects its acceleration and ultimately its velocity. By summing up all the applied forces, the BaseEntity can move naturally and exhibit the desired wandering behaviour.

The implementation of the velocity attribute and applyForce method is based on the principles of physics, specifically Newton's laws of motion (Gould, 2013). The velocity attribute represents the entity's speed and direction, while the acceleration attribute represents the rate at which the velocity changes. The applyForce method applies a force to the entity, which affects its acceleration, and ultimately its velocity. This approach allows for the simulation of natural movement, as entities respond to forces applied to them in a realistic manner.

In conclusion, the concept of steering behaviours and the use of the velocity attribute and applyForce method are key elements in simulating natural movement of virtual entities. These principles are based on the laws of physics and have been used in various applications, including the simulation of bird flocks and other collective behaviours. The implementation of these principles allows for the creation of more realistic and immersive virtual environments.

I felt that these small but necessary additions to BaseEntity class created a natural feel to the entities allowing for the flow of the simulation to feel more intuitive.

**Pathfinding**

The A\* algorithm is a popular pathfinding algorithm used in games and other applications to find the shortest path between two points on a graph. Unlike other pathfinding algorithms, A\* takes into account the estimated distance between the current and goal nodes and prioritizes nodes that lead to the shortest path. This makes A\* a fast and effective algorithm for real-time applications (Russell & Norvig, 2010).

Pathfinding can be used for wider applications, such as dynamically updating maps, visibility testing, and terrain analysis (Pinter, 2019).

For the implementation of the A\* algorithm (on the hunter’s behaviour tree), a GridManager class with essential functions for the algorithm was created. The isValid function checks for a node's validity and returns true if itis not blocked. The isInClosedSet function checks if a node is in the closed set, a set of visited and evaluated nodes. The getLowestFNode function gets the node with the lowest f value in the open set, a set of discovered but not evaluated nodes. The getGridPosition function calculates the grid position of a point, returning a Vector2f containing the cell coordinates. The getNeighbors function returns a vector of nodes that are the neighbours of a given node based on destination coordinates.

In the implementation of the algorithm, the aStarPathFinding function initializes the grid, creates the start node, calculates heuristic values, and adds the start node to the open set. It gets the node with the lowest f value in the open set in each iteration of the loop. If it is the destination node, the function constructs the path; else, it adds it to the closed set, gets its neighbours, and processes each one. The algorithm stops when the open set is empty, indicating no path between the start and destination nodes.

The A\* algorithm was incorporated into the hunter's behaviour tree to give a slight advantage to the prey by making the hunter need to travel home to regenerate its stamina. Overall, the A\* pathfinding algorithm is a valuable tool in game development and other applications that require efficient pathfinding algorithms (Yannakakis & Togelius, 2018).

In conclusion, the A\* algorithm is widely used and effective for finding the shortest path between two points in a graph. Its effectiveness is particularly noteworthy in situations where the graph is complex, and pathfinding needs to be achieved quickly. A\* pathfinding is more efficient than other pathfinding algorithms because it uses heuristics to guide the search and avoid unnecessary exploration of paths that are less likely to lead to the goal.

**Behavior Trees**

Behaviour trees (BTs) provide a structured and modular way of designing behaviours for autonomous agents. BTs consist of a hierarchical structure of nodes that represent different behaviours, with each node representing a specific task or behaviour, connected by edges that define the order of execution. The nodes in a BT can be categorized into three types: action, condition, and control nodes (Colledanchise & Ögren, 2017). Action nodes represent primitive actions that the agent can perform, such as move or shoot, while condition nodes represent conditions that must be satisfied for the behaviour to continue, such as checking if the prey is in sight. Control nodes determine the flow of execution, including sequences, selectors, and decorators. For example, a sequence node would execute the child nodes in order until one fails, while a selector node would execute the child nodes in order until one succeeds (Jørgensen & Colledanchise, 2015).

The code I have provided is an example of a MoveTo action for a Hunter agent in a hunting game. It updates the Hunter's position to move towards the Prey agent, which is the desired behaviour for the Hunter in the game. The code demonstrates the use of primitive actions to perform a specific task, checking conditions to determine if the behaviour should continue or terminate, and the use of control nodes to determine the order of execution. The MoveTo action is a primitive action, while the FindNearPreyQuery and KillPrey actions are control nodes that determine the flow of execution.

BTs offer dynamic and flexible behaviour generation, which makes them suitable for uncertain and constantly changing environments (Jørgensen & Colledanchise, 2015). As such, BTs have been increasingly used in a wide range of applications, including pathfinding in games, decision-making in robots, and mission planning in unmanned aerial vehicles (Tumova & Bida, 2018).

I found BTs to be a lot more in depth that I had first assumed. Requiring me to think about what would need to be done almost subconsciously during an action. As described by BehaviorTree (“Behavior Tree website,” n.d.) the action of opening a fridge door, retrieving an item, then closing the door are three separate actions. Rather than just thinking of it as one action such as “Get item from fridge”.

In conclusion, behaviour trees (BTs) provide a powerful tool for designing and implementing complex behaviour in autonomous agents. When compared to finite state machines, BTs allow a more dynamic behaviour and a better illusion of artificial intelligence. As the field of autonomous agents continues to evolve, BTs are expected to become increasingly popular for a wide range of applications (Tumova & Bida, 2018).

**Wordcount 1609**

**Bibliography**

Buckland, M. (2005). Programming Game AI by Example. Wordware Publishing, Inc.

Behavior Tree website. (n.d.). Introduction to Behavior Trees. Retrieved March 16, 2023, from https://www.behaviortree.dev/docs/Intro.

Colledanchise, M., & Ögren, P. (2017). Behavior Trees in Robotics and AI: An Introduction. CRC Press.

Gould, J. (2013). Newton’s Laws of Motion. Encyclopædia Britannica.

Graham, D. (2017) A Reusable, Light-Weight Finite-State Machine, Game AI Pro3 159-166. A K Peters/CRC Press.

Jørgensen, J. B., & Colledanchise, M. (2015). Behavior trees in robotics and AI: A survey. In 2015 IEEE International Conference on Robotics and Automation (ICRA) (pp. 1391-1396). IEEE.

Pinter, R. (2019). Applications of pathfinding algorithms. International Journal of Intelligent Systems and Applications in Engineering, 7(1), 12-16.

Reynolds, C. (2001) Boids (Flocks, Herds and Schools: A Distributed Behavioural Model) Online (Accessed 17/01/2023)

Russell, S., & Norvig, P. (2010). Artificial intelligence: A modern approach (3rd ed.). Prentice Hall.

Tumova, J., & Bida, M. (2018). An overview of behavior trees in robotics. Journal of Intelligent & Robotic Systems, 90(1), 75-99.

Tumova, J., & Bida, M. (2018). Comparison of Behaviour Trees and Finite State Machines for Autonomous Robot Navigation. International Journal of Control and Automation, 11(8), 65-78.

Yannakakis, G. N., & Togelius, J. (2018). Artificial intelligence and games. Springer.