BeTheBase

Author: Bas de Koningh
Date: 26-03-2020

Aan het begin van deze opdracht ben ik gaan onderzoeken wat er allemaal mogelijk was in robocode en wat er volgens verschillende mensen juist wel of niet goed werkt. Toen ben ik gaan onderzoeken welke onderdelen er cruciaal zijn aan een behaviour tree.

Ik ben begonnen met een Abstract BaseNode class als basis voor de Nodes. Hierop heb ik een SelectorNode en een SequenceNode gemaakt. In de meeste gevallen heb ik een van deze twee "parent nodes" nodig om m'n gedrag mee te schrijven. De BaseNode class heeft twee onderdelen: een protected BlackBoard en een abstract BehaviourTreeStatus Tick() method. De BlackBoard wordt gebruikt om data mee door te geven en elke node moet deze erven. Elke node heeft ook een BehaviourTreeStatus Tick() method nodig om mee aan te geven in welke staat de node zich bevindt. Ik gebruik een enum om de verschillende staten bij te houden. BehaviourTreeStatus is deze enum, hij heeft drie staten: Succes; Running; Failure.

Hierna heb ik een ConditionNode gemaakt die ook een Func<bool> meekrijgt. Als je deze gebruikt kun je dus een conditie meesturen die True of False returned, dan pas gaat de node de volgende node uitvoeren. Na m'n basis voor m'n behaviour tree te hebben opgezet ben ik gaan proberen om het in robocode werkende te krijgen met verschillende "test" nodes.

Voor het drag heb ik gekozen om een paar verschillende werkende manieren van gedrag te gebruiken. Ik heb meerdere behaviour trees aangemaakt en ik wissel de "currentBehaviourTree" telkens af met een andere op het moment dat ik denk dat ik ander gedrag wil tegenover mijn tegenstander. Voor de verplaatsing van mijn robot heb ik gekozen om Anti Gravity movement te gebruiken. Dit is een flexibele techniek die kan helpen patroonanalyse-robots voor de gek te houden doordat ik bepaalde punten(GravPoints) op het veld kan definiëren. Elke gravity point krijgt zijn eigen kracht toegewezen. Door de componenten van deze sterkte op te lossen in de richting x en y, kun je een makkelijke poging doen om alle vijandige robots te ontwijken. Alle gravity points zijn punten op het veld waar de robot juist op af of van weg wilt.

Voor het schieten van mijn robot heb ik na veel iteraties gekozen om een soort enemy container te maken(EnemyData class). Waarin ik alle data die ik nodig dien te hebben van een vijandige robot opsla. Met deze data kan ik makkelijk door middel van Lineair of Circulair targeting inschatten waar de robot zich gaat bevinden na N tijd. Om te kiezen met welke kracht we gaan schieten heb ik ervoor gekozen om de afstand tot de tegenstander te berekenen waarna de robot kiest

met welke sterkte hij schiet(tegenstander dichtbij is sterkste kracht, tegenstander ver weg is laagste kracht). Ik heb er uiteindelijk na testen voor gekozen als de tegenstander te ver verwijderd is van de robot hij beter niet kan schieten omdat dit een lagere kans van raken heeft.

Ik heb best veel getest met test robots van robocode of gedownloade robots die erg goed zijn. Hier een tabel van een stukje van mijn testresultaten:

Enemy:	Crazy	Crazy	Crazy
Bullet Damage:	741	734	680
Times won:	10	10	9
Overall Percentage:	94	94%	91%

Enemy:	Wall	Wall	Wall
Bullet Damage:	806	777	682
Times won:	9	10	8
Overall Percentage:	87%	94%	81%

Enemy:	DrussGT(champion)	DrussGT(champion)	DrussGT(champion)
Bullet Damage:	79	0	0
Times won:	0	0	0
Overall Percentage:	5%	0	0

Voordat ik voor de Lineaire en Circulaire targeting gekozen had, heb ik eerst geprobeerd om met A* en Pattern Matching te werken. Helaas kreeg ik dit niet 100% ideaal werkend en was het resultaat matig.

Hieronder een stukje pseudo code hoe je een pattern matching algorithm kunt bouwen:

- Keep a log of enemy movements that stores the enemy's velocity and heading change each tick.
- When aiming, take the last 7 enemy movements and use this as your pattern (let's call it p, made up of p[0], p[1], ... , p[6]).

- Search through your history of enemy movements and find the series of 7 ticks that most closely matches `p`. For each position `x` in your search history:
    - Measure the difference between `x` and `p[0]`, `x + 1` and `p[1]`, up through `x + 6` and `p[6]`. This measurement could simply be `abs(velocity1 - velocity2) + abs(turnRate1 - turnRate2)` for each individual tick.
    - Sum these differences - the lower the result, the closer the match. Call this sum `s`.
    - Remember the lowest value of `s` and its position in the log.
- Once you have found the closest match, imagine that the enemy will move with the exact same series of movements that he did last time. Note the enemy's current position and heading, then iterate through the movements immediately after the match we found.
    - For each successive movement frame, move the enemy's position ahead by the past frame's velocity and then turn the enemy's heading by the past frame's heading.
    - Figure out how long it will take your bullet to get there and repeat the prediction that many times, then fire at that spot.

Sources:

http://robowiki.net/wiki/Play_It_Forward

http://robowiki.net/wiki/Pattern_Matching

http://robowiki.net/wiki/Linear_Targeting

https://www.ibm.com/developerworks/library/j-antigrav/index.html