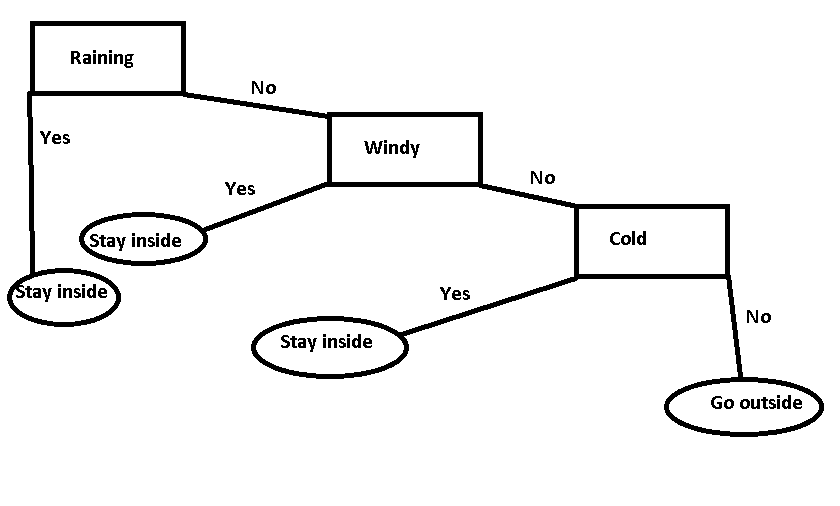
Investigation and Research Steven Smith

The main method being used in the creation of the coursework is decision making AI, namely decision trees that will dynamically make choices on what action is to be taken at any given time. The following paper documents investigations into several techniques to achieve the desired effect and research into how the investigation can be used in the coursework.

One basic method for constructing decision trees is by constructing them from a written example or diagram(Almuallim 2002) as seen below:



This example shows a simple decision tree that is displaying whether you should go outside or not. In this example, information is given about the weather; this could be given as a set of Booleans. The tree shows tests it makes in the rectangle boxes and the routes it will take from the results of the tests are in the paths coming off the rectangles. Finally the action that will be taken is shown in the ovals at the end of the tree.

Commonly, the tree will compare the information fed into it at each test and make a decision based on this information. For example, if there was Booleans that equalled the following:

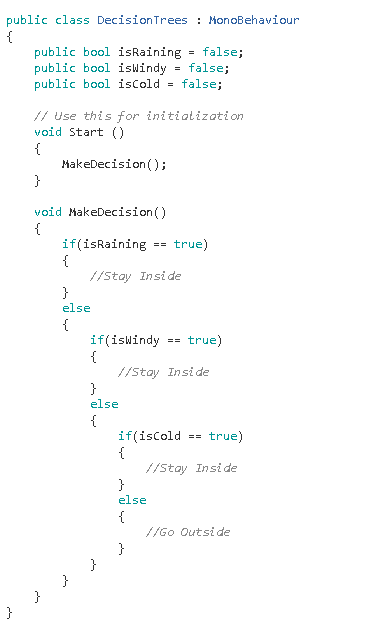
isRaining = false

isWindy = false

isCold = false

The decision tree would go to the first test (Raining) and check the information that was relevant to the test; in this case it would be the Boolean isRaining. As this equals false the decision tree would make the decision to go down the No route, this would continue until the decision tree hit the action it would take. In this scenario the action arrived at would be to Go Outside.

Drawing out the decision tree helps the programmer to think logically about how to insert the decision tree into code, this decision tree could be easily translated to code and would resemble the following:



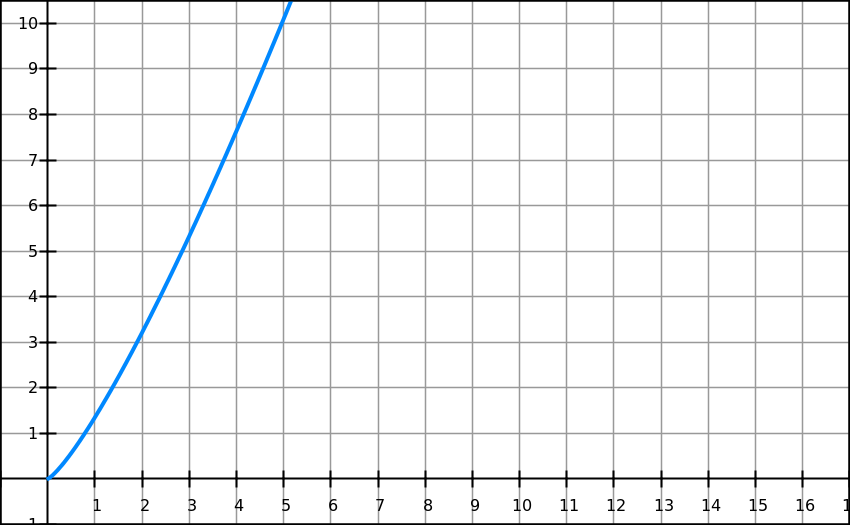
Creating decision trees by example is an incredibly simple and easy method of creating decision trees as once the decision tree has been drawn out it is always relatively simple to translate into functioning code. One major downside is that the code is neither maintainable nor reusable as it fits only the exact purpose it was created for.

This type of decision tree will be used for the Alpha and Beta personalities previously discussed in the pitch. The tests will, however, be influenced by the following sum:

f(x) = (x^0.8) \* (e^0.3)

Where x = the number of agents near another agent at any one time.

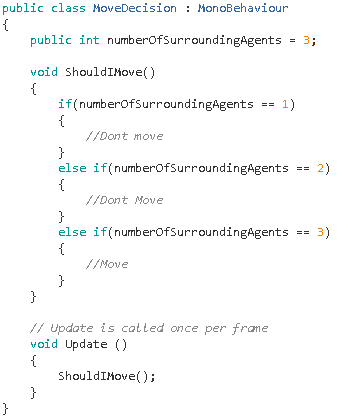
This sum gives out the following graph:



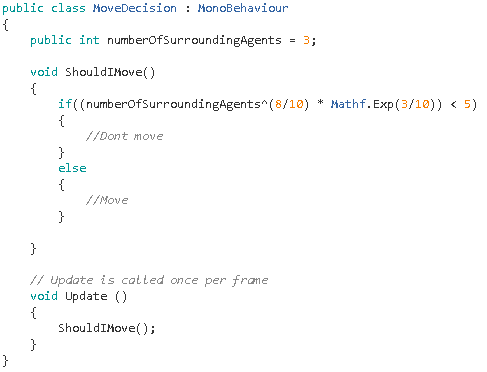
Graph made with: https://graphsketch.com/

This is useful as the number on the Y-axis is increasing exponentially as the X-axis increases, this is the exact behaviour we want for the finished game as the higher the count of agents near any other given agent the more the agent should want to make the decision to move. This example is for the Beta agent, for the Alpha agent the graph and sum would be reversed to force the agent to move when there are fewer agents in the area.

This method is being used over the simpler method of typing each value individually such as in the following example:



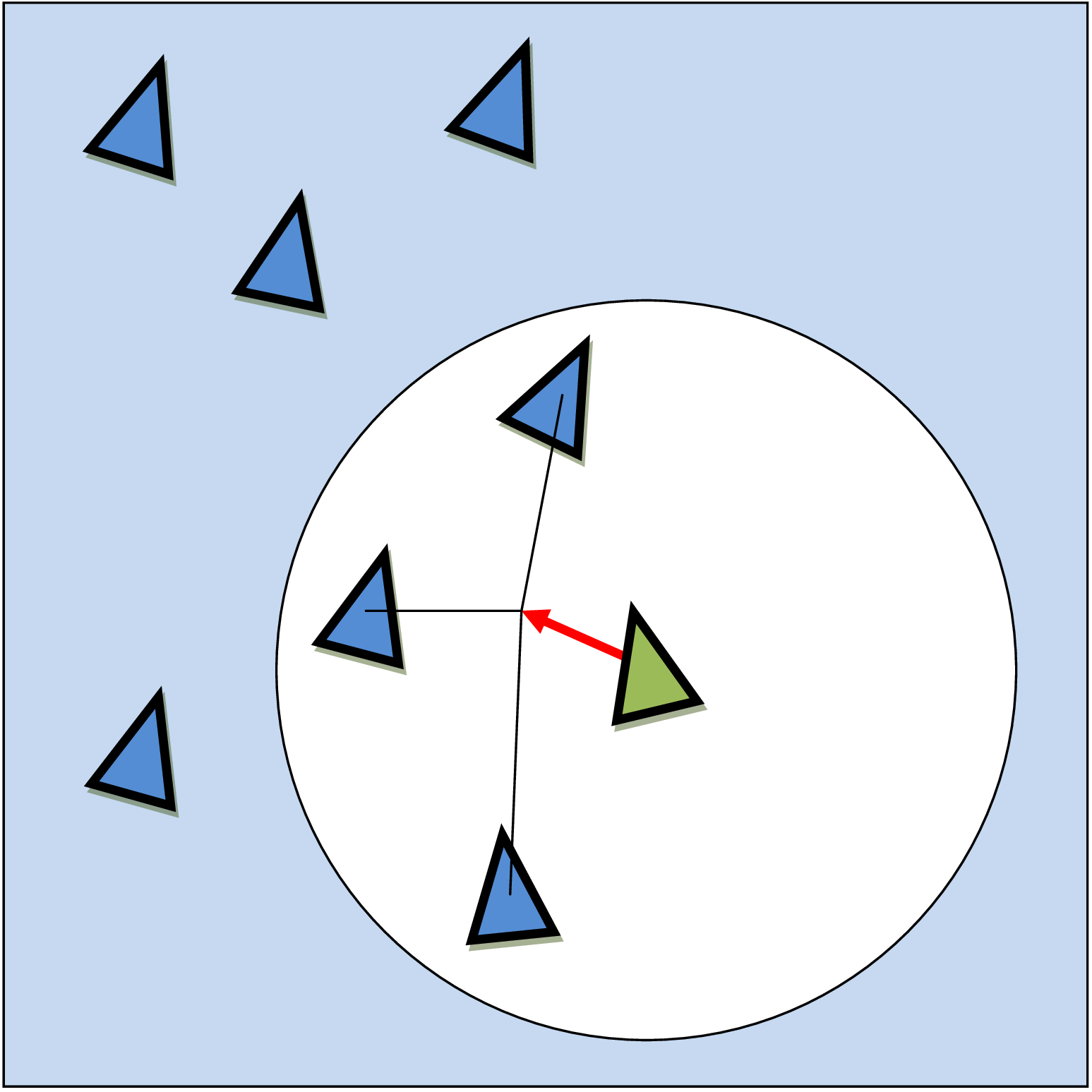
This method is not ideal as the decision tree could get out of hand quickly if the number of agents increases drastically. With the method using a sum it only requires one check to be performed and cuts down on the time it would take to write and run the code. An example of this follows:



This method also allows for fine tweaking of the sum and comparison values until the desired behaviour is achieved. This method is also being used for the Omega personality, however, a random element will be introduced to differentiate it from the two other agent types.

Another type of artificial intelligence that is going to be used in the creation of the coursework is steering behaviours, in particular, cohesion and separation.

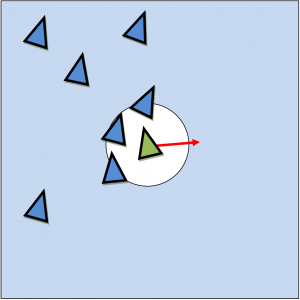
Cohesion is a behaviour which gives an agent the ability to approach and form a group with other nearby agents(Reynolds 1999). This steering behaviour suits the exact behaviour we want the Alpha agent to have as it will attempt to form groups with other agents. The picture below displays the behaviour expected from this steering behaviour:



Picture taken from: <http://scottsnowden.co.uk/flocking-lego-nxt-full-report/>

This steering behaviour is computed by finding all neighbouring agents within a set distance and adding together their current positions to form the desired position, the number of neighbours is tracked and the desired position is then divided by the total number of neighbouring agents. The current position is then subtracted from the desired position and finally normalized to get the steering velocity(Reynolds 1999).

The other steering behaviour being used is separation. This steering behaviour gives the agent the ability to maintain a certain separation distance from other agents nearby, which can be used to prevent crowding(Reynolds 1999). This steering behaviour fits the Beta agent as the Beta agent attempts to distance itself from groups of other agents. Again the picture below shows the expected behaviour of an agent with this steering behaviour:



Picture taken from: <http://scottsnowden.co.uk/flocking-lego-nxt-full-report/>

To compute this steering behaviour again all neighbouring agents are found, the direction vector is calculated by subtracting the positions of the agent and the neighbouring agents, the neighbours are counted and the result is divided by the number of neighbours and normalized. This then returns a direction vector with the desired direction leading away from the surrounding agents[2].

Cohesion will be used for the Alpha agent and separation will be used for the Beta agent. Both these steering behaviours will be added to the Omega agent and it will choose at random when to move and what steering behaviour to perform.

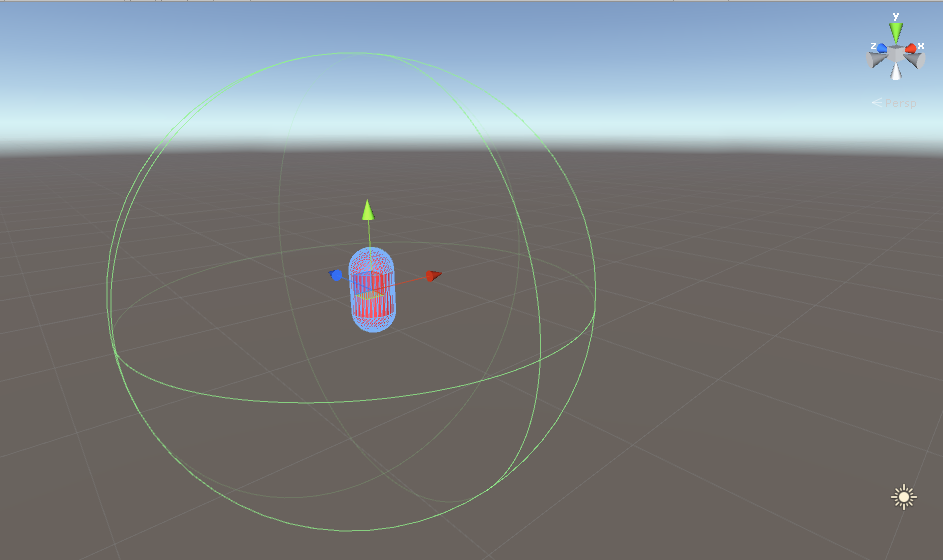
In conclusion, most of the methods looked at will be used to create the full coursework with decision trees handling the decision making of the agents and steering behaviours handling the movements of the agents. If done correctly, the methods looked at should create the exact behaviour that is being looked for from each agent type.

**References:**

* **Almuallim, H ET AL. 2002. Development and Applications of Decision Trees. In: Leondes, C. Expert Systems- The technology of knowledge management and decision making for the 21st century. Volume 1. Massachusetts: Academic Press. pg53-pg77**
* **Reynolds, C. 1999. Steering Behaviours for Autonomous Characters. California: Sony Computer Entertainment America**

Implementation Steven Smith

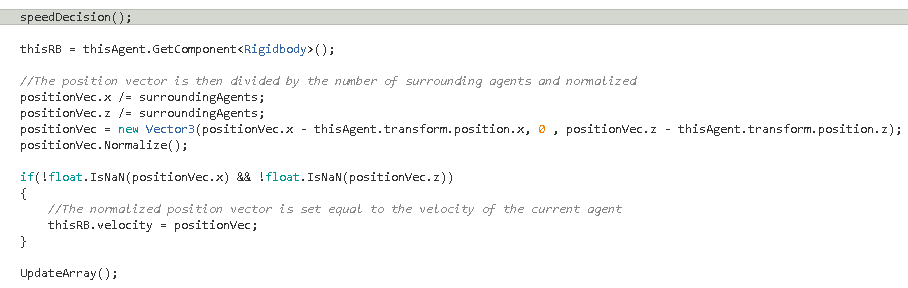
The project undertaken was to showcase how different agents would react if given different steering behaviours and decision trees. The first thing created was a script to tell each agent how many agents were surrounding it, this was at first achieved by using a sphere collider and using the OnTriggerEnter() and OnTriggerExit() functions:



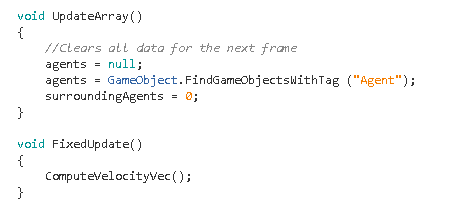
The collider was made a trigger and oversized to act as a bubble for the agent, when another agent entered the trigger the number of agents increased by one and when an agent left the trigger the number would decrease by one. This approach worked but was ultimately found to be obsolete in favour of another approach.

The second approach created was to create an array of all the agents at the beginning and loop through them, checking the distance of each against the current agent. The distance is calculated by setting a float equal to the distance between the agent attached to the script and the current agent in the array. This is done with the use of the Vector3.Distance operator, once the float is calculated it is used to check if the distance between each agent is less than ten units. If this is returned as true the agent is seen as being in the surrounding area and the number of surrounding agents is increased by one. The current agent in the arrays position is added to the position vector here too but this will be looked at later on. The second approach to fetch the number of surrounding agents is shown here in code:



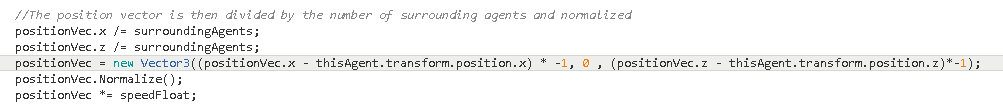
This approach was far more favourable to the first as it enables the developer to access all the information on each surrounding agent. This is useful as the next stage implemented was to create the cohesion steering behaviour for the alpha agent which requires the position of the agents close to the agent attached to the script. In the code above, it is shown that if an agent is in the surrounding area of the main agent the agent in the surrounding areas position is added to a position vector named positionVec. This position vector is then divided by the number of surrounding agents and the main agent’s position is subtracted. After this the position vector is normalized and a check is made to ensure the position vectors x and z position is valid the position vector becomes a velocity vector and is added to the rigid body of the main agent. This can be seen in the below code:

In the code it is shown that the UpdateArray function is called after this, this was added in to clear the surrounding agents and the position vector as if they are not cleared it creates a problem as each is increased each frame so need to be cleared at the end of each frame to keep all information correct. The UpdateArray function can be seen below:



It is also shown here that the main function of ComputeVelocityVec is called on each fixed update. Fixed update was chosen to keep the movement of each agent smooth and minimize any possible juddering from the agent’s movements.

The beta agent uses the separation steering behaviour which operates extremely similarly. The one difference being that the velocity vector is multiplied by negative one to get the opposite direction from surrounding agents. This can be seen below:

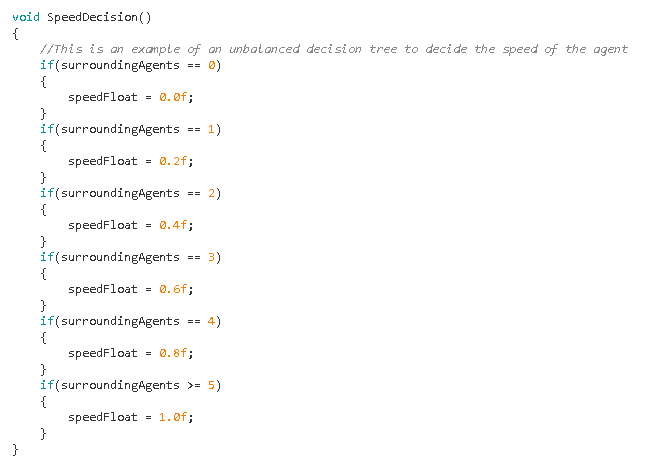


The next function implemented was to implement a decision tree, the original idea was to use the decision tree to make the agent choose if it wanted to move or not but as there was only two end states (move or not move) it did not work well in practice. Instead a float named speedFloat was created and the decision trees lead to what the speedFloat will equal at the end. The speedFloat is then multiplied by the velocity vector before it’s added to the agents rigid body which affects the agents speed. The speedFloat is chosen depending on the number of surrounding agents, as the Alpha agent is supposed to prioritise grouping up with other agents it gets quicker the less agents it is surrounded by, the opposite was done for the Beta agent as it prioritises being further away from other agents. Both decision trees are shown below:

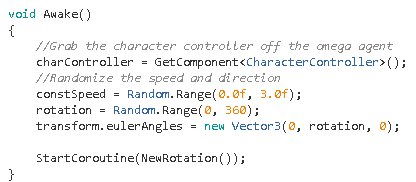
Alpha decision tree:

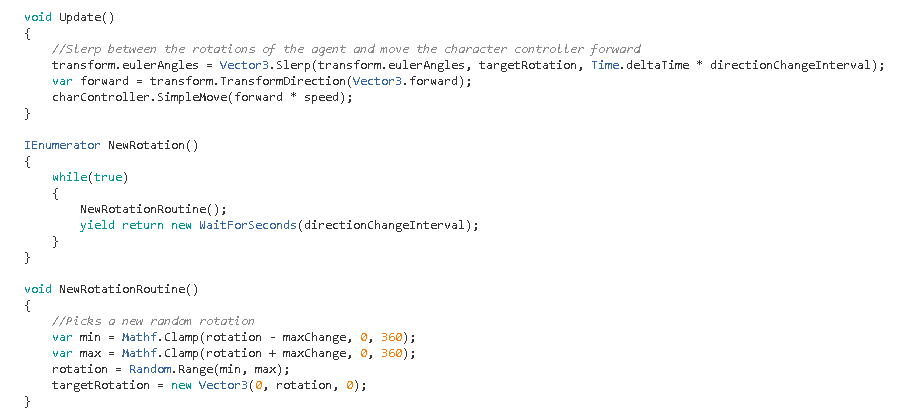


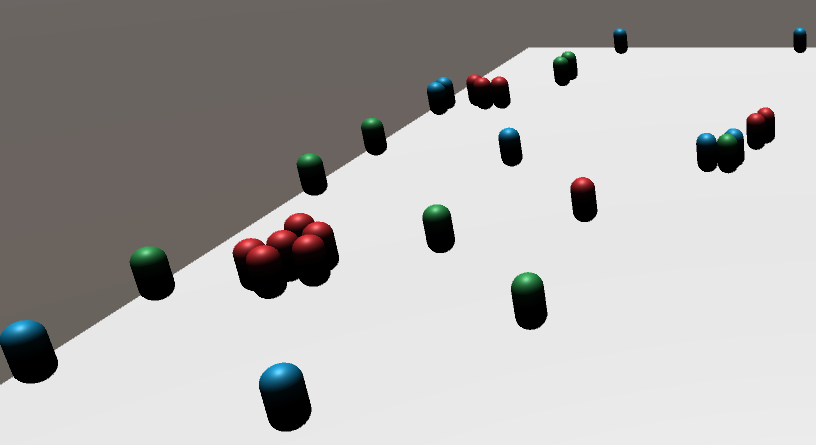
Beta decision tree:



As can be seen, each agent has a different type of decision tree; the alpha agent tree is an example of a balanced decision tree whereas the beta agent tree is an example of an unbalanced decision tree. This approach was chosen to show the difference between the two types of decision tree and has no real effect on the outcome of the application.

The Omega agent was added in next, the omega agent was designed to stop the alpha agents from clustering together and not moving. As a result of this the omega agent’s movements are completely random. When the omega agent wakes the rotation of the agent and the speed are randomized. The rotation is then set and the character controller component is retrieved as this is what is used to move the omega agent, a character controller was used as it provided the simplest means to move the agent using the CharacterController.SimpleMove function. After these starting variables have been calculated a new co-routine is started to choose a new random rotation. This is shown below:

The co-routine goes straight into a function that chooses a new random rotation and waits a few seconds before looping through the function again. The random rotation is chosen by finding the agents current rotation and taking the maximum change away from the current rotation for the minimum and adding on the maximum change for the maximum. This was done to stop the agent from snapping too far in each direction and ensure the agent would travel smoothly in an even curve. The rotation is then chosen using the Random.Range function getting a rotation between the maximum and minimum rotations found previously. The new rotation is then set and the update function handles the Slerp animation to stop the agent from juddering and moves the agent forward at a constant speed. This is seen in the code that follows:

A few problems encountered when implementing this code was that the Alpha agents would group up together and remain still as their speed would equal zero from the decision tree, an example of this is shown:

This was not the desired behaviour so the alpha agents code was changed to ignore other alpha agents, this fixed the grouping between alpha agents but they would still at times surround one agent and box it in.

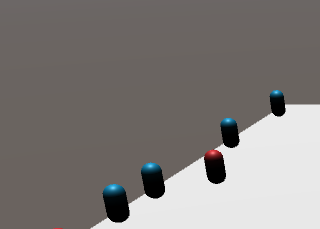
Another problem encountered was lag and an extremely low framerate (around 12 frames per second), many fixes were employed that failed until the debug log was found to be the case of the lag as it was logging a message every frame for every agent in the application. This was removed from the code and the framerate recovered immensely.

To create a unique experience each time the application is run, a simple agent spawning script was created to create unique spawn points for each agent every time the application is run. The script can be seen below:



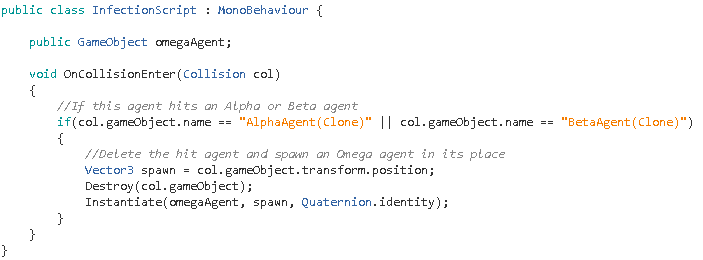
This script was useful as it loops through a certain number of times that can be changed by the developer, this makes it easier to change the number of agents that are in the scene until the developer feels the right amount has been chosen. Each loop creates three random spawn points for each agent and then spawns that agents prefab at the randomly generated location. This is a simple way to increase replay ability of the application as the agents should act differently every time.

Another problem that was encountered was that the beta agent would move to the edge of the map and hit the maps invisible wall and remain still, an example is shown below:

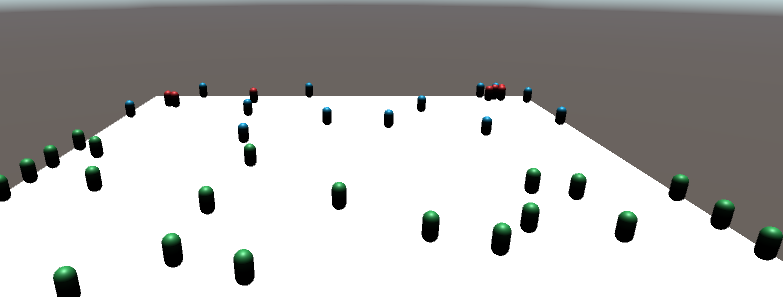


Originally, this problem was to be solved by using a NavMesh to ensure the agents never got too close to the edge but due to time constraints could not be implemented and remains a problem in the final application.

The first scenario created was to showcase the steering behaviours and decision trees in action, a second scenario was created in which an omega agent acts as an infection and if it touches another agent that agent becomes an omega agent. This scenario was added to determine what agent out of the alpha and beta agents would remain uninfected the longest. A simple script was created and added to the omega agent for this scenario, the script is shown below:



This script activates when the game objects collider the script is attached to hits another collider, a check is made to ensure the omega agent is not hitting another omega agent, if this check is true the collided objects position is saved and the collided object is deleted, a prefab of an omega agent is then spawned in its place. From this scenario the beta agents kept from being infected the longest which was the suspected outcome for this scenario, a picture showing this is below:



In conclusion, the project works as intended as it was created to show different behaviours in agents, however, there remains a few lingering problems that, if solved, could increase the effectiveness and interest of the algorithms present.

Evaluation Steven Smith

This evaluation is to determine the effectiveness of the algorithm created for the artificial intelligence coursework. It will take into consideration any successes and failures of the algorithm and evaluate each success or failure encountered. The technical side of the algorithm will be looked at including computation time, ease of use, flexibility etc. The results of the algorithm will also be compared to results found during the investigation stage of the coursework. Finally the code will be analysed for strong and weak points and improvements that could be made to the algorithm.

One of the successes of the algorithm are the steering behaviours for both the alpha and beta agent, both work fully as intended, the code for both steering behaviours was taken from the pseudocode given in the lecture on flocking behaviours. Cohesion was used on the alpha agent and separation was used on the beta agent as both fit the behaviour desired for each agent perfectly. Both these steering behaviours are part of a larger flocking behaviour but are not used in this way in the coursework as flocking is not what is being investigated. Instead the investigation was into how agents with different behaviours would interact with each other. In the future, the algorithms for each agent could be changed to make their behaviour much more different from each other; an entirely new agent with different behaviours is also a possibility that could be looked at. For the coursework given, however, the algorithms for both agents worked under two separate scenarios, one where there was no interference and one where if an agent was touched by an omega agent the agent that was touched would then change into an omega agent.

A failure of the algorithm is that the agents will jump up on top of each other if they collide, this happens as there is no check to separate an agent if it collides with another agent so the only possibility is for the agent to go over the top of the other agent, this takes away from the immersion of the investigation and looks less appealing overall. To fix this, a check could be made if an agent collides with another agent and send them in opposite directions of each other. This would ensure two agents would never jump on top of each other. This problem happens in both scenarios as agents are colliding with each other in each scenario within the application. This is harmless, however, and will not result in a crash or bug of any kind and is purely an aesthetic problem.

Another success of the algorithm is the omega agent’s script as it performs perfectly, the omega agent was planned to move completely randomly to keep the other agents moving and it performs this without fail. This is the case as almost all variables in the omega agent’s script are randomised to a limit. This creates the truly random behaviour of the omega agent, in future it may be advised to add a check for if the omega agent is going to collide with a wall and have it change direction, this would stop the omega agent from walking into walls for extended periods of time. The omega agent performs well in each scenario as in the first scenario it ensures the alpha and beta agents are in motion which is what it was created for. In the second scenario it also performs well as it is able to, in time, infect a large amount of agents and turn them into omega agents which was the purpose of the omega agent in the second scenario.

The second failure of the algorithm is the alpha agents’ tendency to group up, this is not seen as ideal as the idea was to have all agents spaced out and interacting with agents of a different type. To stop this the alpha agents had their scripts changed to no longer take into account other alpha agents, this solved the alpha agents grouping up together but they would still group up around other agent types creating large clumps of alpha agents. This could have been solved by creating a check to see if there was a large number of alpha agents surrounding another alpha agent and if this returned true the alpha agents would disperse in a random direction for a few seconds before reverting to regular behaviour. This behaviour occurred under both scenarios, however, it was particularly troublesome in the infection scenario as if the alpha agents grouped up and were touched by an omega agent the full group would turn into omega agents, this had the possibility to drastically change the outcome of the infection scenario.

The third and fourth success is the spawning algorithm and the extra scenario that was added. The spawning algorithm works well as it is intended to randomise spawns of the agents every time a new scenario is played which it does very well. There is not much to improve upon this algorithm except from possibly checking the agents are not spawning inside each other. The second scenario was created to show how the different behaviours of the alpha and omega agents would affect them differently, the scenario created shows the omega agent turning other hit agents into an omega agent, the scenario was a success as the beta agent almost always outlasts the alpha agent. These repeatable results showcase how the different behaviours in the algorithms of the agents affect them in different scenarios.

The final failure of the code was the beta agents’ algorithm causing them to stick to walls, this was a result of the steering behaviour created as it drives the beta agent away from other agents and towards the maps limits. To solve this, a NavMesh could have been added and checks could have been performed to check if the beta agent was nearing a wall and if so, steer the agent away from the wall, this would have stopped the beta agents from facing a wall and remaining still. This problem occurs in both scenarios and takes away from the constant movement that was intended for the agents.

The time taken for the CPU to respond on average is around 15ms and the average frames per second is around 68 frames for a computer with an Intel Core i7-2600k CPU, 16.0GB RAM and an NVIDIA GeForce GTX 980 GPU, the ease of use of the application is high as it is a demo instead of a program that the user must learn, all the user has to do is simply chose a scenario and watch the application load the scene, the menu is simple with descriptions of each scenario that can be chosen and a large red exit button is displayed in each scenario that, when pressed, takes the user back to the main menu. The time taken to implement the algorithms was relatively short as heavy investigation and research was performed beforehand, this ensured there was no time wasted on being stuck and making no progress in the creation of the algorithms. The algorithms have been fully documented and commented to help with the flexibility to modify the algorithms in future, this ensures any developers that try to use the algorithms will both know how they work and know what to change if they want different behaviours from the agents.

There are a lot of changes from the algorithm described in the investigation and the final algorithm. The main change is the decision tree, originally the decision tree was going to be used to decide if an agent would more or not, this posed a problem as it only gives two final states. An equation was also created to help decide if an agent would move, this was also scrapped and replaced with a decision tree that decides the speed of the agents. This approach was chosen as the speed of an agent can have several outcomes allowing for more fleshed out decision trees. The other major change is that the NavMesh for the agent algorithms was never implemented.

The slowest part of the code is the update function in the omega agents algorithm, this is because calling the Lerp function every frame is relatively intensive when there is several omega agents calling it at once, there isn’t much solutions to this problem as the Lerp function is being called to keep the omega agents moving smoothly so it is a trade-off between processing and the desired behaviour from the omega agent.

In conclusion, the algorithms provide several different behaviours for each agent involved and shows how different behaviours make the agents react differently to each other, this was the main aim of the algorithms created, due to this the algorithms can be determined to be successful.

Some improvements such as adding an extra agent with a new steering behaviour and decision tree and implementing a NavMesh could greatly increase the main aim of showing different behaviours in agents.

Reflection Steven Smith

This reflectance essay will focus on my own performance and an evaluation of myself on how I handled the coursework as a whole and what I have enjoyed and disliked from the entire AI course.

My strongest point in the full coursework was the original idea and the investigation and research, as the coursework allowed you to pick what you wanted to pursue I was able to pick a topic that was interesting to me which helped me with the original pitch as I was able to easily describe and pitch my original idea rather than being made to pick one from a list. It also helped me in the investigation and research as investigating something that is interesting to you is much easier than trying to force yourself to investigate a topic you have little interest in. This resulted in a strong amount of research and gave me a solid starting point for the implementation stage.

The implementation stage was more of a struggle as coding was enjoyable but I found myself lacking in the documentation as I was getting lost in what I was doing, this resulted in me taking less notes than I would have liked and possibly resulted in a weaker implementation paper because of this. I also struggled to comment code and had to go through all the code at the end and document it all which was much more time consuming than doing it during the first pass. I also failed to make the coursework as fleshed out as I would have liked due to time constraints, this could have resulted in me missing out on further learning opportunities.

The evaluation paper was my weakest part of the coursework as I struggled to hit the required word count even after I had talked about the algorithms completely, this resulted in me possibly repeating things already mentioned and a weaker paper than what could have been accomplished. The evaluation paper was also affected by time constraints.

In the future I should manage my time better as I found myself being dragged into working on other coursework from different modules and giving this one the least attention, this resulted in the last half of the coursework being fairly rushed and possibly not being up to the standard of the first half of the coursework that I took my time on. To improve on this I should write up a study plan that I will stick to that will divide my time up equally between all modules in the course. I also struggled with referencing for the investigation and research as it was the first time I had done Harvard referencing before, however, a lecture was supplied to help with this which ensured I knew the correct formatting etc. I still struggled with where the references should go and how to find good papers relating to the topic area I had chosen. In future, to prevent this, I should make sure I understand the concept of Harvard referencing fully before moving on to implementing it into a paper.

I have learned several new techniques that will help me in the future, steering behaviours can be used in many different scenarios and I now know how to implement flocking fully. This coursework also helped me with debugging which was an area I was particularly weak in, I would get stuck on errors for days before this but now I am learning how to debug properly it takes much less time to debug any code that is causing problems. I am now fully confident in balanced and unbalanced decision trees, how to implement them and include random elements within them. The structure of my code has also improved, making my code easier to navigate through and understand.

I can now confidently say I could implement several different types of steering behaviours and couple them with decision trees to create different behaviours in agents. Before this coursework I had very weak knowledge in these areas even although artificial intelligence has always been a topic of interest for me. I feel from this coursework, I could now confidently create a simple AI agent for any future games I develop as either part of the university’s syllabus or as an independent project.

I enjoyed looking into steering behaviours as I found them an interesting way to create movements for AI agents. Pathfinding algorithms also took my interest as it is obvious how useful a strong knowledge of pathfinding can be when creating an AI agent. Decision making was interesting as it showed how AI agents change their behaviours in games. I had trouble understanding fuzzy logic as I found it to be quite complex when compared to the other AI techniques, I could however see the usefulness of it. Procedural content generation was also not as interesting as the other fields; this was more due to personal interest rather than the lectures/labs to do with the topic. In future I will hopefully continue to develop my knowledge and understanding of artificial intelligence in games as I feel it can do nothing but help in many real world scenarios that may take place in the video game industry.

In conclusion, I feel the first half of my coursework was strong as I put a lot of time and effort into it but due to poor time management the last half of my coursework was possibly lacking. I would like to work on this issue and solve it immediately. Overall I have enjoyed the topic area of artificial intelligence and the coursework as a whole. I also feel the reflective paper has helped highlight how important further research is for the future and it has also helped to highlight some key areas I could improve on whilst also highlighting areas I am strong in.