Student Name: Thomas Jacobs.
Student ID: S212046.
Student Email: S212046@uos.ac.uk.

Games Programming

Artificial Intelligence for games

Introduction:
Detailing the definition, feature(s) and implementation of each required technique in the order they were taught/established.

Finite State Machines:
A finite state machine is a computational model that is used to organise, execute and simulate sequential logic in a deterministic or even non-deterministic configuration. For my project, I used what is called deterministic finite automata. As explained by (tutorialspoint, no date) “In DFA, for each input symbol, one can determine the state to which the machine will move.” In other words, this means that the state machines featured in my project are very predictable; providing you are aware of the logic executed within each state, you can determine and even map out its outcome.

The “hunter” class highlighted in my presentation at its core uses a finite state machine to determine its behaviour (how it will react to the world) for each frame during the runtime of the application. Garcia (2018) elucidates, “Finite State Machines are simply a mathematical computation of causes and events”, which works as a rationale for the “hunter” class because it has a variety of behaviours that I wanted to structure into a computational module where the processing and transition of these behaviours could be automated.

The 'hunter' agent was implemented into my presentation to preserve balance for the population of various other agents. This goal of perseverance is achieved by splitting up each possible action the hunter could take into a relevant state. The state machine is utilised similarly to how (AI and Games, 2019) summarised a state as behaving, “When modelling AI behaviour in a game, a state represents a specific behaviour”.

In more detail, the hunter will state with the root state set as what I called a "passive state" in this state it will perform a few calculations to decide what sub-state to switch to next. If the hunter sees an influx in the plant life then it will return to a state where it will tackle this problem; by deleting some plant agents. It will continue ticking in this state until the problem is resolved where it will return to the 'passive state' so the agent can look for any new issues.

Student Name: Thomas Jacobs.
Student ID: S212046.
Student Email: S212046@uos.ac.uk.

One final note you may be interested to be informed of was my decision to implement a very simple polling system into the state machine. (Tech Target, 2005) defines polling as "the continuous checking of other programs or devices". When cycling through states the algorithm will first check if anything state is currently polled and if there is a state present it will then override the current state and instead run the state at the start of the queue (first in first out system). Once the queue is empty the state machine will revert to the previous state before polling occurred. This is ideal for any short states that may need to occur between two larger states, for example, the hunter polls a state which changes its colour and speed before switching to an 'aggressive' state.

Steering Behaviours:
Steering behaviours categorise a range of computational calculations that can be used to provide propulsion to an agent in a way that can evoke a desired characteristical trait. Furthermore, (Reynolds, 1987) explains “Steering behaviours are described in terms of the geometric calculation of a vector representing the desired steering force.” which means all of the steerable behaviours can be broken down to fairly simple vector calculations (calculus to be exact).

In my presentation, you will be introduced to two types of agents that inherit from a “Seeker” and “Boid” class. The “Boid” class provides the framework for implementing a runtime ‘flock’ of birds. The “flocking” steering behaviour of a boid can be divided up into three distinctive sub-behaviours as described by Reynolds (1987 p.25) as being, “Collision Avoidance”, “Velocity Matching” and “Flock Centering”.

The first sub-behaviour described by Reynolds as “Collision Avoidance”, rather self-explanatory, steers the boid away from any potential obstacles. In my simulation most of these collidable ‘obstacles’ would be other boids, this was achieved by finding the difference in position between a boid and any nearby neighbouring boids.
The next two behaviours provide a trajectory that encourages the boids to form groups of flocks. “Velocity Matching” simply gathers an average velocity of all neighbouring boids and then applies its force in the form of a normalised vector. “Flock Centering” attracts the boids towards the theoretical centre of a group of neighbouring boids by simply gathering all individual boid positions within a flock and then dividing it by the number of boids within said flock to find the local centre. It is vital all three of these behaviours are used for the flocking simulation since a flock needs to follow in a grouped direction but without being as close as to collide into its flock mates.

The seeker behaviour as described by (Reynolds, 1987) is “to steer the character towards a specified position in global space.” To achieve this desired characteristic I effectively flipped the equation used for the boid sub-behaviour mentioned above as ‘Cohesion Avoidance’. To ‘flip’ the equation, the seeker finds the difference between its target’s position and its current position which outputs a vector force steering the seeker towards its designated target’s position. The

Student Name: Thomas Jacobs.
Student ID: S212046.
Student Email: S212046@uos.ac.uk.

algorithm used then normalizes the resulting force vector again since the unnormalized vector would be too large for it will be applied to the "Seeker" for every frame of the simulation.

Pathfinding:
In mathematics, pathfinding is used as a solution for a problem within graph theory that (Chopra, 2019) elegantly summarised as a "Problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edge is minimized". To elaborate, this means that we can use techniques within pathfinding to provide a path, which leads from a start point to an endpoint with the shortest weight possible.

Unlike steering behaviours which effectively use real-time instrumentation to avoid collidable game objects, pathfinding uses a graph of vertices, that can account for the context of the map allowing an agent to avoid static objects. The agent provided by the "Insect" class in my project does exactly that; it looks for static targets to feast upon. Conveniently, all of the "Plant" agents in my project are assigned a vertice from the grid on the call of initialization which they sit upon. This means the "Insect" class simply needs to find out where it currently is relative to the graph and then needs to plot a path from its current position to the plant's position.

For the "Insect" class I chose to use a version of Dijkstra's algorithm known as "A*". As (Red Blog Games, 2014) explains "A* is a modification of Dijkstra's Algorithm that is optimized for a single destination." In more detail the optimisation that A* has which "Red Blog Games" eludes to is mainly down to a core component in its formula were as (Red Blog Games, 2014) states "prioritizes paths that seem to be leading close to a goal". Unlike Dijkstra's algorithm, the Insect class uses what is called heuristic value (Red Blog Games, 2014) explains is "to reorder the nodes so that it's *more likely* that the goal node will be encountered sooner."

In my adaptation of the algorithm, I simply chose to use euclidean distance to calculate the heuristic weight from a node being considered to the ending node most because I already had the implementation for the distance formula in my two vector class. Using this heuristic cost as well as a "G" cost which considers the cost from the start node to the current node being considered A* frequently produces a short and efficient path because unlike other pathfinding algorithms it vitally considers the end node with every node that it considers adding to its path.

Behaviour Trees:
In computer science, a tree is an abstract data type that follows a hierarchical structure. In the context of a behaviour tree, this means we have a structure with a fixed operation where a root starting behaviour leads into a tree of child nodes, which can be further divided by nodes with a predetermined function such as a sequencer that allows the execution of multiple nodes in one cycle. Notice, because a behaviour tree is an abstract data type, the algorithms that go in within these child nodes are not predetermined and are up to the programmer to implement. (AI and Games, 2019) describes the structure as being an "acyclic graph", which in mathematics means it's a type of graph with no cycles, it simply follows the hierarchy going from the root to the child nodes.

Student Name: Thomas Jacobs.
Student ID: S212046.
Student Email: S212046@uos.ac.uk.

In my project, you will have hopefully noticed a static agent with a few interesting runtime characteristics, the “plant” class, was designed to display the use of some behavioural mechanics as well as being a potential target for other pathfinding agents to utilise. A behaviour tree was a fairly ideal data structure to use because it allowed me to very clearly divide each behaviour into individual nodes, with a clear hierarchical order of execution. It also unlike a state machine allowed me to execute a group of nodes together in one cycle, so rather than only reacting to one type of agent at a time, it can potentially have a reaction to multiple types of agents at the same time.

The “Plant” class is set up to have a selector behaviour at its root that decides whether to choose what I call a passive or defensive branch. The passive branch is a sequence that (*What is a behaviour tree?* no date) explains “runs each task once until all tasks have been run.”.

To gather the details needed when deciding whether or not to process a particular behaviour an abstract data structure called a “blackboard” can be used. (AI and Games, 2019) explains that “blackboards are used to store useful data that the tree uses as part of the behaviour of multiple nodes.” This means a plant agent can store and refer to data such as the location of the nearest boid and decide what is an appropriate behavioural response using the behaviour tree.

Conclusion:
In conclusion, looking back at the work I have completed for this module, I can confidently say I have successfully implemented all mandatory techniques listed in the assignment brief. Not only was I able to implement these features myself in a direction that was suitable for the context of my project but I am also left feeling very well informed with each technique covered.

Looking on towards the positive side of the project I am very happy with not only how I managed to embed each technique but also how each technique had been designed to build a highly visual demonstration where each agent interacts with each other. If I were to change one thing about the project if I had to restart the project with my present knowledge, it would be to more cautiously plan out what type of agent would utilise each technique and whether it was beneficial to the agent.

In more detail, most of the time it should be pretty obvious what technique you want to use for your agents, for example, if you want an agent to have pathfinding capabilities you obviously won’t use a state machine explicitly to elicit this trait. That being said for techniques such as behavioural trees and state machines the reasoning for using such techniques can be almost identical. Because of this, I made the time-costly, early mistake of not thoroughly considering the vital differences in both performance and capabilities associated with state machines and behavioural trees, which resulted in me switching around certain techniques later in the project than I would have liked because it gave me less time to more thoroughly plan out the agent’s behaviour.

Student Name: Thomas Jacobs.
Student ID: S212046.
Student Email: S212046@uos.ac.uk.

Student Name: Thomas Jacobs.
Student ID: S212046.
Student Email: S212046@uos.ac.uk.

Bibliography

AI and Games (2019) *Behaviour Trees: The Cornerstone of Modern Game AI | AI 101*, *YouTube*. Available at: https://www.youtube.com/watch?v=6VBCXvfNICM.

AI and Games (2019) *The AI of Half-Life: Finite State Machines | AI 101*, *YouTube*. Available at: https://www.youtube.com/watch?v=JyF0oyarz4U (Accessed: 30 November 2021).

Chopra, C. (2019) *Pathfinding Algorithms*, *Medium*. Available at: https://medium.com/swlh/pathfinding-algorithms-6c0d4febe8fd (Accessed: 25 November 2021).

Garcia, J. (2018) 'Advantages & Disadvantages of Finite State Machines: Switch-cases, C/C++ Pointers & Lookup Tables (Part II)'. Ubidots. Available at: https://ubidots.com/blog/advantages_and_disadvantages_of_finite_state_machines/ (Accessed: 20 October 2021).

GeeksforGeeks (2019) *Abstract Data Types*, *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/abstract-data-types/.

Lague, S. (2014) *A* Pathfinding (E01: algorithm explanation)*. Available at: https://www.youtube.com/watch?v=-L-WgKMFuhE (Accessed: 1 November 2021).

Lague, S. (2019) *Coding Adventure: Boids*. (Boids). Available at: https://www.youtube.com/watch?v=bqtqltqcQhw (Accessed: 25 October 2021).

Moore, K. and Gupta, D. (no date) 'Finite State Machines', *Brilliant* [Preprint]. Available at: https://brilliant.org/wiki/finite-state-machines/ (Accessed: 27 November 2021).

Opsive (no date) *What is a behaviour tree?*, *Opsive*. Available at: https://opsive.com/support/documentation/behavior-designer/what-is-a-behavior-tree/.

Pound, M. (2017) *A* (A Star) Search Algorithm - Computerphile*. Available at: https://www.youtube.com/watch?v=ySN5Wnu88nE (Accessed: 1 November 2021).

Student Name: Thomas Jacobs.
Student ID: S212046.
Student Email: S212046@uos.ac.uk.

Reynolds, C. (1987) ‘Flocks, herds and schools: A distributed behavioural model’, *ACM SIGGRAPH Computer Graphics*, 21. Available at: https://www.cs.toronto.edu/~dt/siggraph97-course/cwr87/ (Accessed: 25 October 2021).

Tech Target (2005) *What is polling?*, *Tech Target*. Available at: https://whatis.techtarget.com/definition/polling (Accessed: 7 December 2021).

Tutorialspoint (no date) ‘Deterministic Finite Automaton’. Available at: https://www.tutorialspoint.com/automata_theory/deterministic_finite_automaton.htm (Accessed: 27 November 2021).

***5.1 Autonomous Steering Agents Introduction - The Nature of Code* (2021). (The Nature of Code). Available at: https://www.youtube.com/watch?v=P_xJMH8VvAE (Accessed: 25 October 2021).**