14128381_Tubb_Nathan_DAC619_AE1_Report

AI FOR GAMES

Nathan Tubb | 02/12/20

Table of Contents

Chosen Algorithm: Hierarchical State Machine

As a non-software student, having no previous experience with AI solutions, I felt that I should choose an algorithm that lent itself to a more logical and simplified design mindset. This was to not prevent me from creating more complex and interesting Ai behaviours in the timeframe of the unit. After researching viable options, I found that State Machines (SM) are considered to be a time-efficient, popular and an effective approach to solving ai related problems in games (Sweetser and Wiles, 2002; Gill, 2014).

One known problem with basic SMs, which became apparent to me early in my design process, is that they can often become large, unclear and do not scale well (Millington and Funge, 2009; Sweetser and Wiles, 2002). The solution I came to for this was to use a Hierarchical State Machine (HSM), which allows for the creation of what can be referred to as “super” or “Outer” states as mentioned in (Alur, 2003). These states act as ordinary states themselves, however, have the added capability of utilizing their own internal state machines, allowing for a more structured design. An example of this might be having an outer Combat state with an internal state machine running combat related states such as retreat or attack enemy.

PROS:

The use of a HSM allowed me to create a clear structure for the implementation of my Ai, allowing me to separate agent behaviours into defined internal state machines as appose to creating one single SM that contained over twenty-one individual states all needing transitions to one another. Furthermore, it allowed for states to be more abstract and atomic making them more optimized in execution calls at runtime.

Another important benefit of using a SM is that it catered easy debugging and implementation. This allowed me to quickly solve issues that arose in testing and spend more time trying to optimize or improve the designs of specific behaviours. I found the easiest way of stepping through the algorithm in real-time was by using debug logs in the enter function of each state allowing me to track the agent’s decisions whilst watching them play.

Finally, the modularity of SMs allowed for a more iterative design process, an example of this was the addition of the regroup state. Originally overlooked in the original design this state was abstracted from the retreat state, which initially housed logic for both behaviours and was too complex. To make this change I created a new state from the template and pasted the logic in from the retreat state, then added the transitions needed to the appropriate states and updated the design document. A process which would have been far more time-consuming in algorithms such as Goal-Oriented Behavior or Behavior trees which would have needed far more tweaking and adjusting to accommodate the change.

CONS:

Although the use of a HSM allowed for a broader and more in-depth design overall, the initial problem regarding scaling complexity still applied. This mainly affected the design process of the project and as best described by Gill, (2014) “it is difficult to arrange large state-machine diagrams so that they are readable and understandable.”. This issue resulted in the design of the project going through multiple iterations before it became abstract and simple enough to understand.

As explained by Sweetser and Wiles, (2002) an ai using a state machine can only make as decisions dependent on the conditions the design allows for. This can create repetitive conditions, in which agents become predictable, requiring more complex or an increased number of conditions to make them appear to have intelligent or dynamic decision making. This can result in messy or confusing code.

DECISION TREES:

As another simple and straight forward alternative to State Machines, Decision trees are an extremely modular algorithm that allow for finely tuned decision making, that can appear predictive or intuitive. As discussed by Millington and Funge, (2009) and Fernández and Barragan, (2011) this algorithm uses a system containing condition nodes that recursively evaluate predefined conditions to choose the best leaf node “action” to perform.

Using this algorithm for the project would allow for agents to appear far less reactive with a greater sense of decision making, as unlike state machines agents they can evaluate all possible conditions each tick. Furthermore, well-designed decision trees are very efficient, prioritizing common decisions can improve this further as the algorithm does not need to test every created decision each traversal.

However, whilst decision trees are modular and efficient algorithms that unlike state machines, are scalable, they are well known for needing a lot of tuning and tweaking to avoid poor decision making as small changes in values can result in drastic changes of behaviour (Millington and Funge, 2009;). This would have made balancing the choices made by ai using this algorithm a more time-consuming task which may have resulted in a less advanced system overall.

Furthermore, whilst decision trees at a basic level are considered very simple and basic, Millington and Funge, (2009) state that extensions to the algorithm can fast become sophisticated and in-depth. This would have required a steeper learning curve than the already well-documented HSM extensions of for the State machines algorithm. This made me feel that given the time frame, attempting a HSM was more feasible.

GOAL ORIENTED ACTION PLANNING (GOAP):

More complex than the two previously mentioned algorithms I investigated using GOAP for this project as it allows for more flexible and realistic decisions from the Ai. The system is a development on Goal-Oriented behaviour, in which an agent can select a pre-defined action or action sequence based on how well it satisfies a goal. GOAP in comparison generates action sequences in real-time. Taking a goal the algorithm works backwards using a planner to find actions, who's postconditions satisfy the pre-condition of said goal. Actions are also evaluated based on a cost heuristic defined by the developer to generate the most efficient plan. This plan is then given to an agent who then performs it. An example of this given by Owens, (2014) can be seen in appendix A.

The main benefit of using this algorithm is that it does not present the same issues as State Machines in maintaining set transitions between states. Transitions of behaviour are handled by GOAP itself rather than the developer needing to predefine them in code. This is supported by Owens, (2014), who explains that due to the algorithm evaluating actions cost and availability in real-time, completely different action sequences or transitions can be generated for the same problem. This increases how realistic and dynamic agents appear.

Another benefit discussed by Owens, (2014), is that by not having to maintain constant transitions the algorithm is very scalable, the developer only needs to assign a new action with cost and satisfaction values when implementing it rather than working out its precise transitions. This allows for much larger systems to be designed without having to greatly manipulate their design diagrams or documentation, unlike SMs.

However, a drawback to the algorithms reliance on variables is that if not handled or set up appropriately the algorithm can generate undesirable or in-correct behaviours which can ruin the overall look of the Ai. This often requires much time spent tweaking or adjusting to achieve the desired results.

Other drawbacks to the algorithm are discussed by Long, (2007). Firstly, debugging a GOAP system can be difficult due to its unpredictability, actions selected for the current goal may fail planning checks due to the poor assignments of values and parameters or errors in their code. Solving these checks can require the developer to step through each action to identify why it might fail in the given circumstance.
Secondly, the framework of this algorithm is a complex and difficult process. As quoted by Long, (2007), "The GOAP system requires a goal manager, planners, A* goal, A* map, action container before a start can be made on the goals and actions.".

Taking these factors into consideration, I decided that the simplicity and ease of implementing and maintaining the HSM far outweighed the benefits in decision making gained by developing a GOAP system. I concluded that I would have more success in trying to optimize a more simple algorithm, than trying to get such a complex one working in the first place. However, with more time , I believe a GOAP system would be feasible for this project.

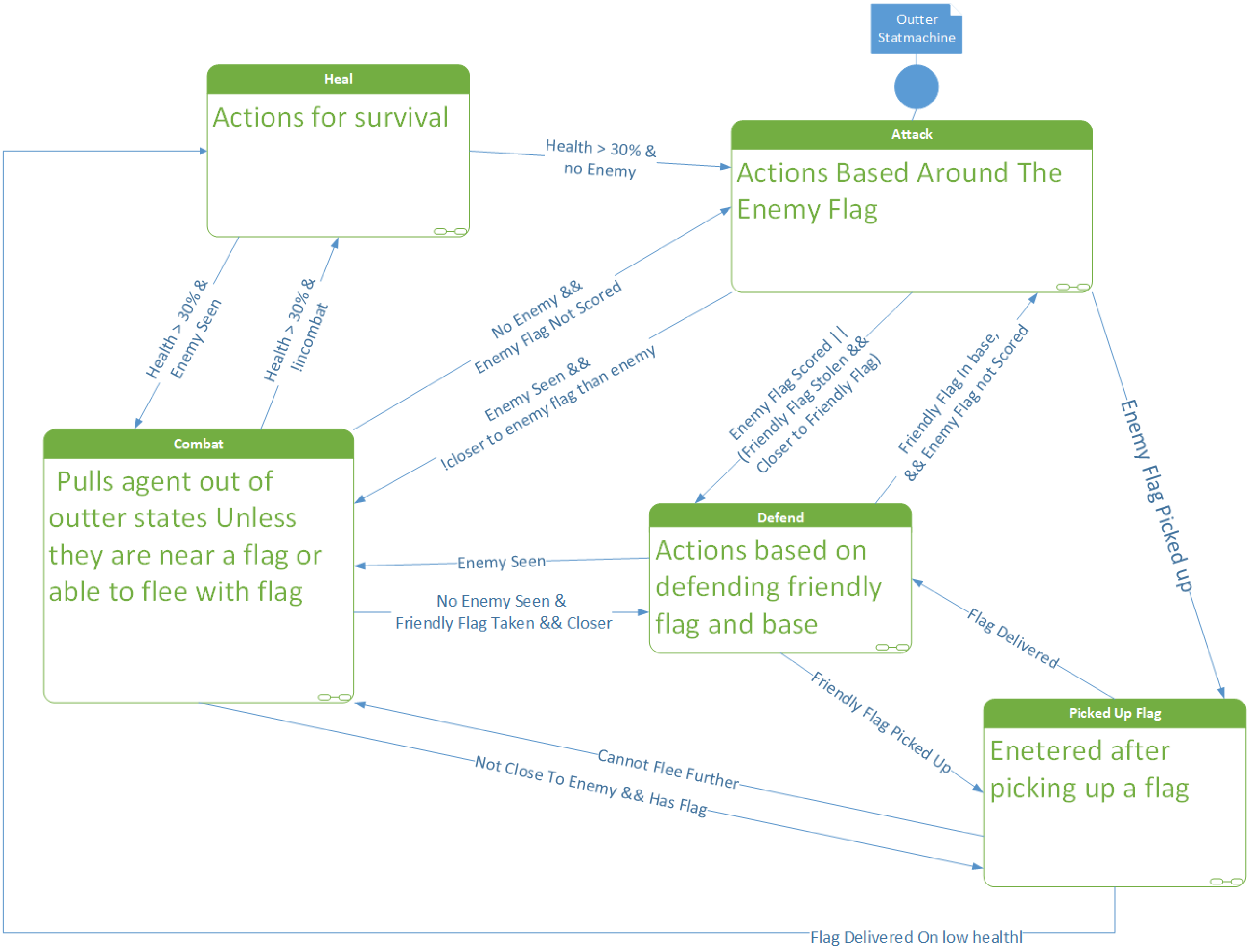
Design

My Design split the agent’s behaviour into five different states, which are based on the generalized behaviours seen in a capture the flag game mode, each outer state then runs its internal state machine, which performs actions relevant to that behaviour dependent on the conditions of the world at that moment. All state machine designs are shown below.

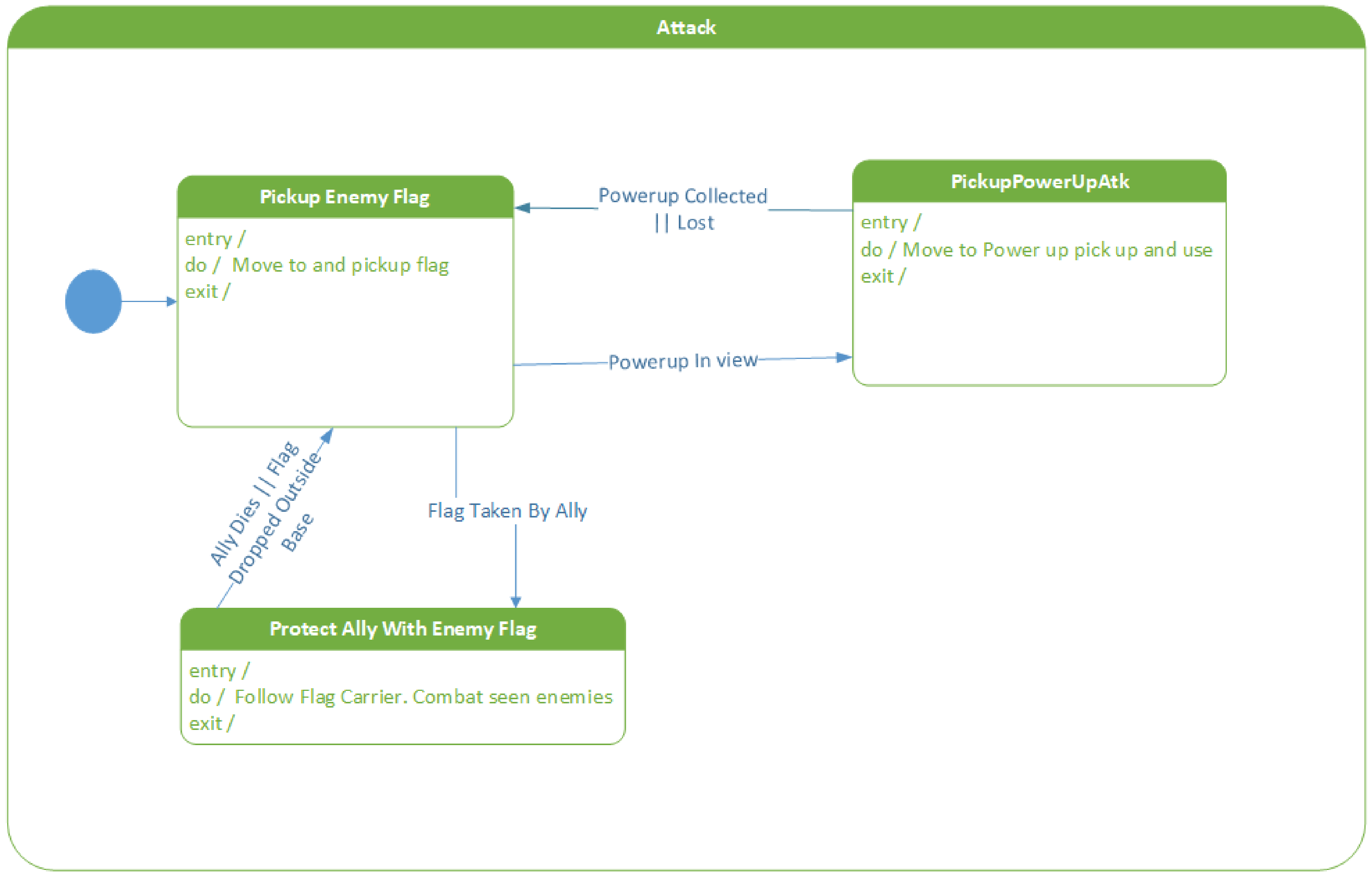
STATE MACHINES

Outer State Machine

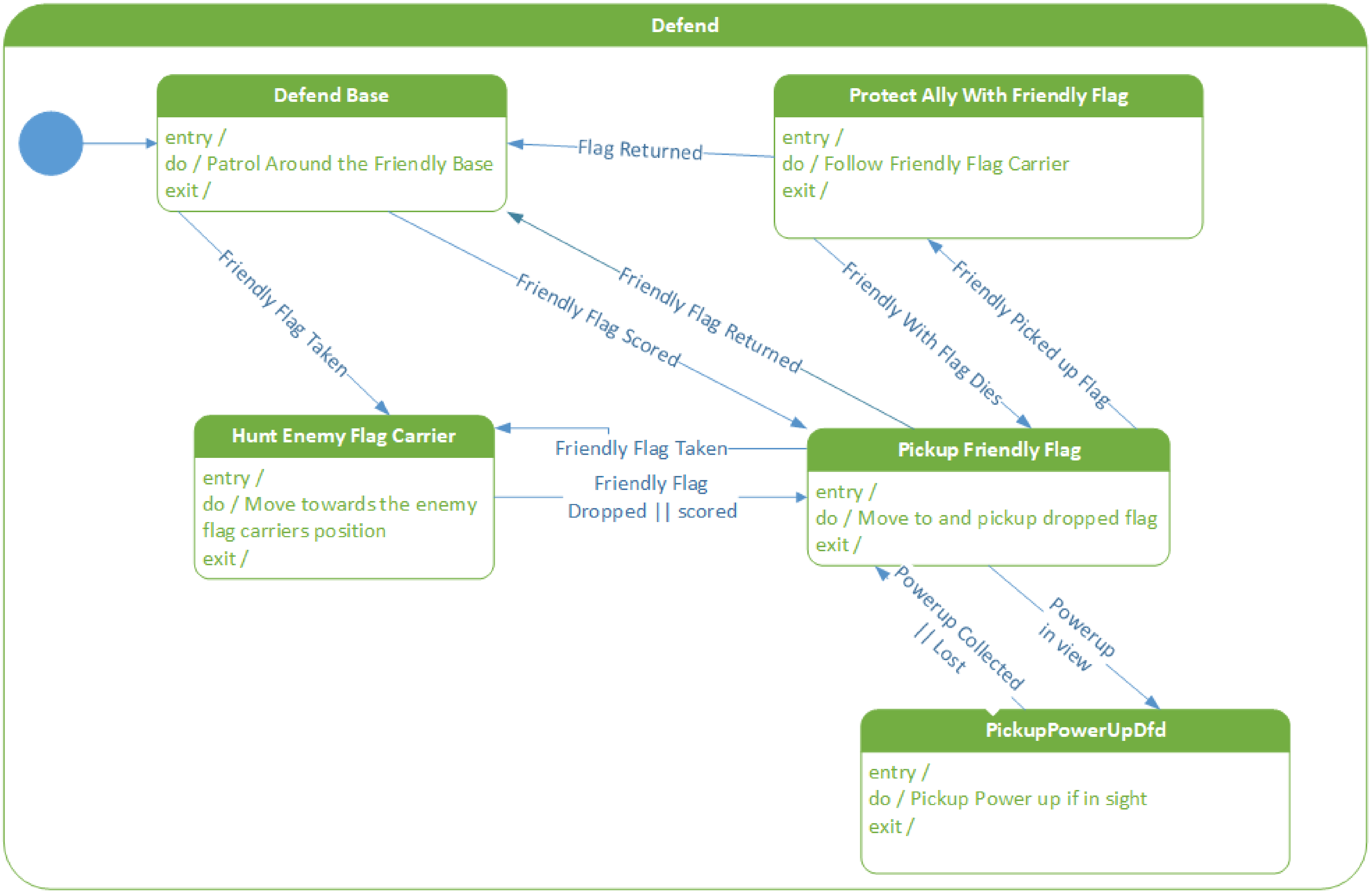
The overall design allows for agents to be pulled into the combat state regardless of the current state when they see an enemy, however, this can be blocked by certain conditions. This allows the agents to appear as though they are making more intelligent decisions rather than just fighting mindlessly.


Outter Statmachine
Heal
Actions for survival
Attack
Actions Based Around The Enemy Flag
Combat
Pulls agent out of outter states Unless they are near a flag or able to flee with flag
Defend
Actions based on defending friendly flag and base
Picked Up Flag
Enetered after picking up a flag
Health > 30% & no Enemy
Health > 30% & Enemy Seen
Health > 30% & !incombat
No Enemy && Enemy Flag Not Scored
Enemy Seen && !closer to enemy flag than enemy
Enemy Flag Scored || (Friendly Flag Stolen && Closer to Friendly Flag)
Friendly Flag In base, && Enemy Flag not Scored
Enemy Flag Picked up
Enemy Seen
No Enemy Seen & Friendly Flag Taken && Closer
Flag Delivered
Friendly Flag Picked Up
Cannot Flee Further
Not Close To Enemy && Has Flag
Flag Delivered On low healthl

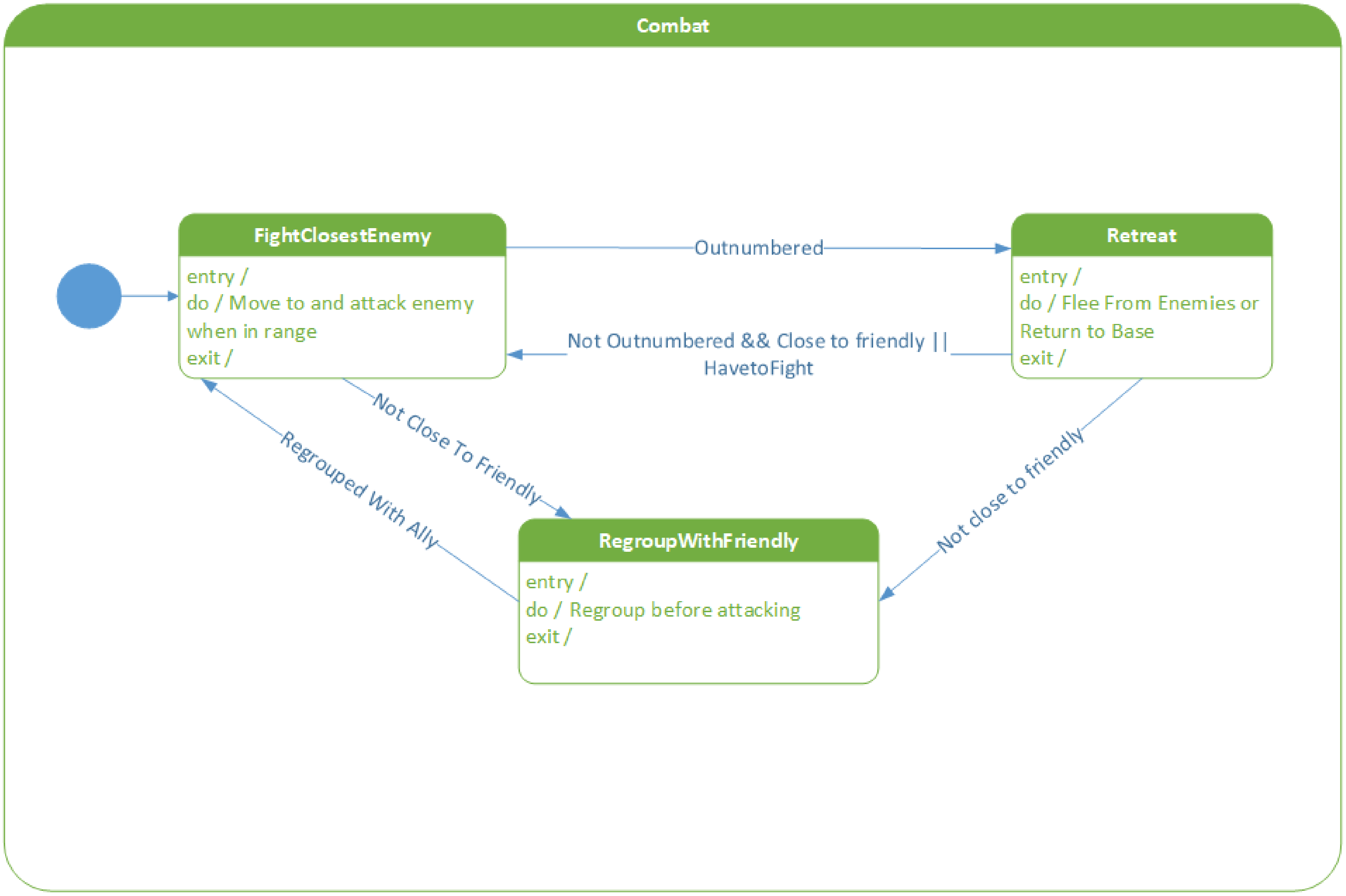
Attack State Machine


Attack
Pickup Enemy Flag
entry /
do / Move to and pickup flag
exit /
Powerup Collected || Lost
PickupPowerUpAtk
entry /
do / Move to Power up pick up and use
exit /
Powerup In view
Ally Dies || Flag Dropped Outside Base
Flag Taken By Ally
Protect Ally With Enemy Flag
entry /
do / Follow Flag Carrier. Combat seen enemies
exit /

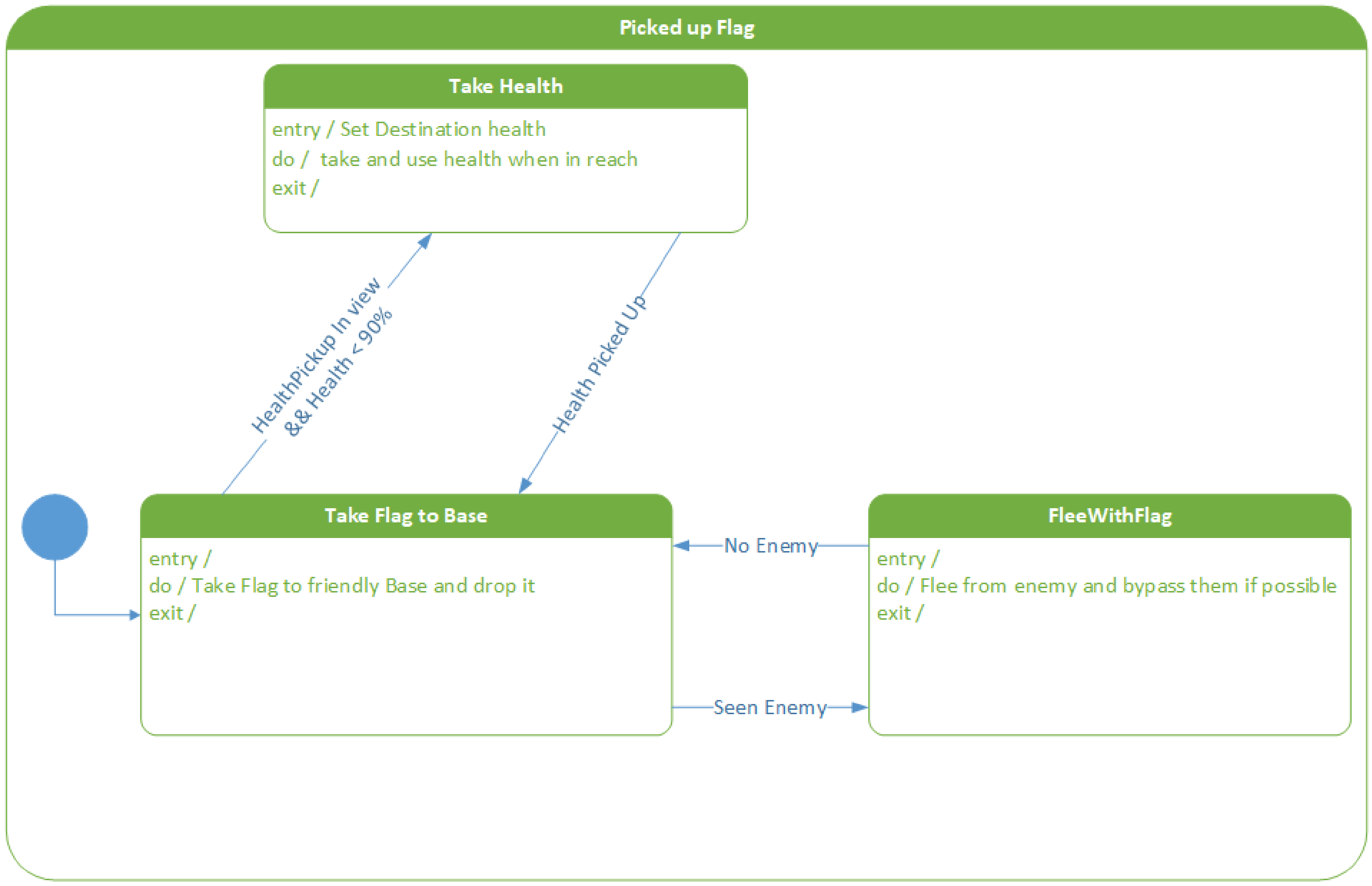
Defend State Machine



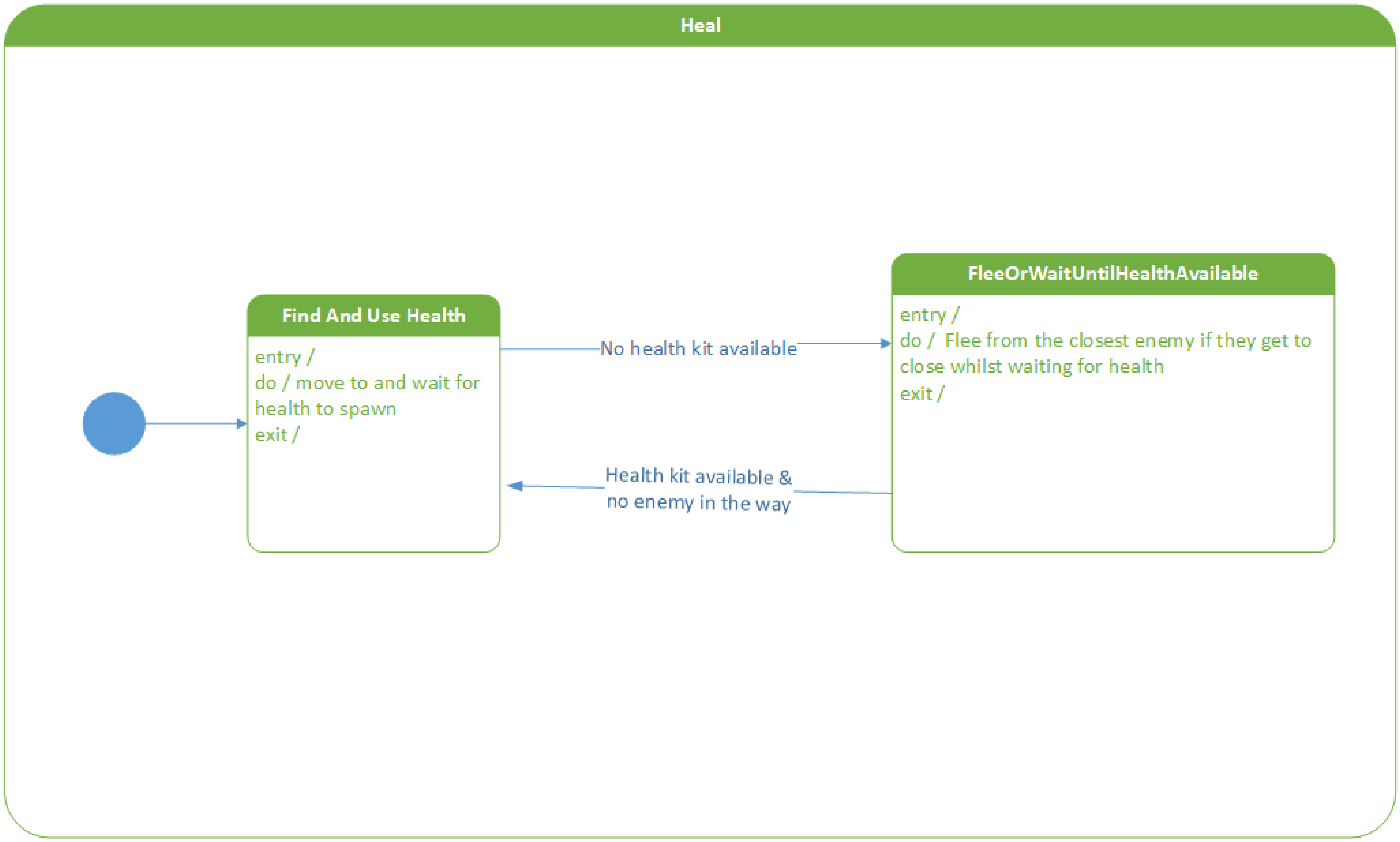
Combat State Machine


Combat
FightClosestEnemy
entry /
do / Move to and attack enemy when in range
exit /
Retreat
entry /
do / Flee From Enemies or Return to Base
exit /
RegroupWithFriendly
entry /
do / Regroup before attacking
exit /
Outnumbered
Not Outnumbered && Close to friendly || HavetoFight
Not Close To Friendly
Regrouped With Ally
Not close to friendly

Picked Up Flag State Machine


Picked up Flag
Take Health
entry / Set Destination health
do / take and use health when in reach
exit /
HealthPickup In view && Health < 90%
Health Picked Up
Take Flag to Base
entry /
do / Take Flag to friendly Base and drop it
exit /
No Enemy
Seen Enemy
FleeWithFlag
entry /
do / Flee from enemy and bypass them if possible
exit /

Heal State Machine


Heal
Find And Use Health
entry /
do / move to and wait for health to spawn
exit /
No health kit available
FleeOrWaitUntilHealthAvailable
entry /
do / Flee from the closest enemy if they get to close whilst waiting for health
exit /
Health kit available & no enemy in the way

ADDITIONAL HELPERS

To assist the Ais decision making I added some additional bools and functions to the pre-implemented API which are listed below.

Agent Data

Enemy Flag Taken By Friendly

Returns if the Enemy Flag is in a friendly agent's inventory

Enemy Flag Taken By Enemy

Returns if the Enemy Flag is in an Enemy agent's inventory

Friendly Flag Taken By Friendly

Returns if the Friendly Flag is in a Friendly agent's inventory

Friendly Flag Taken By Enemy

Returns if the Friendly Flag is in an Enemy agent's inventory

Enemy Flag Scored

Returns if the Enemy Flag is scored (dropped) in the friendly base

Friendly Flag Scored By Enemy

Returns If the friendly flag is scored in the enemy base

Friendly Flag In Base

Returns if the friendly flag is dropped in the friendly base

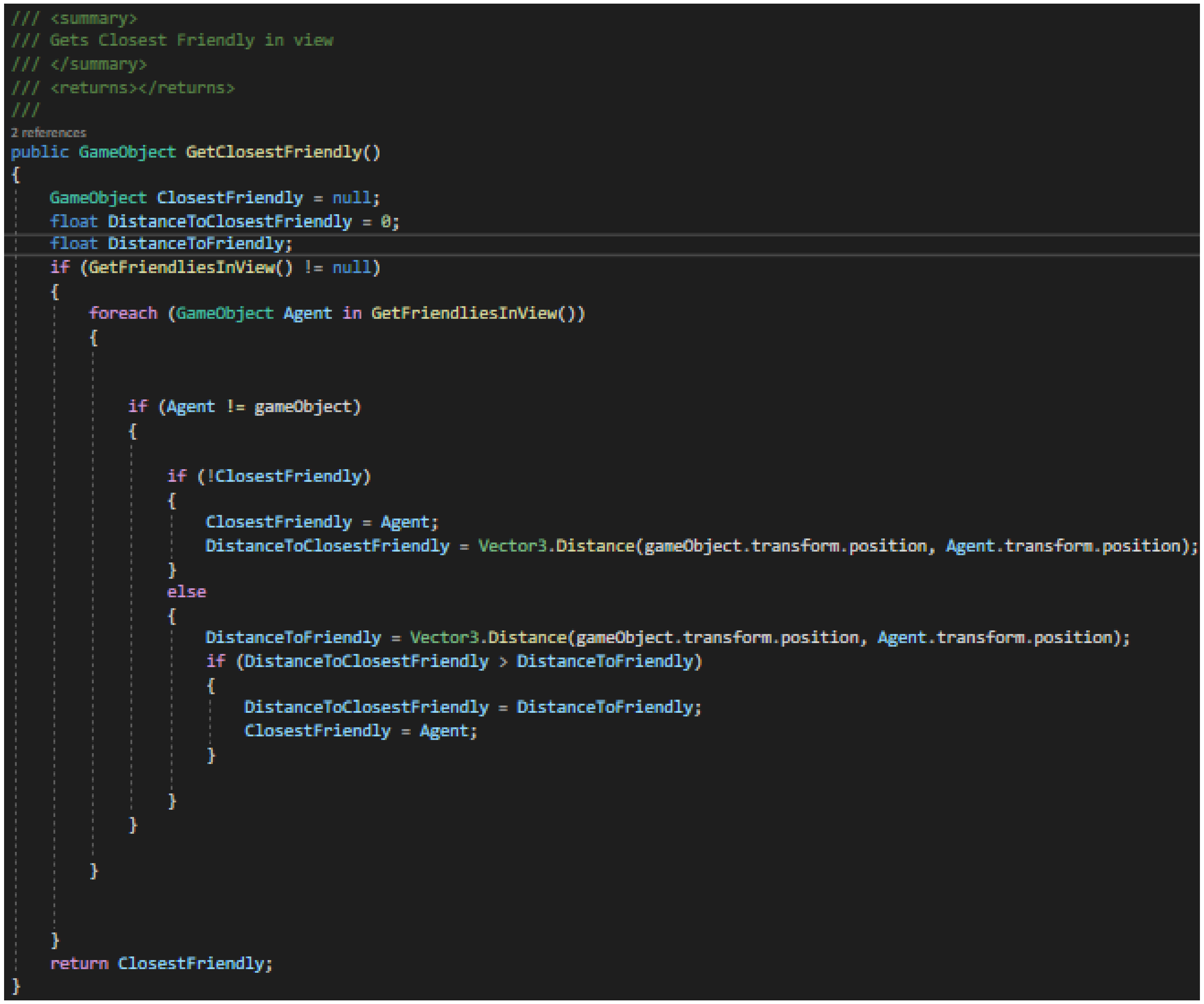
Closer To Friendly Flag

Returns if the agent is closer to the friendly flag than the enemy one. (Used in the transition to defend from attack)

Agent Sensing

Get Closest Friendly

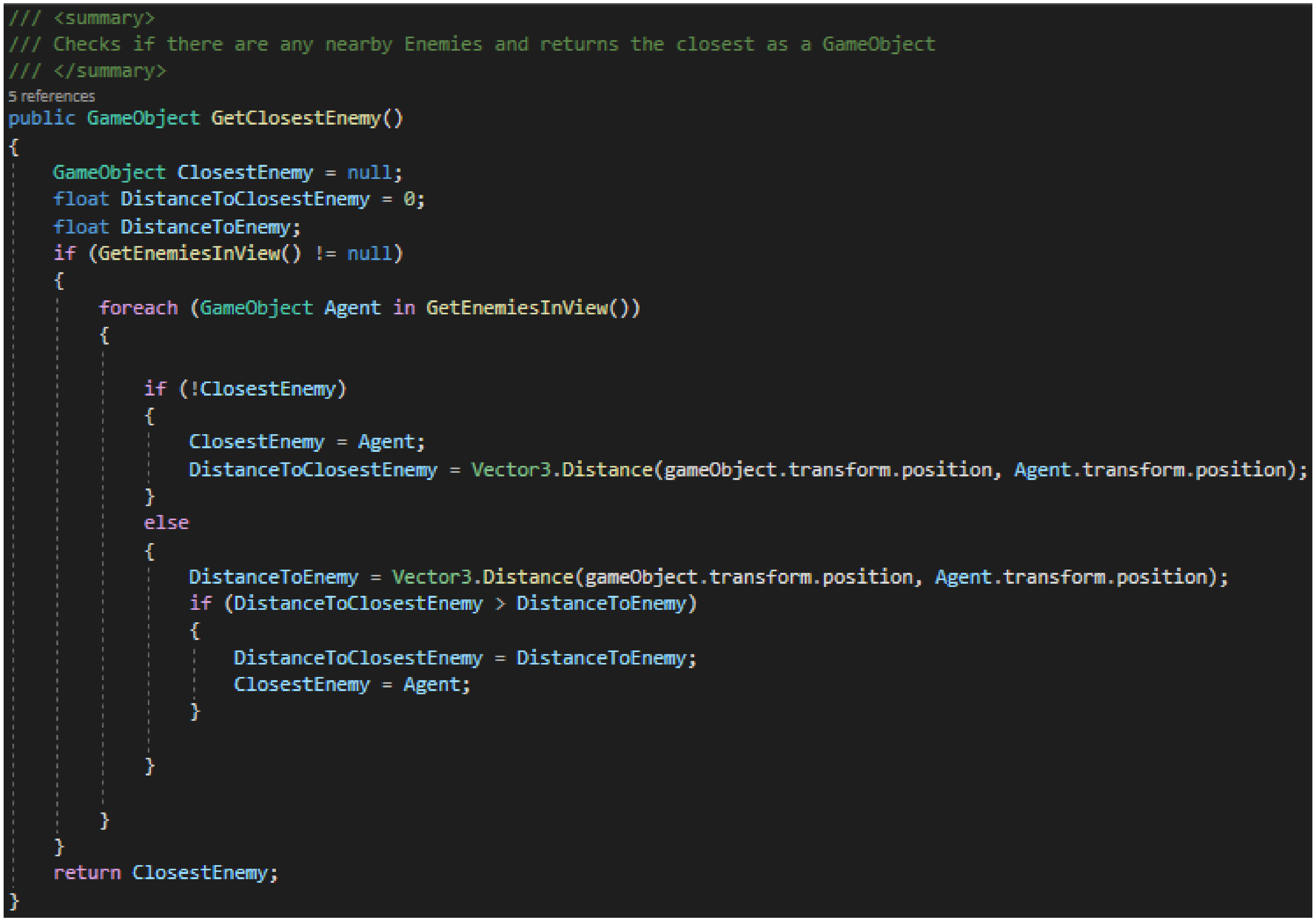
Returns the closest friendly in view as GameObject



```
/// <summary>
/// Gets Closest Friendly in view
/// </summary>
/// <returns></returns>
///
2 references
public GameObject GetClosestFriendly()
{
    GameObject ClosestFriendly = null;
    float DistanceToClosestFriendly = 0;
    float DistanceToFriendly;
    if (GetFriendliesInView() != null)
    {
        foreach (GameObject Agent in GetFriendliesInView())
        {


            if (Agent != gameObject)
            {

                if (!ClosestFriendly)
                {
                    ClosestFriendly = Agent;
                    DistanceToClosestFriendly = Vector3.Distance(gameObject.transform.position, Agent.transform.position);
                }
                else
                {
                    DistanceToFriendly = Vector3.Distance(gameObject.transform.position, Agent.transform.position);
                    if (DistanceToClosestFriendly > DistanceToFriendly)
                    {
                        DistanceToClosestFriendly = DistanceToFriendly;
                        ClosestFriendly = Agent;
                    }

                }
            }

        }


    }
    return ClosestFriendly;
}
```

Get Closest Enemy

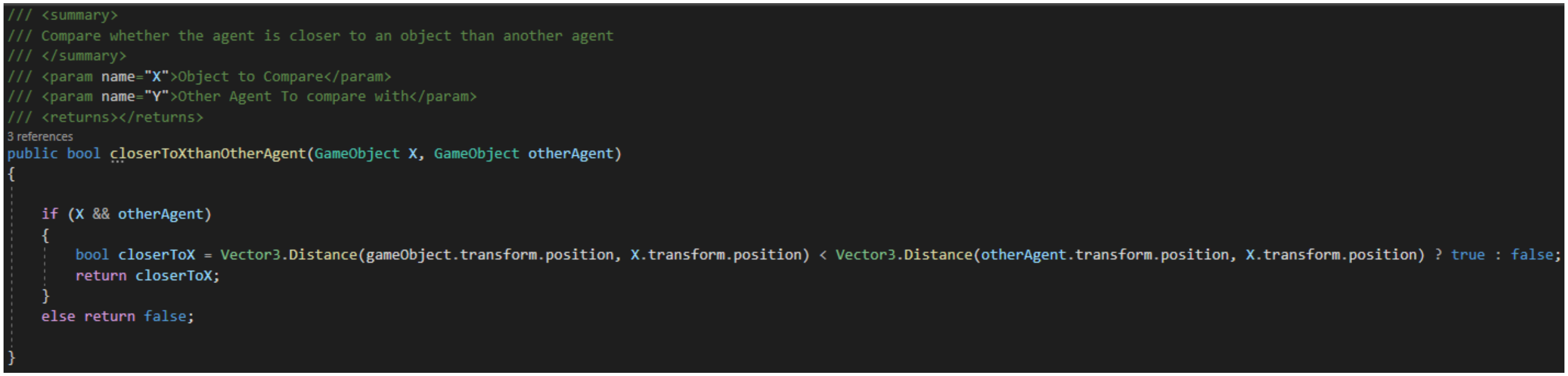
Returns the Closest Enemy in view as GameObject



```
/// <summary>
/// Checks if there are any nearby Enemies and returns the closest as a GameObject
/// </summary>
5 references
public GameObject GetClosestEnemy()
{
    GameObject ClosestEnemy = null;
    float DistanceToClosestEnemy = 0;
    float DistanceToEnemy;
    if (GetEnemiesInView() != null)
    {
        foreach (GameObject Agent in GetEnemiesInView())
        {

            if (!ClosestEnemy)
            {
                ClosestEnemy = Agent;
                DistanceToClosestEnemy = Vector3.Distance(gameObject.transform.position, Agent.transform.position);
            }
            else
            {
                DistanceToEnemy = Vector3.Distance(gameObject.transform.position, Agent.transform.position);
                if (DistanceToClosestEnemy > DistanceToEnemy)
                {
                    DistanceToClosestEnemy = DistanceToEnemy;
                    ClosestEnemy = Agent;
                }

            }

        }
    }
    return ClosestEnemy;
}
```

Closer To X Than Other Agent

Compares the Distance between the agent, the otherAgent and the specified object “X” and returns whether the agent is closer than otherAgent

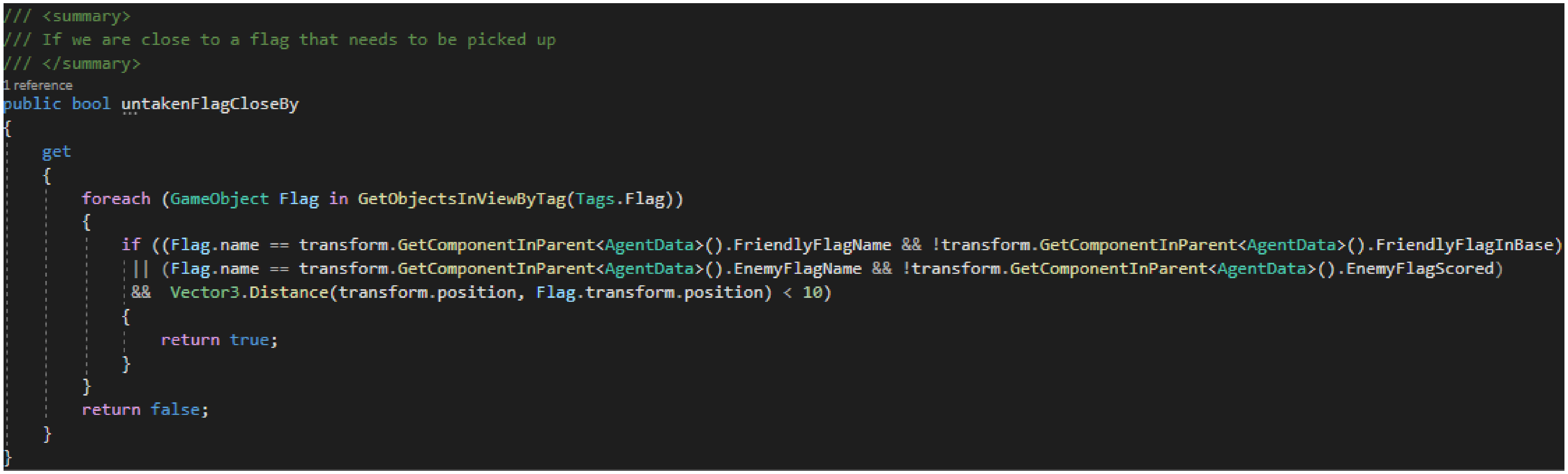


```
/// <summary>
/// Compare whether the agent is closer to an object than another agent
/// </summary>
/// <param name="X">Object to Compare</param>
/// <param name="Y">Other Agent To compare with</param>
/// <returns></returns>
3 references
public bool closerToXthanOtherAgent(GameObject X, GameObject otherAgent)
{

    if (X && otherAgent)
    {
        bool closerToX = Vector3.Distance(gameObject.transform.position, X.transform.position) < Vector3.Distance(otherAgent.transform.position, X.transform.position) ? true : false;
        return closerToX;
    }
    else return false;

}
```

Untaken Flag Closeby

Returns if the agent is close to a flag that need picking up

First Checks the flag by name and if it is taken, then checks if the flag is within 10 meters of the agent

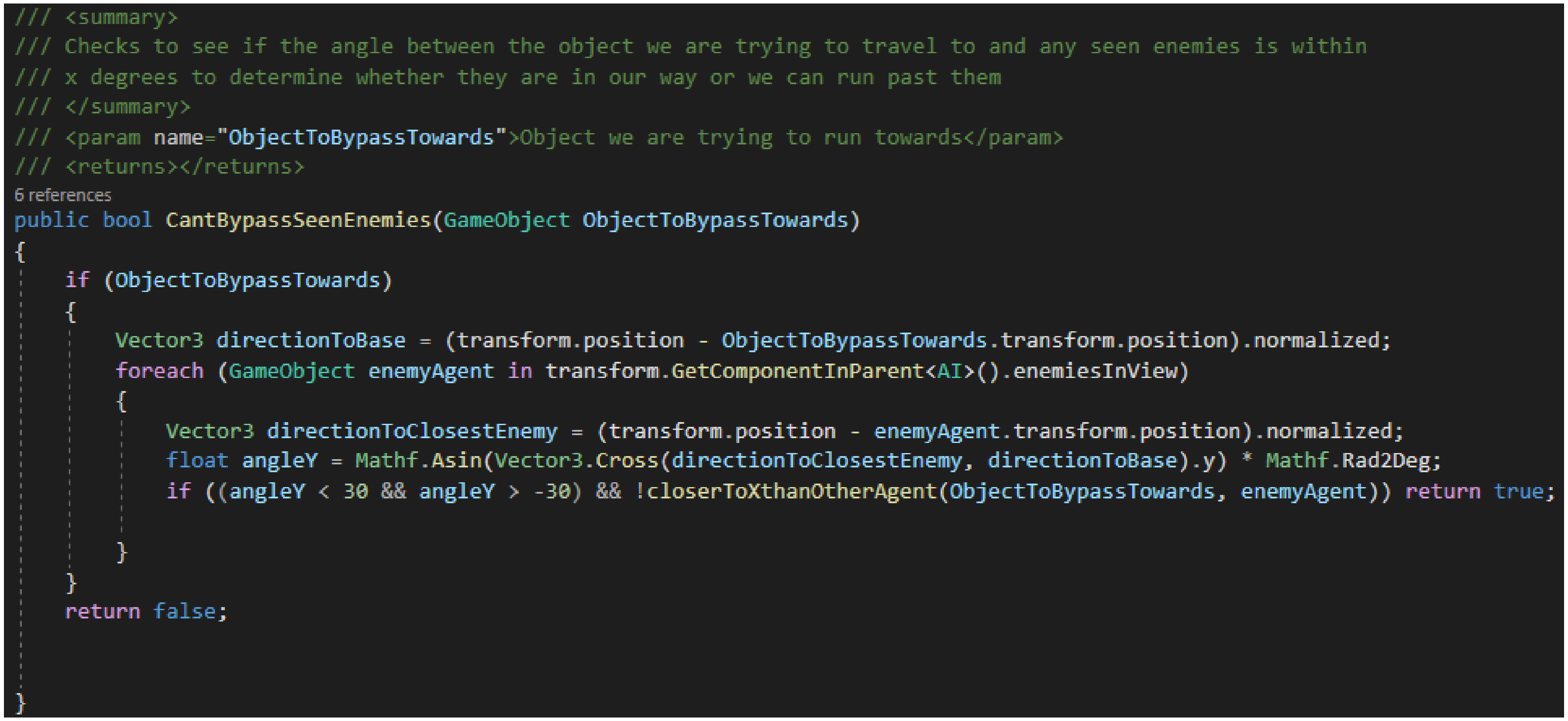


```
/// <summary>
/// If we are close to a flag that needs to be picked up
/// </summary>
1 reference
public bool untakenFlagCloseBy
{
    get
    {
        foreach (GameObject Flag in GetObjectsInViewByTag(Tags.Flag))
        {
            if ((Flag.name == transform.GetComponentInParent<AgentData>().FriendlyFlagName && !transform.GetComponentInParent<AgentData>().FriendlyFlagInBase)
             || (Flag.name == transform.GetComponentInParent<AgentData>().EnemyFlagName && !transform.GetComponentInParent<AgentData>().EnemyFlagScored)
             &&  Vector3.Distance(transform.position, Flag.transform.position) < 10)
            {
                return true;
            }
        }
        return false;
    }
}
```

Cant Bypass Seen Enemies

I am responsible for the original design of this function however mathematical calculation was externally sourced. The function takes an object and calculates the agent's angle toward it, then it compared that angle to the angle of every seen enemy and if the two angles are within 30 degrees of each other then the function returns false.

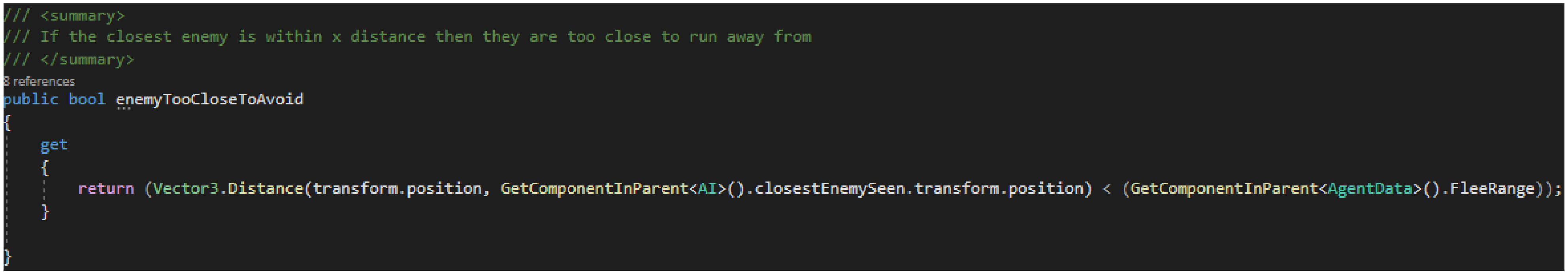
The closer to x than other enemy check is also done to see that the enemy is not closer to the desired object than the agent in case they we within 30 degrees but behind the agent, not in front of him.



```
/// <summary>
/// Checks to see if the angle between the object we are trying to travel to and any seen enemies is within
/// x degrees to determine whether they are in our way or we can run past them
/// </summary>
/// <param name="ObjectToBypassTowards">Object we are trying to run towards</param>
/// <returns></returns>
6 references
public bool CantBypassSeenEnemies(GameObject ObjectToBypassTowards)
{
    if (ObjectToBypassTowards)
    {
        Vector3 directionToBase = (transform.position - ObjectToBypassTowards.transform.position).normalized;
        foreach (GameObject enemyAgent in transform.GetComponentInParent<AI>().enemiesInView)
        {
            Vector3 directionToClosestEnemy = (transform.position - enemyAgent.transform.position).normalized;
            float angleY = Mathf.Asin(Vector3.Cross(directionToClosestEnemy, directionToBase).y) * Mathf.Rad2Deg;
            if ((angleY < 30 && angleY > -30) && !closerToXthanOtherAgent(ObjectToBypassTowards, enemyAgent)) return true;

        }
    }
    return false;


}
```

Enemy Too Close Too Avoid

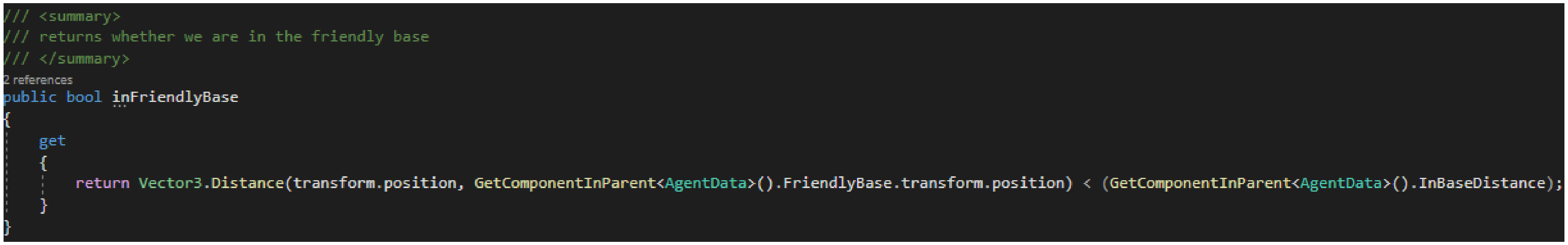
Returns if the distance to the closest enemy is less than flee range



```
/// <summary>
/// If the closest enemy is within x distance then they are too close to run away from
/// </summary>
8 references
public bool enemyTooCloseToAvoid
{
    get
    {
        return (Vector3.Distance(transform.position, GetComponentInParent<AI>().closestEnemySeen.transform.position) < (GetComponentInParent<AgentData>().FleeRange));
    }

}
```

In Friendly Base

Returns if the agent is inside the friendly base

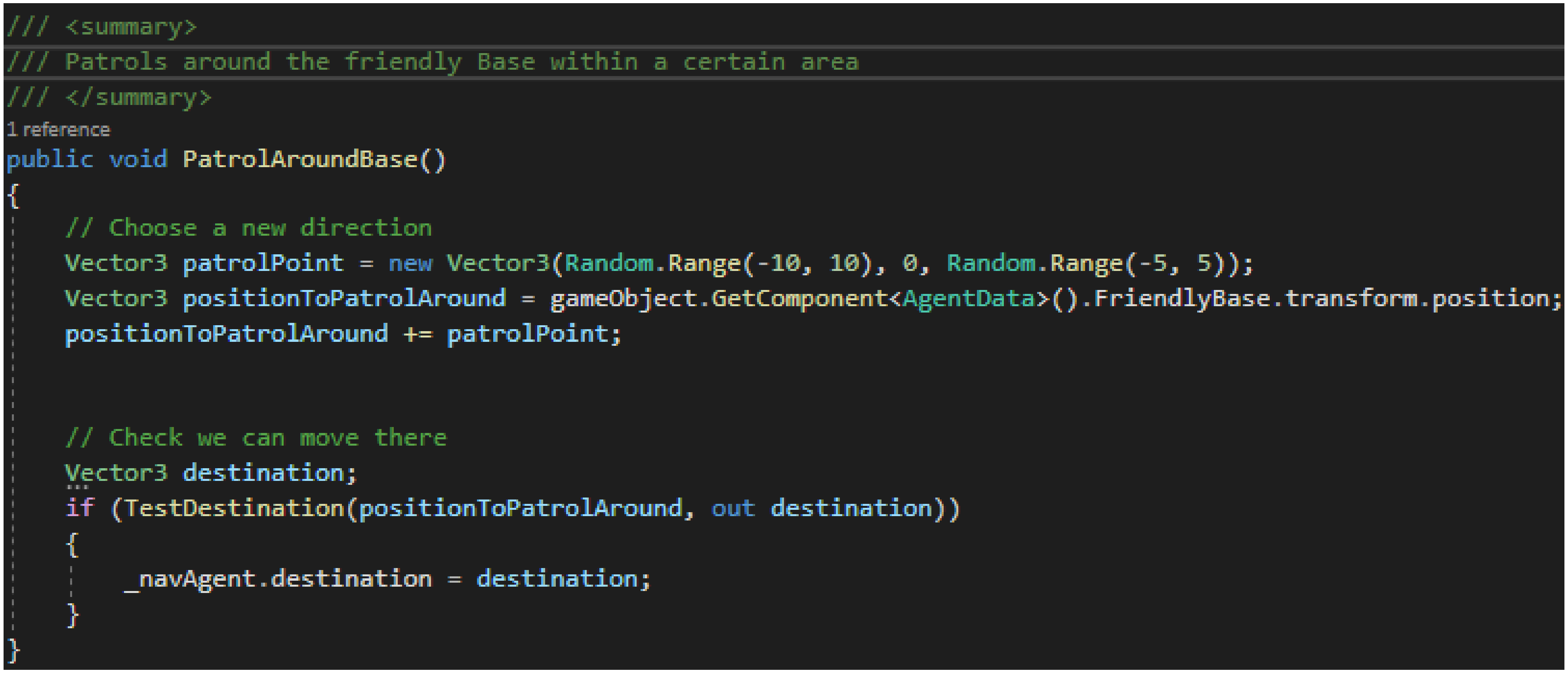


```
/// <summary>
/// returns whether we are in the friendly base
/// </summary>
2 references
public bool inFriendlyBase
{
    get
    {
        return Vector3.Distance(transform.position, GetComponentInParent<AgentData>().FriendlyBase.transform.position) < (GetComponentInParent<AgentData>().InBaseDistance);
    }
}
```

Agent Actions

Patrol Around Base

Choose a random point near the base and test it, if it is on the navmesh then move there



```
/// <summary>
/// Patrols around the friendly Base within a certain area
/// </summary>
1 reference
public void PatrolAroundBase()
{
    // Choose a new direction
    Vector3 patrolPoint = new Vector3(Random.Range(-10, 10), 0, Random.Range(-5, 5));
    Vector3 positionToPatrolAround = gameObject.GetComponent<AgentData>().FriendlyBase.transform.position;
    positionToPatrolAround += patrolPoint;


    // Check we can move there
    Vector3 destination;
    if (TestDestination(positionToPatrolAround, out destination))
    {
        _navAgent.destination = destination;
    }
}
```

Testing Plan

A black Box test plan was employed for this project. Each Test observed both the action and transitions of each state through manipulating the project at run time to forcefully trigger the transitions to them from other states

TEST DATA

ID	Owning State Machine	State	Test	Expected Result	Actual Result	Action
1	Outer	Attack State	Positioned Friendly Flag in base and enemy flag out of the base Then observed that Agents transitioned to attack appropriately after combat, defend or heal during gameplay	When the agent's friendly flag is in the base but the enemy flag is not agents should transition to the attack state from the other states appropriately	All transitions to attack work as expected	
2	Attack	Inner Pickup Enemy Flag	Added debug log to check it is the first state entered by Attack and moved the enemy flag around to make sure the agent is always moving toward it Checked Transitions by killing the friendly flag carrier whilst they are protecting him and placing the agent near the powerup to make sure they go back to pick up after using a power-up	The first state entered by attack, only exited if near power-up or ally picks up the flag When in the state the agent moves to the location of the enemy flag wherever it is Upon reaching enemy flag the agent picks it up and changes to picked up flag outer state	Works as expected	
3	Attack	Protect Ally with Enemy Flag	Positioned the agent near friendly who is carrying the flag to check they transition and follow correctly	Agent transitions when ally picks up the enemy flag Follows ally with the flag back to the base	Works as expected	

4	Attack	Pickup PowerUp Atk	Placed the enemy near the powerup whilst they are in the pickup enemy flag state	When the agent can see the powerup and they are in the pickup enemy flag state they should walk to use the powerup before transitioning back to pick up the enemy flag	After using the powerup transitions instead to pick up the friendly flag and walks back to the friendly base (Appendix B)	Fixed the changestate parameter from Pickup Friendly Flag to Pick up Enemy Flag
4.1 (Retest)	Attack	Pickup PowerUp Atk			Works as originally expected	
5	Outer	Defend	Positioned the friendly flag outside of the friendly base Forced agent into attack state and moved them near the friendly flag to make sure they transition	Agent should transition to defend friendly flag if they are closer to it, when taken, than the enemy flag or when the enemy flag is scored.	Works as expected	
6	Defend	Defend Base	Check it is the first state entered when in defend Make sure that they transition to it by placing both flags in friendly base Observed them patrolling	When both flags are in friendly base the agent should patrol the friendly base transition for combat or lost flags accordingly.	Transitions to Pickup Friendly Flag, combat, and attack As Intended Agents continue to patrol when they have scored enemy flag an agents patrol in a horizontal line and not in an area. (Appendix C)	Error in condition for hunt enemy flag carrier caused agents to check if friendlies had the friendly flag not enemies. Changed to check for enemy taking friendly flag instead Error in patrol base function causing random position to only be created on X axis inside the base. Changed other Random Range to z axis instead of y.

6.1 **(Retest)**	Defend	Defend Base			Transition to Hunt enemy flag carrier now works and agents now patrol in an area, not a horizontal line.	
7	Defend	Protect Ally with Friendly Flag	Positioned the agent near friendly who is carrying the flag to check they transition and follow correctly	If in the defend state and ally picks up friendly flag then transition to this state Follow the ally with the friendly flag until they reach the base or agent sees an enemy	Works as expected	
8	Defend	Hunt Enemy Flag Carrier	Placed both flags in friendly base to force defend state Forced enemy to pick up friendly flag Observed with debug logs if friendly bots then transitioned to hunt	If and enemy has picked up the friendly flag, then agent follows them causing a combat transition when they see them.	Works as intended	
9	Defend	Pickup Friendly Flag	Placed the friendly flag outside of the friendly base	Agent moves to the friendly flags position then picks it up	Error in base conditions causes friendly flag to not be registered after being dropped Agents then crowd round flag picking up and dropping it endlessly (Appendix D)	Forgot to drag friendly flag into the inspector parameter for each base causing the Boolean check “friendly flag in base” To return no reference Dragged each bases friendly flag in repsectivley

9.1 (retest)	Defend	Pickup Friendly Flag			Works as originally intended	
10	Defend	Pickup PowerUp Dfd	Placed the enemy near the powerup whilst they are in the pickup Friendly flag state	Agent should move to and pickup the power up whilst in the pickup friendly flag state Once they have used to powerup or it is gone they should transition back to pickup friendly flag	Works as intended	
11	Outer	Combat	Check transitions from attack and defend by placing an enemy infront of and further away from a flag they are trying to reach to make sure they fight or choose to steal the flag accordingly Checked transition from heal and picked up flag by placing enemy within the “tooclosetoretreat” distance whilst the ai is in one of those two states	Agent should transition to combat from attack state if they see an enemy that is closer to them than Enemy. Transition from Defend should happen when an enemy is seen. Transition from pickup flag and heal should only happen if the enemy is too close Agent should transition out of combat if they need to heal or have the flag and the enemy become far enough to flee from	Transitions from attack and defend work as intended Agent does not transition to heal when exiting combat (Appendix E) Agent does not transition back to picked up flag after combat if they have the flag (Appendix F)	Re-shuffled the order of condition checks inside the combat state so that pickup flag and heal are checked first Problem was that attack or defend are always true and were being checked first, not giving pickup flag or heal a chance to be checked Moved heal and pickup flag checks before attack and defend.
11.1 (retest)	Outer	Combat			All transitions now work as intended	
12	Combat	Fight Closest Enemy	Checked it is the first state entered by placing a debug log in its enter function	Agent should pick the closest enemy to their position and move in range to attack them	Works as intended when dueling	Removed the transition from regroup back to retreat.

			Place numerous enemies around it and observe if it attacks the closest or transitions to retreat or regroup dependent on conditions.	Agent should transition to retreat if outnumbered Or Regroup if too far from friendlies	Conditions loop every tick between retreat and regroup when Team fighting whilst outnumbered generating thousands of debug calls for entering the two states.	Retreat can still transition to regroup however regroup can now only transition to fight closest enemy which then calls the original check for retreating.
12.1 (retest)	Combat	Fight Closest Enemy		Agents should now transition between retreat and regroup when needed without generating thousands of transitions per second	Agents now transition as intended. Agents in team fights regroup and now retreat together until they can fight evenly or are caught	
13	Combat	Retreat	Place 2 red and 1 blue agent in a side lane to see if the blue will retreat to the base when outnumbered. Repeated with 2 reds and 2 blues. Check if agent flees from enemy if they are blocking his route to the base	When agent can see greater enemies than friendlies it turns and retreats to its base. If the route to base is blocked it flees from the closest enemy		
14	Combat	Regroup with Friendly	Place 2 grouped reds and 2 separated blues in the side lane to check if the blues will walk to each other before moving towards the reds	Blue agents should regroup before heading towards the red agents. Red agents should keep their cohesion and regroup if they drift too far apart	Both sets of agents instantly move towards each other and do not then engage the enemy once regrouped (Appendix G)	Increased regroup distance variable to 5 from 2 as agent were colliders were stopping them from getting close enough to cause the transition

						back to fight closest enemy
14.1 (Retest)	Combat	Regroup with Friendly			Regroup now works as intended agents regroup when fight as one	
15	Outer	Picked up Flag	Force and Ai to pick up friendly and enemy flags to make sure transition works	When picking up one of the two flags the agent should transition into picked up flag	Works as intended	
16	Picked up Flag	Take Flag to Base	Check this is the first state entered with a debug log. Make sure it paths the ai agent back towards the friendly base by giving it an unobstructed path	Agent should path directly back to the friendly base with whatever flag it has and drop that flag when getting into the base.	Works as intended	
17	Picked up Flag	Flee with Flag	Place Enemies in flag carriers' sight but not blocking his path to make sure transition does not travel unnecessary	Should be entered if a seen enemy starts blocking the agent's path towards the friendly flag and exited if they are no longer blocking it. Causes the flag carrier to flee directly from the closest enemy	Transition works both when enemy is blocking the agent's path but also when the enemy is directly behind the enemy (Appendix H)	Added check to bypass function to also check that the blocking enemy is closer to the friendly base than the agent. (the enemy agent is in front not behind)
17.1	Picked up Flag	Flee With Flag			Now works as expected	
18	Picked up Flag	Take Health	Placed the enemy near the Healthkit whilst they are in the Take Flag to base state and their health is below 70%	Should be entered if the flag carrier's health is below 70% and they can see the health pickup they should pick it up and use it	Agent picks up health but does not use it.	Added a line making the agent use the health after picking it up Added an additional check to the transition that the agent must

					Secondly, a design over-sight means that flag carrier will travel away from the friendly base with the flag to pick up health if they can see it. (Appendix I)	be further from the friendly base than he is from the health kit to retrieve it
18.1 (Retest)	Picked Up Flag	Take Health			Now works as intended	
19	Outer	Heal	Placed Ai in a duel to see if when the victor won with less than 30% health, he would transition to heal Repeated the duel over the Red flag until the blue agent won with less than 30% health	Agent should transition to heal after winning a fight with less than 30% health If fighting near the flag the agent should override heal to pick the flag up and then transition to heal after returning the flag to the base	Agent transitions to heal after combat but the range at which he will stay if there is a flag to be picked up is too small (Appendix J)	Increased heal override range
20	Heal	Find and Use Health	Moved the health spawners location to make sure agents always moved to its position. Make sure transition from flee works properly by using debug logs in the enter function. Move enemies within and out of sight whilst waiting for health to spawn to test transition from flee	Agent should move towards the health spawners location, pickup and use health if available. Otherwise, they should wait at the health spawner	Works as intended however agents flee too early when seeing an enemy whilst waiting for health. (Appendix K)	Added Distance check on closest enemy to only allow agent to flee from health pickup if the enemies are within a certain distance.
21	Heal	Flee or Wait Until	Moved Enemy agents within sight whilst an agent is waiting	Observe the agent continuing to the health if they see an	Works as intended.	

		Health Available	to use health or when agent is travelling to it	enemy but it is not blocking their path Observe the agent fleeing when an enemy is blocking their path		

Critical Evaluation

Overall, the implementation of this algorithm has allowed for the scope and complexity of my design to be met with relative ease. The simplicity of State machines has allowed for more time working on the logic behind more intelligent and dynamic behaviours, rather than tweaking values like I may have had to with the two algorithms compared.

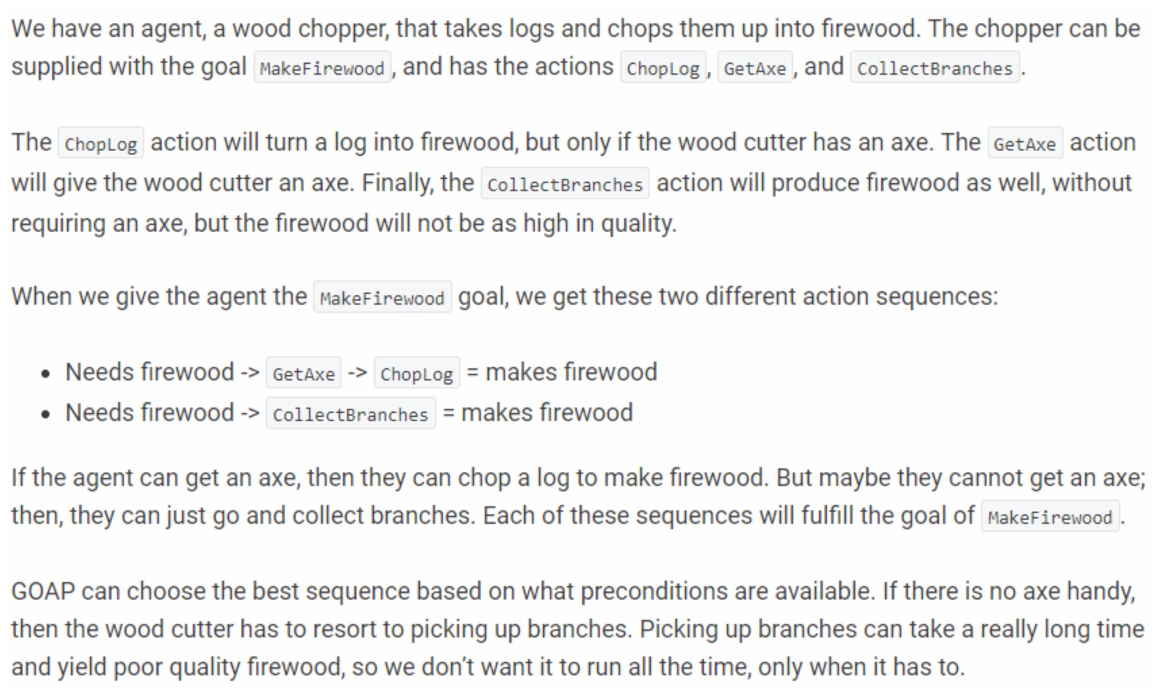
However, as pointed out in my original summary, a known drawback that I also experienced with this algorithm was its issue with scaling complexity and maintainability. The original design seen in this report is the fourth iteration of a base design, this was not due to me adding further features or extending it, but merely down to the need for it to be clear, understandable, and efficient. I also had to redesign many of the transitions between states, giving them more complex conditions so they did not occur at wrong or stupid times. This was not difficult in most cases but as pointed out by Millington and Funge, (2009), may have made the code appear as “ugly and unclear”. I would have preferred clean and perfectly concise code, however, due to the level of complexity I was aiming for, this become difficult to maintain. Overall, I believe a HSM was a good choice for the complexity of my current design as I still found it manageable and efficient to implement. On the other Hand, I do not believe it would be possible expand my design any further on an outer level, there may be space in some of the internal SMs though this would also quickly become stretched or unclear.

One thing I found difficult using this algorithm was managing the transitions for more complex behaviours such as flee, retreat, or regroup. The decisions on these behaviours from real players would be taken dependent on a number of different variables or circumstances, it was hard to adapt the transitions of the SM to show this. I settled on a simple system that has the ai check whether they are outnumbered or if they are close to their teammates in a fight to determine whether they should retreat or regroup. Fleeing takes place when an agent cannot bypass the enemies in its view on the way to its target destination. One improvement I would make to this system with more time in my schedule would be to implement a simple memory system for the agents, through adding seen friendlies and enemy to a list that would then persist on for x amount of time after they went out of view. This would make the retreat and regroup systems that technically work flawlessly even more impressive as agents would then still flee or chase each other around corners, something they currently struggle with.

In conclusion, I feel as though I chose the correct algorithm to accommodate the needs of this assignment. Whilst using something like GOAP may have given me a greater foundation to build a larger and more sophisticated system, I believe that the cons I identified previously, supported by the overall scope, time frame of the unit and my current skill as a non-software student made choosing a HSM far more feasible option for producing a finished product. I have learnt and picked up vast knowledge from this unit, so much so that were I to tackle it again, I may be inclined to use a more sophisticated algorithm, however, given the choices before me and the knowledge gained from my research, I feel I made the correct decision.

Appendices

Appendix A: Owens, (2014) basic GOAP example



We have an agent, a wood chopper, that takes logs and chops them up into firewood. The chopper can be supplied with the goal `MakeFirewood`, and has the actions `ChopLog`, `GetAxe`, and `CollectBranches`.

The `ChopLog` action will turn a log into firewood, but only if the wood cutter has an axe. The `GetAxe` action will give the wood cutter an axe. Finally, the `CollectBranches` action will produce firewood as well, without requiring an axe, but the firewood will not be as high in quality.

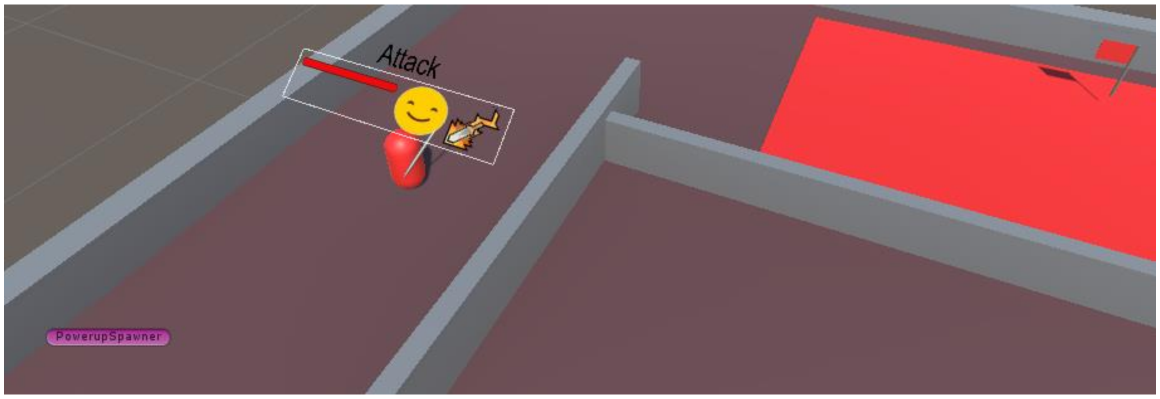
When we give the agent the `MakeFirewood` goal, we get these two different action sequences:

- Needs firewood -> `GetAxe` -> `ChopLog` = makes firewood
- Needs firewood -> `CollectBranches` = makes firewood

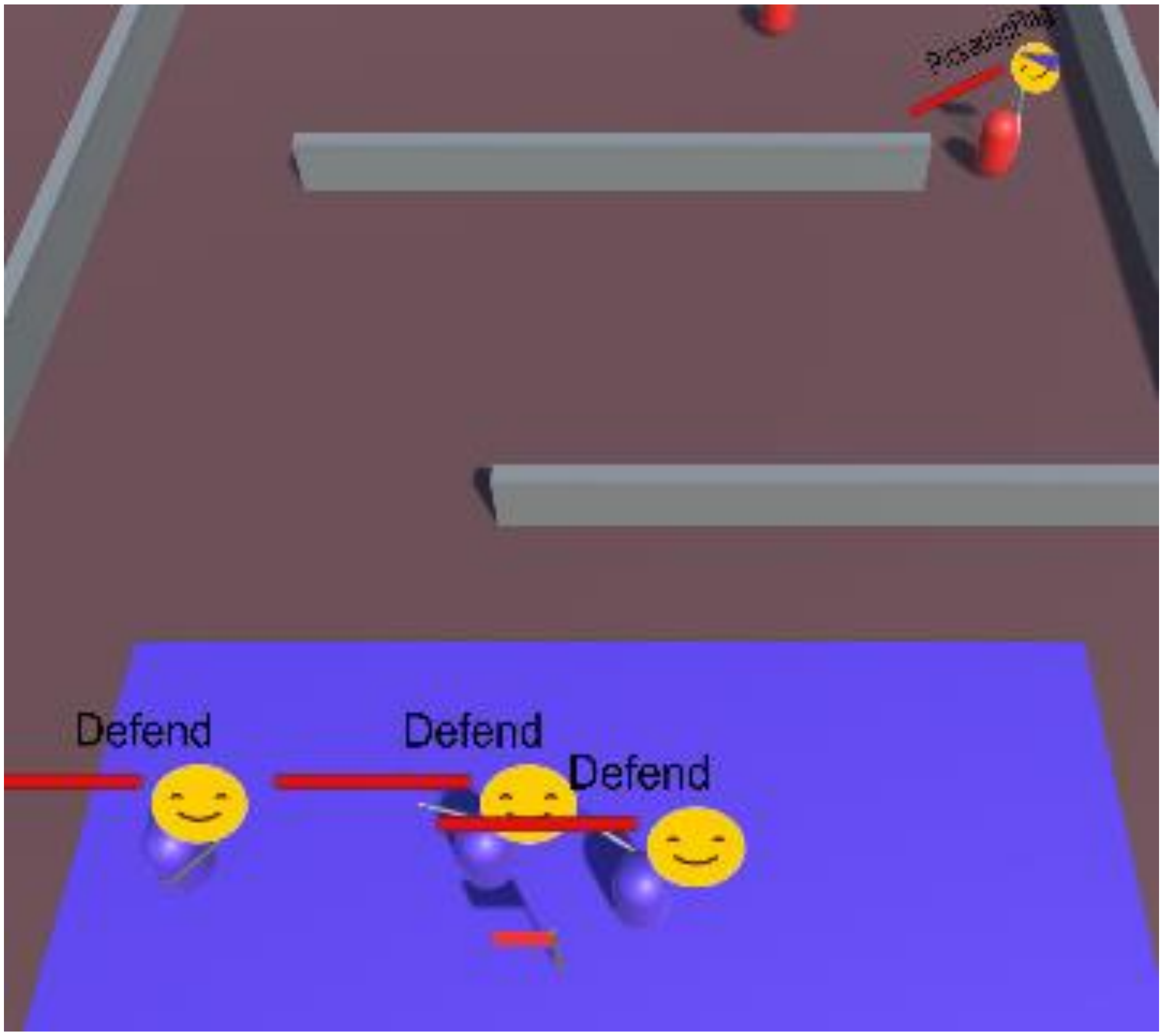
If the agent can get an axe, then they can chop a log to make firewood. But maybe they cannot get an axe; then, they can just go and collect branches. Each of these sequences will fulfill the goal of `MakeFirewood`.

GOAP can choose the best sequence based on what preconditions are available. If there is no axe handy, then the wood cutter has to resort to picking up branches. Picking up branches can take a really long time and yield poor quality firewood, so we don't want it to run all the time, only when it has to.

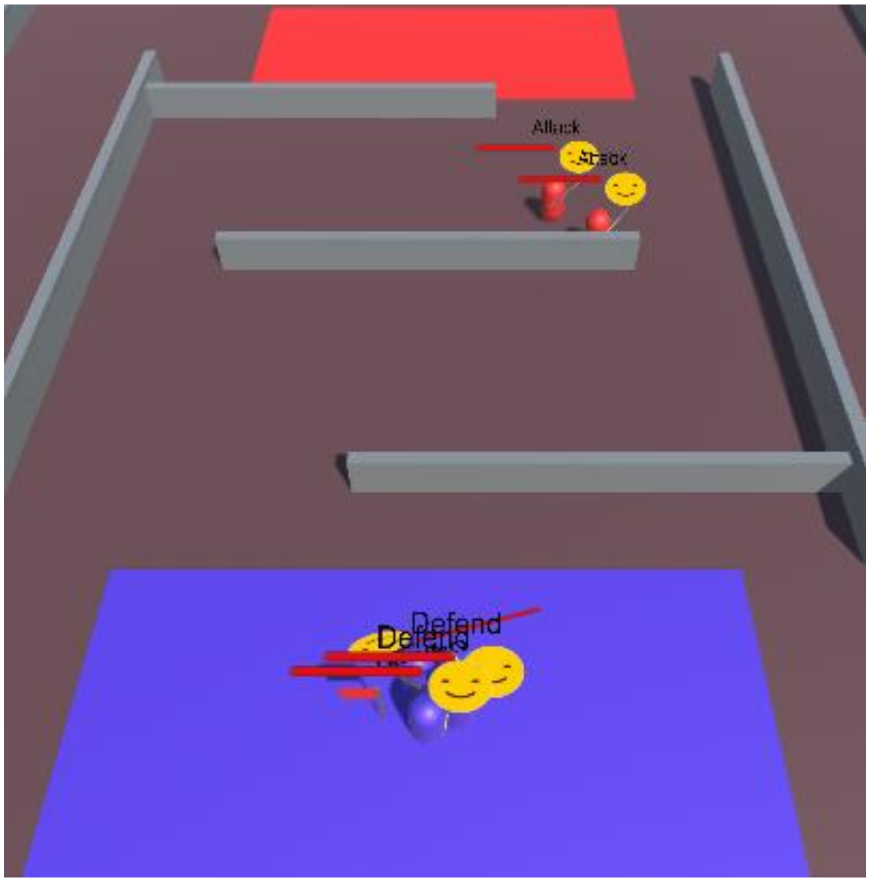
Appendix B: Pickup Power UP attack bug showing agent traveling back to friendly base in attack after picking up powerup



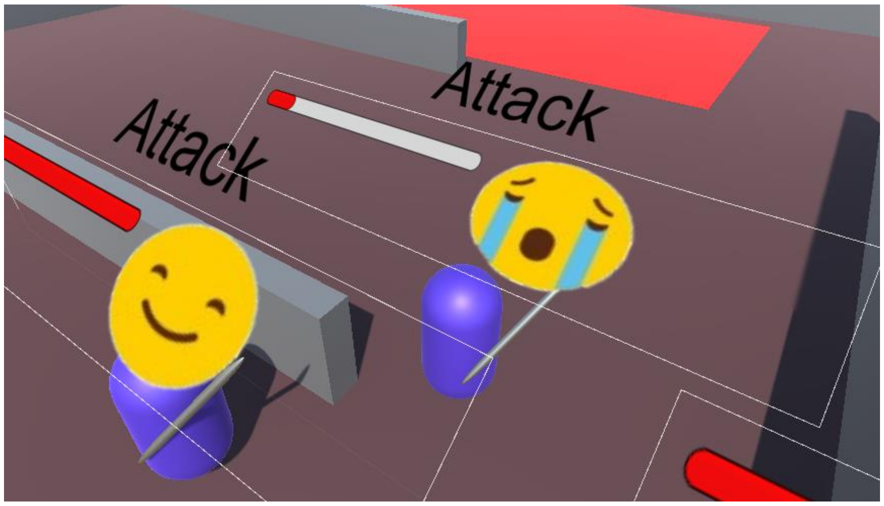
Appendix C: Agents Continuing to patrol in a horizontal line when friendly flag in enemy inventory



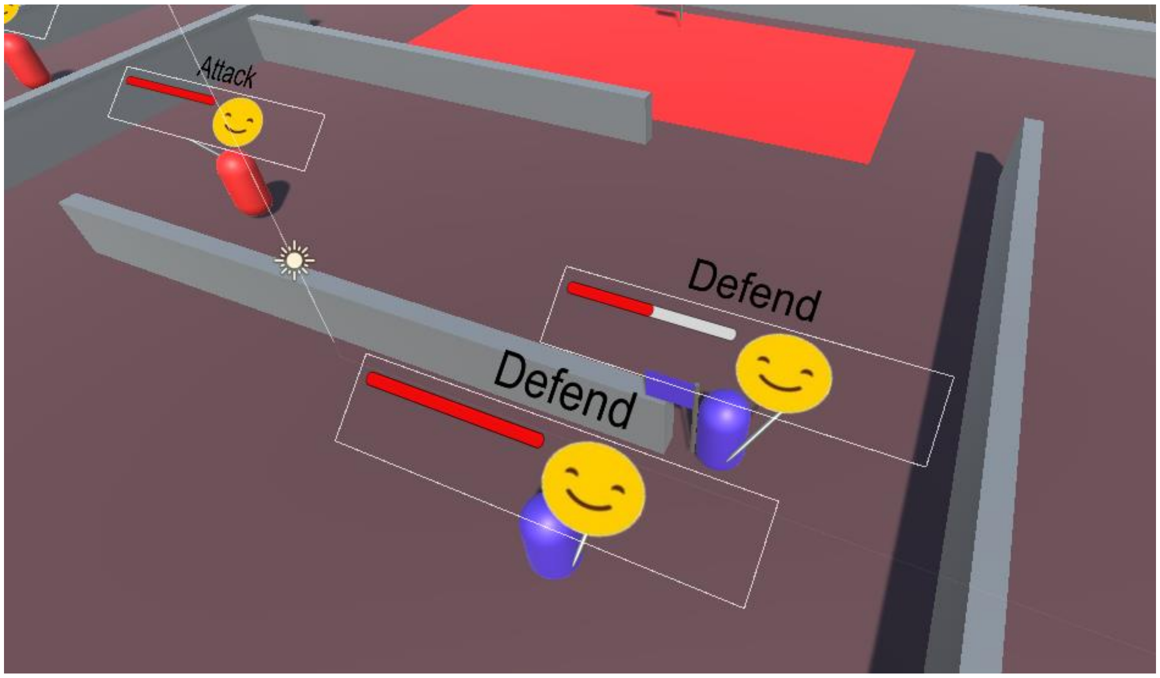
Appendix D: Agents Crowing around friendly Flag due to base error in returning if flag is in friendly base



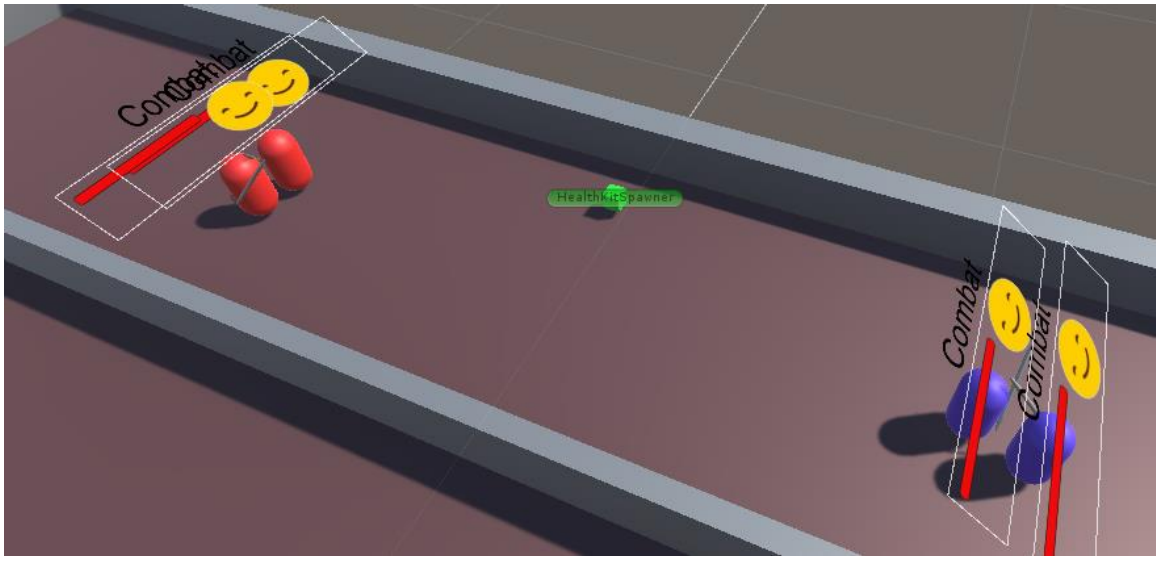
Appendix E: Agent Not entering heal after combat



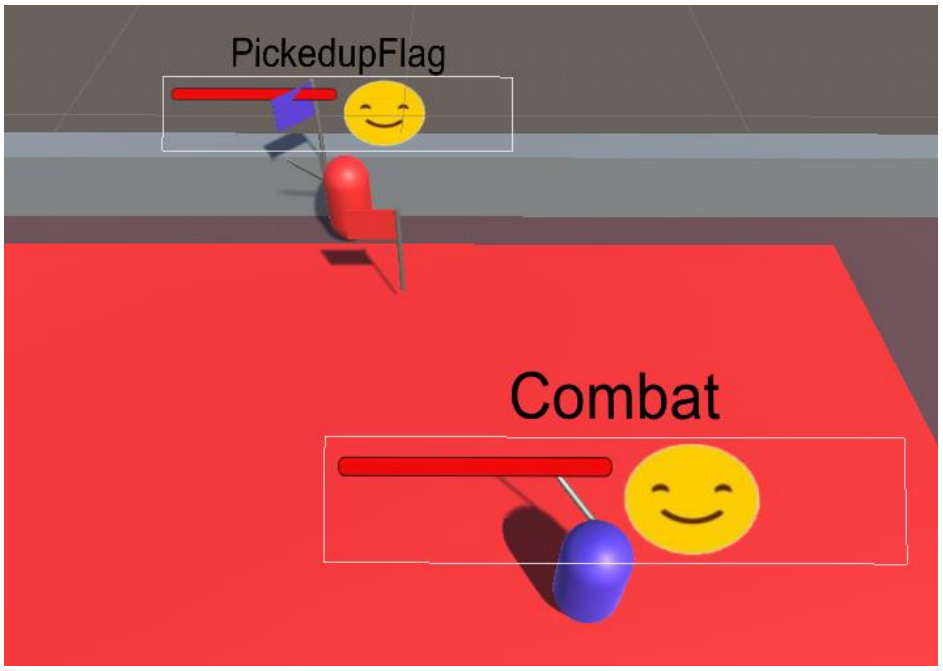
Appendix F: Agent not transitioning back to picked up flag after finshing combat



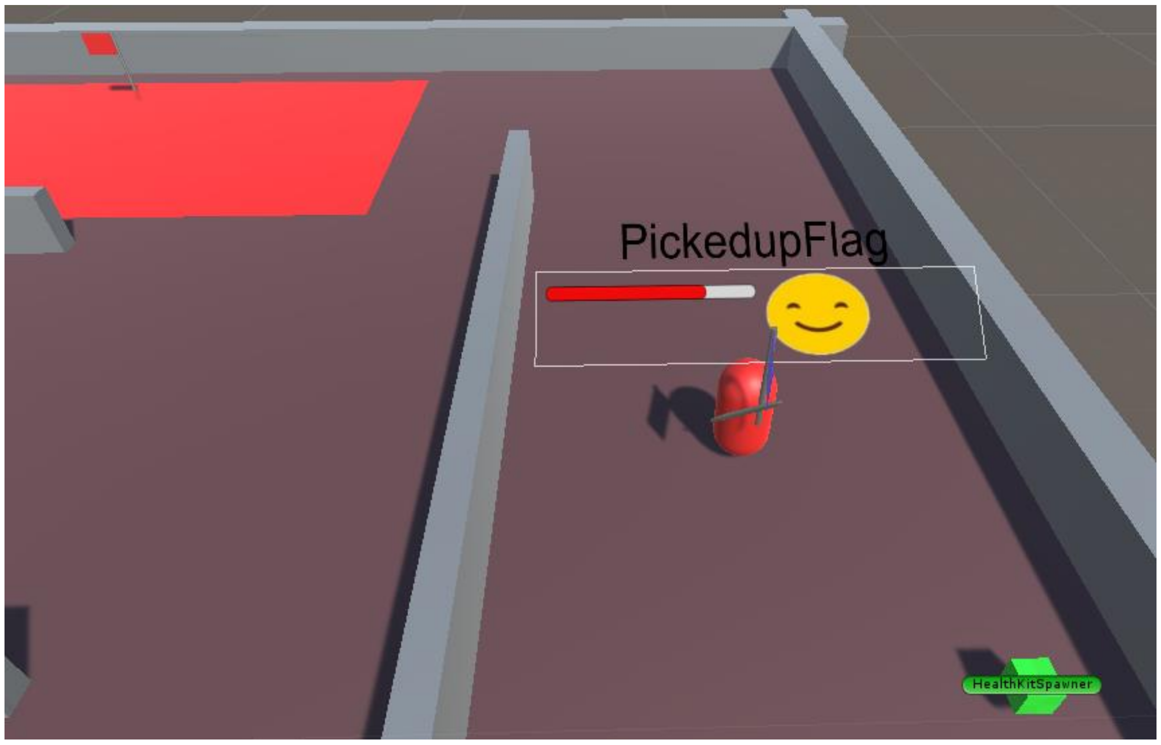
Appendix G: Agents constantly regrouping when placed in a 2v2 situation



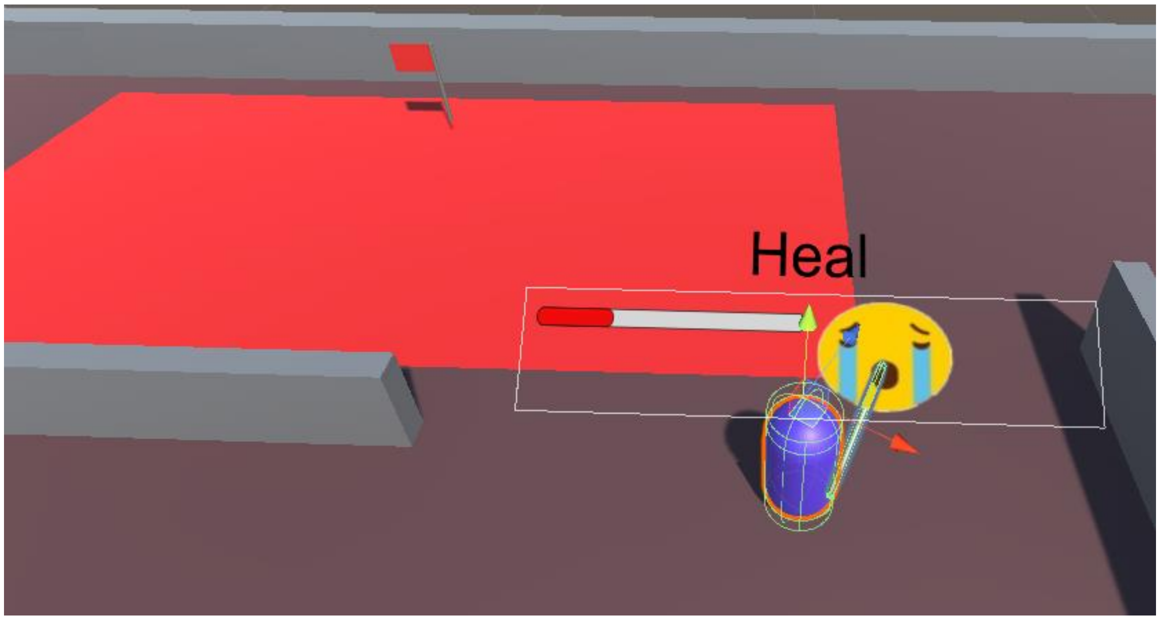
Appendix H: Agent With flag fleeing due to error resorting in enemy directly opposite of the angle to flag being considered as not bypassable (In Image Agent has fleed straight over the spot where he could have dropped the flag and then turned to fight)



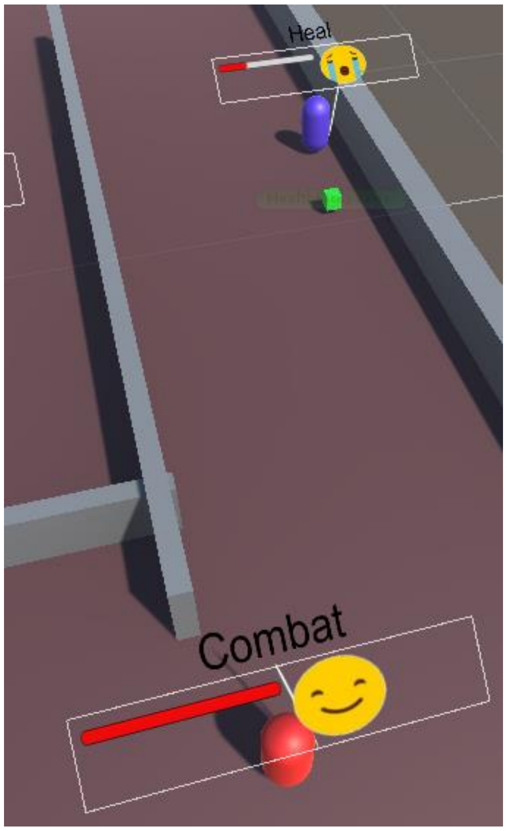
Appendix I: Error in the Take Health function of pickedupFlag causes agent to run away from the friendly base in order to pickup the health kit if their health is low



Appendix J: Agent transitioning to heal when close to enemy flag after teamfight due to range for “closetoflag” being too small



Appendix K: Agent fleeing too early when waiting for health after seeing enemy



Bibliography

Alur, R. (2003) ‘Formal Analysis of Hierarchical State Machines’, pp. 42–66. doi: 10.1007/978-3-540-39910-0_3.

Fernández, A. J. and Barragan, J. (2011) ‘Decision Tree-Based Algorithms for Implementing Bot AI in UT2004’, *Iwinac (1)*, 6686(September), pp. 372–382. doi: 10.1007/978-3-642-21344-1.

Long, E. of A. D. (2007) ‘Enhanced NPC Behaviour using Goal Oriented Action Planning’, (September), p. 110. Available at: http://www.edmundlong.com/Projects/Masters_EnhancedBehaviourGOAP_EddieLong.pdf.

Millington, I. and Funge, J. (2009) *Artificial Intelligence For Games (Second Edition)*. 2nd edn, *Morgan KaufMann*. 2nd edn. Morgan KaufMann. doi: 10.1017/s0263574700004070.

Sweetser, P. and Wiles, J. (2002) ‘Current AI in Games: A review’, *Australian Journal of Intelligent Information Processing Systems*, 8(1), pp. 24–42. Available at: http://cs.anu.edu.au/ojs/index.php/ajiips.

Gill, S., 2014. Visual Finite State Machine AI Systems. [online] Gamasutra.com. Available at: <https://www.gamasutra.com/view/feature/130578/visual_finite_state_machine_ai_.php> [Accessed 7 December 2020].

Owens, B., 2014. Goal Oriented Action Planning For A Smarter AI. [online] Game Development Envato Tuts+. Available at: <https://gamedevelopment.tutsplus.com/tutorials/goal-oriented-action-planning-for-a-smarter-ai--cms-20793> [Accessed 8 December 2020].