Due
31/03/2023

IGB383 Assignment 1

FSM & Navigation Algorithms

Thomas Peters - N10810251

Concepts & Changes

Changes

Navigation

The navigation system provided has been altered to use LinkedNodes as their dependent data object rather than a node ID system. The reason for this was both debugging and optimisation as when debugging a node ID can fail to provide enough information to identify the problem.

Concepts

Astar Algorithm

The Astar algorithm is a form of advanced graph traversal algorithm which employs three main parts to determine valid nodes during operation. These key parts come in the form of two heaps, a heuristic and a measure of cost. During operation, the algorithm weighs the traversal cost of nodes not previously covered or previously covered but greater than the newly determined cost to find a short and valid path. The heuristic is commonly the distance from the node to the intended goal, which is added to the cost of traversal to that node to generate a value which is used to determine the next node to progress from. Prior to each operation this would require the heaps to be re-ordered to return the appropriate node.

During implementation this algorithm was modified slightly from the provided version to utilize a *Parent/Child* design scheme which reduces additional operations from the provided version in the event the goal cannot be reached. The pitfall of this method is it removes the ability for a node to be utilized by multiple agents simultaneously, a problem that cannot occur in Unity due to its single threaded design for script events.
Additionally, the open heap was implemented using a minimalistic version of the Fibonacci Heap, which only implements necessary functionality such as *Contains, Pop, Push, Clear*. The reason surrounding this is the Fibonacci Heaps design which inherently makes the Push operation Θ(1) and Pop Θ(LogN) similar to other forms of heaps, however, I had made an attempt to implement this heap in a prior project, so my understanding of its functionality was greater.
Refer to the included image for demonstrations of functional implementation.

Greedy Algorithms

Greedy's implementation is similar to that of Astar in terms of functionality, however, rather than a value representing the next best position using a cost and heuristic, it solely relies on the heuristic from a node to the goal. This means whilst it may select a valid path that finds the goal, it may take an additional few nodes due to only validating a node on its distance to the goal rather than distance between. In some cases Greedy can outperform Astar depending on the design of the level as Astar may get stuck in a corner or dead end due to its proximity and cost from the goal and start respectively being lower.

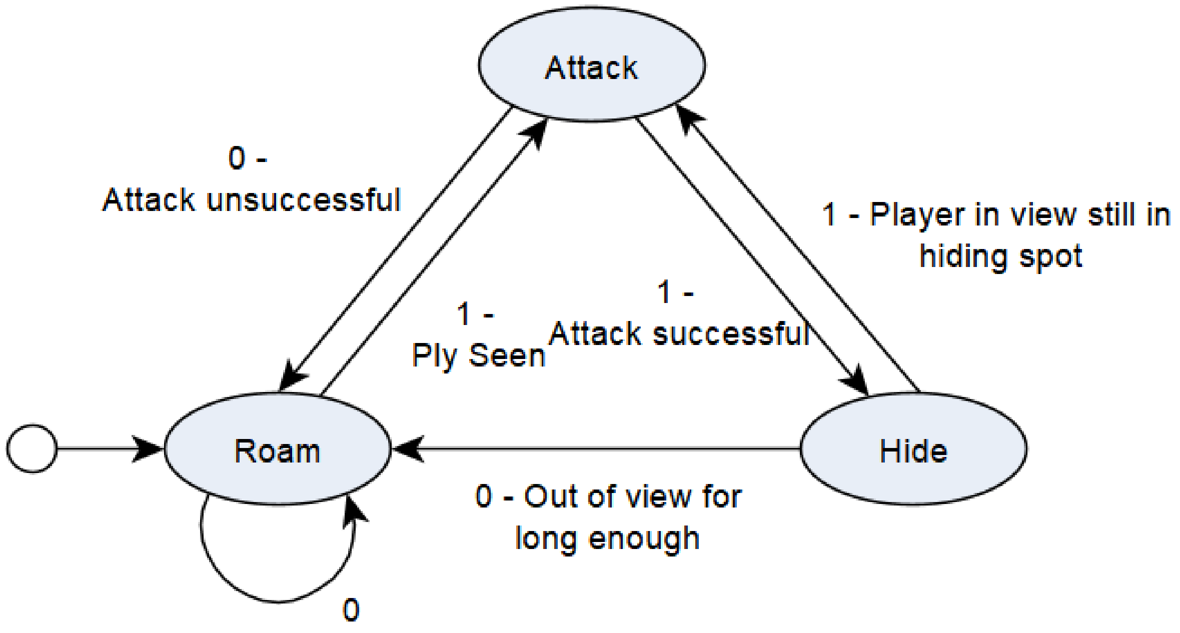
For this implementation, I referred to a previous assignment on graph theory in which I implemented a stack to represent the current path due to its design for LI-FO operations. This just replaces the use of an "open heap" but also removes additional code overhead with addressing items in the heap directly.

Finite State Machines (FSM)

Finite State Machines are the concept that a set of actions can relate and connect to another. For example, when opening a locked door you must insert the key. Upon inserting the key the state of the situation inherently changes. With the presence of the key in the lock, the next action would be to turn the key, unlocking the door, so on and so forth.
FSM's are a fixed set of user states & trigger states.

For implementation, a Finite state machine is included with each AI. This "Manager" contains a dictionary of variables used for functionality and transitions, as well as a dictionary of usable states. Each state is added to the FSM on *Setup()* through a *FSM_Setup()* method which also defines the starting state. A state in this case is a separate inheritable class containing an OnEntry, OnExit & OnUpdate virtual function and an internal reference to the *Manager* they are currently in allowing each state to change to another state depending on transition variables accessible through the *Manager* reference.

Enemy Default (DFA)

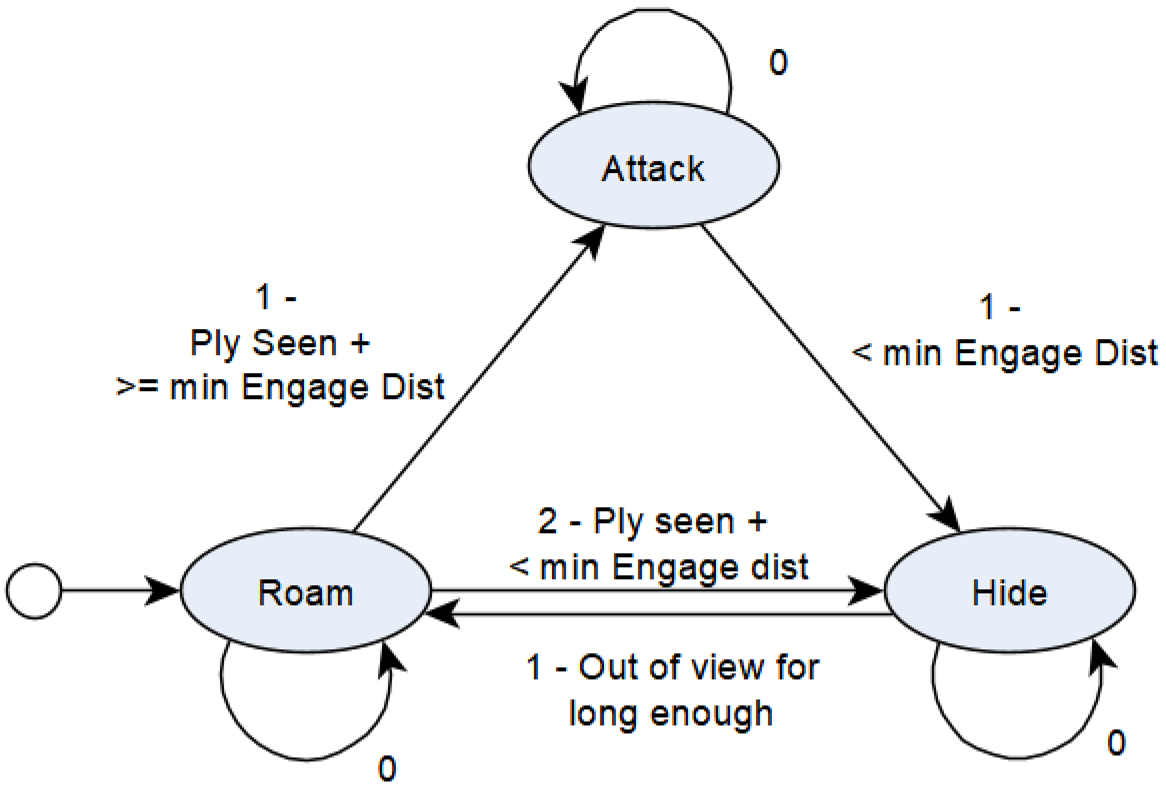


State	Transitions	‘0’	‘1’
Roam	true	Roam	Attack
Hide	true	Roam	Attack
Attack	true	Roam	Hide

State transition conditions

State	Transition	Type	Description
Roam	Attack	Bool	Line of sight with player
Attack	Roam	Bool + Time	A path to the player cannot be obtained or the enemy has chased for a duration but not reached the player
Attack	Hide	[redacted]	The player was reached & attacked
Hide	Roam	Time + LOS	The player hasn’t seen the enemy & a duration of successful hiding has passed

Enemy Gunner (NFA)



State	Transitions	‘0’	‘1’	‘2’
Roam	true	Roam	Attack	Hide
Hide	true	Hide	Roam	■
Attack	true	Attack	Hide	■

State	Transition	Type	Description
Roam	Attack	Distance + Bool	LOS + > min Engage distance
Roam	Hide		LOS + < min Engage distance
Attack	Hide	Distance	< min Engage distance
Hide	Roam	Time + LOS	The player hasn’t seen the enemy & a duration of successful hiding has passed

Environmental Reaction - Hiding

Extending from FSM’s I implemented a method for a degree of perception from the AI. Specifically, the ability to identify valid hiding spots behind environmental objects without using Greedy, Astar or graph iteration. The reason for this was the belief reactivity from an AI should function independently of navigation. Akin to how a player in a horror game is aware of nearby hiding spots, running and hiding from the user should be viable independently of where in the environment the AI is. For this the AI sends out a predefined number of raycasts radially. Each raycast is then extended and its direction is reversed using the collider of the originally hit object to provide a point on the opposing side of the object. This point is then converted into a viable point outside of the player's view.

Within the scene is a *HideTesting* object complete with gizmos and debug lines demonstrating the implementations results.