|  |
| --- |
| [Company name] |
| AI for Games |
| [Document subtitle] |

|  |
| --- |
| Elliot Robinson (Q5130385), Gabriel Menezes (S6088315), Jonathan Archer (T7125507)  [Date] |

Contents

Introduction

Results and Evaluation

Decision Making

Flocking

Pathfinding

Conclusion

References

Introduction

This report focuses on the implementation of several AI algorithms into a game based on the multiplayer online battle arena (MOBA) genre. The algorithms include:

* Decision making
* Flocking
* Path finding

The game is influenced by popular games such as League of Legends and DOTA 2 and uses a map with a very similar layout consisting of bases, a jungle, a series of towers and 3 lanes.

We discussed several different game types including real time strategy and role-playing games but felt these styles of gameplay may be too time consuming. Ultimately, we settled on the MOBA due to our team being more familiar with this style of game.

The AI algorithms were to be used on different areas of the game to allow each one to be demonstrated individually. The decision making is used on minions and champions to control where they are going and what to target. Flocking is being used on jungle minions allowing them to roam freely and group up in the jungle and the path finding is used to replace the NavMesh for the minions and champions.

Flocking

Flocking algorithms use a set of behaviours that can be applied to a set of agents. These behaviours make the agents appear to move as a sensible and coordinated group. The group can exhibit patterns which make it look like there is some form of flocking manager deciding where the agents go, however, each agent is making its own decisions based on the position of its neighbours.

The flocking behaviours should apply a movement to an agent based on how many neighbours are within a given radius of it and the direction in which they are moving. If an agent has no neighbours, it should continue to move in a forward's direction. For each agent, we iterate over its neighbours and calculate its destination by combining the results from three core behaviours.

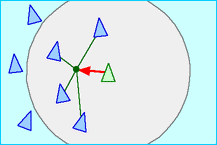


Figure . Cohesion (Reynolds, n.d.)

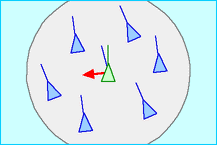
The cohesion behaviour *(Fig. 1)* tries to keep agents within the flock by finding the average position of the neighbours and moving towards it. This behaviour can sometimes cause the movement to look erratic and may need some form of steering or smoothing incorporated into it.

Figure . Alignment (Reynolds, n.d.)

The alignment behaviour *(Fig. 2)* tries to get the agents to move in a common direction by finding the average heading of all its neighbours and moving in that direction.

Figure . Avoidance (Reynolds, n.d.)

The avoidance behaviour *(Fig. 3)* helps to prevent collisions and overlapping. Agents will determine if neighbours are too close and attempt to move away from them. The distance should be a radius smaller than that used to find nearby neighbours otherwise the other behaviours may not work.

Implementing the core behaviours works well in scenes with no obstacles, but in scenes where there are many, there can be issues with agents becoming stuck or not handling obstacles correctly. Our flocking was originally set up to only consider other agents and we needed a way to avoid objects in their path and steer away from them where possible. We managed to achieve this by using Unity's NavMesh. Our function uses the agents position to find the closest point on the closest edge of the NavMesh and then calculates a distance between the two points. When the agent gets within a set distance, we can create a new movement vector that will make the agent avoid the edges. This works well in most cases as the NavMesh wraps around every obstacle. An alternative way we could have achieved this would have been to allow each agent to get a list of every nearby object. This list could then be filtered to get the obstacles and a movement vector calculated to move the agent away.

To help fine tune how our flocks work we implemented a composite behaviour class which allows us to apply a weighting to the individual core behaviours. This weighting allows us to increase or decrease how much of an impact each behaviour has on the agent's movement.

We encountered the issue of erratic movement caused by the cohesion behaviour and have updated our class to gradually change the agent's vector towards the desired vector over time. This has helped to remove any jerky or sudden movements that were much more prevalent beforehand.

Our flocks can operate independently of each other. Each flock knows which members belong to it when it is created, we can use this information to filter out any neighbours that do not belong to the same flock and ignore them when calculating our movement. This allows for multiple flocks to be on screen at a given time. Two flocks may still encounter each other but they will both simply try to move past each other. This may not necessarily be needed for every game type and is more of a nice to have feature.

We originally planned to apply the flocking to the minions that travel down the lanes, however, due to the nature of MOBA style games, lane minions having very linear movement and this proved not to be a very good way of displaying how the flocking works. Instead we chose to move the flocks into the jungle where there are multiple obstacles, the agents would be able to roam more freely and eventually group up into a flock.

Decision Making

A behaviour tree was used to showcase decision making. The primary reason for choosing to implement and behaviour tree over a decision tree is that the behaviour tree seems to be more versatile and allows for a wide range of decisions to be chained together. Behaviour Trees work by switching between different tasks based on conditions determined by a composite node.

The behaviour tree was implemented in two stages. First, the composite nodes. Composite nodes work as a flow control; they are used to determine which behaviours are seen and executed or skipped. Two Composite nodes were implemented for this demo.

Selector:

Selector nodes execute their children from left to right. They stop executing when one of their children succeeds. If a Selector's child succeeds, the Selector succeeds. If all the Selector's children fail, the Selector fails.

Sequence:

A sequence nodes execute their children from left to right. They stop executing when one of their children fails. If a child fails, then the Sequence fails. If all the Sequence's children succeed, then the Sequence succeeds.

A decorator was also implemented, but many and the use cases are pretty distinct; only the invertor decorator was implemented.

Invertor:

The Invertor node works similarly to the not operator, effectively flipping the state of its child node. So if the child returns success, it becomes a failure.

With these core nodes in place, stage 2 began, and the focus was on creating custom execution nodes that make up the behaviour of the AI.

Custom nodes can be any be used to create any behaviour a unit can perform. All Customs nodes have to return one of three states.

Success:

The Node has successfully performed its action. E.g. attacking an enemy unit.

Wait:

The Node is still processing the action. E.g. walking to a location.

Failure:

The Node has been unsuccessful in its execution. E.g. attacking an enemy that was just killed.

Combining all of these different nodes makes the behaviour tree.

For this demo, only one behaviour tree is used for all the agents. The reason is that most of the agents would share similar behaviours, such as attacking a tower or another champion/minions. The only difference in the behaviour would be from retreating when health is low or waiting by a tower to regroup with champions. The whole behaviour tree can be seen below.

While implementing the behaviour tree was not too difficult, there were a few hiccups along the way, mainly with the custom nodes. The main problem is that many nodes require knowledge of game objects that can be destroyed. Meaning that sometimes, when another unit destroys a game object, they break for a moment until the garbage collector can clean up the memory and the list can be cleared and updated.

Another unexpected behaviour that was not directly caused by the behaviour tree was that champions would sometimes choose to go through the jungle when attacking a tower. However, this behaviour seems to be due to the navmesh picking the jungle as the most efficient route. This caused another unforeseen problem with the flocking agents. The Champions that wandered into the jungle would effectively never come out, as the attacking enemies take precedence over attacking a tower. Meaning that the Champion would stay in the jungle, killing any flock the cross its path. As this was noticed very late in development, the fix was not to allow champions to attack flocking agents.

Pathfinding

Conclusion

Individually the algorithms work well and can be demonstrated without many issues. Each algorithm was designed in its own test scene prior to being implemented into the main map which created some problems when we decided to finally bring everything together. It may have been beneficial to try and implement the algorithms into the main map from the beginning instead of creating them in separate tests scenes.

Decision making and flocking did not work together as well as we expected them to. We encountered issues where the scripts would try to override each other causing the AI to either move directly to a location without flocking or flock together and not move anywhere. We worked around this problem by separating the two algorithms and using them on different areas of the game.

While the pathfinding worked fine in the test scene this did not function as intended in the main map and we had to resort to using the NavMesh in place of it.

Overall, we were happy with MOBA as our genre of choice however we agree that the flocking may have worked better using a different game type.

Despite the issues encountered the algorithms still work in the final map and it is possible to play a full game.

References

Reynolds, C. (n.d.) *Boids*  
Available at: <https://www.red3d.com/cwr/boids/>  
(Accessed, 2021)