Bot AI: Behaviour Trees

Hailey Graham

Table of Contents

Statement of Completeness

CHECKLIST OF COMPLETED ITEMS

Behaviour Tree for NPC Actions	
A working behaviour tree, converted from a provided FSM solution for transitioning between primary leaf node states (Attack, Roam, Pursue, Flee)	✔
Adjustments made to the Attack, Roam and Flee leaf node states	✔
Behaviour Tree for Weapon Firing	
A secondary behaviour tree for controlling the NPC weapon systems	✔
Adjustments made to the firing system to improve weapon efficiency and usage	✔
Firing activities can operate simultaneously	✔
Firing activities are action driven	✔
Consideration for ammunition/energy levels and environmental factors are made for firing system	✔
NPC Behaviours	
Customised NPC Behaviours	✔
Realistic Player-like Behaviour	✔

Behaviour Trees

In the base of this project, the MechAi was based on a finite state machine (FSM) for NPC actions (such as Attack, Roam, Pursue and Flee) and the firing system was activated if there was an active target. Through this assignment I have converted the FSM for NPC actions into a behaviour tree and constructed a separate behaviour tree for the firing system so that it can work in tandem with the NPC actions.

The benefits of using a behaviour tree over a FSM is that a behaviour tree is more scalable than a FSM. Instead of having to re-write the FSM if a designer wants to add another state, it is simple enough to attach a leaf node to the behaviour tree to add another state. Furthermore, rather than thinking of the behaviours/actions as a circular machine, behaviours trees allow states to be thought of as branching actions that have specific requirements to be executed.

Initially, NPC actions were executed using the logic below:

```
void Update() {

    //Acquire valid attack target: perform frustum and LOS checks and determine closest target
    mechAIAiming.FrustumCheck();

    if (!attackTarget) {
        attackTarget = mechAIAiming.ClosestTarget(mechAIAiming.currentTargets);
        mechAIWeapons.laserBeamAI = false;  //Hard disable on laserBeam
    }
    else
        FiringSystem();

    //FSM - Behaviour Selection
    switch (mechState) {
        case (MechStates.Roam):
            Roam();
        break;
        case (MechStates.Attack):
            Attack();
        break;
        case (MechStates.Pursue):
            Pursue();
        break;
        case (MechStates.Flee):
            Flee();
        break;
    }

    //FSM Transition Logic - Replace this with Decision Tree implementation!
    if (attackTarget && !mechAIAiming.LineOfSight(attackTarget) && !StatusCheck())
        mechState = MechStates.Pursue;
    else if (attackTarget && mechAIAiming.LineOfSight(attackTarget) && !StatusCheck())
        mechState = MechStates.Attack;
    else if (StatusCheck())
        mechState = MechStates.Flee;
    else
        mechState = MechStates.Roam;
}
```

Figure 1: Original FSM for NPC Actions

The Mech transitioned between states based on whether there was an attackTarget, the attackTarget was in their line of sight and if they had enough resources and health (determined by StatusCheck()). This was replaced by the following behaviour tree and helper functions:

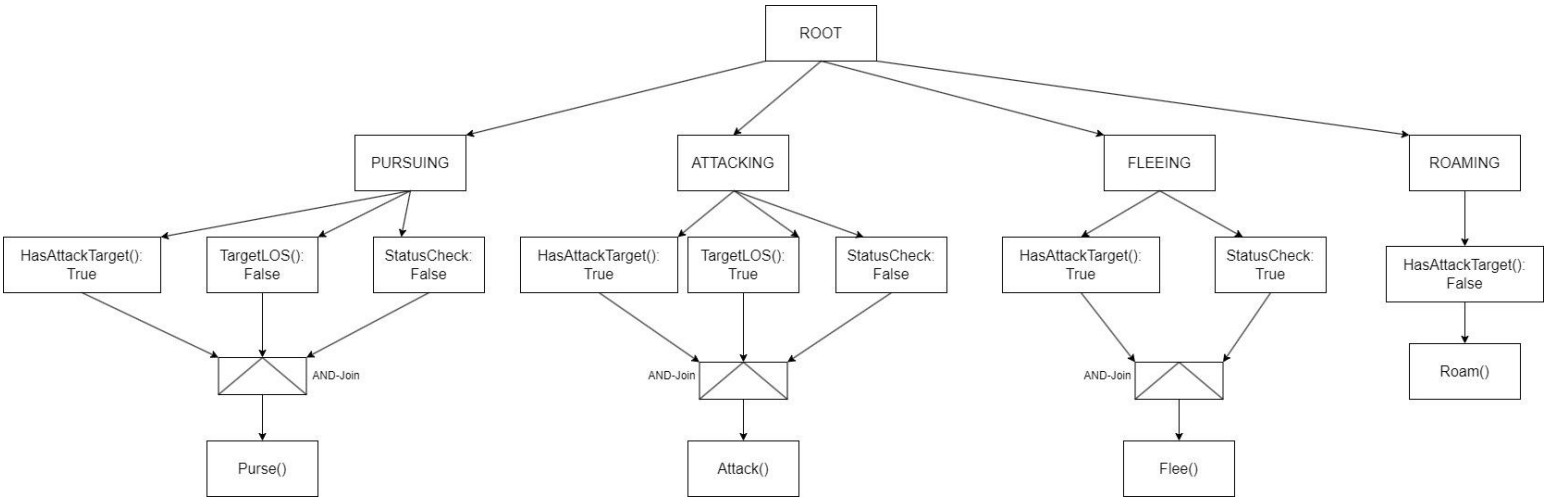

Figure 2: Behaviour Tree for NPC Actions

As for the firing system, this method was converted into a behaviour tree for firing weapons. Each weapon was giving it's own leaf node so that the weapons could fire simulatenously and then within the methods for each weapon, it was determined whether it was appropriate to use the weapon. The behaviour tree is displayed below:

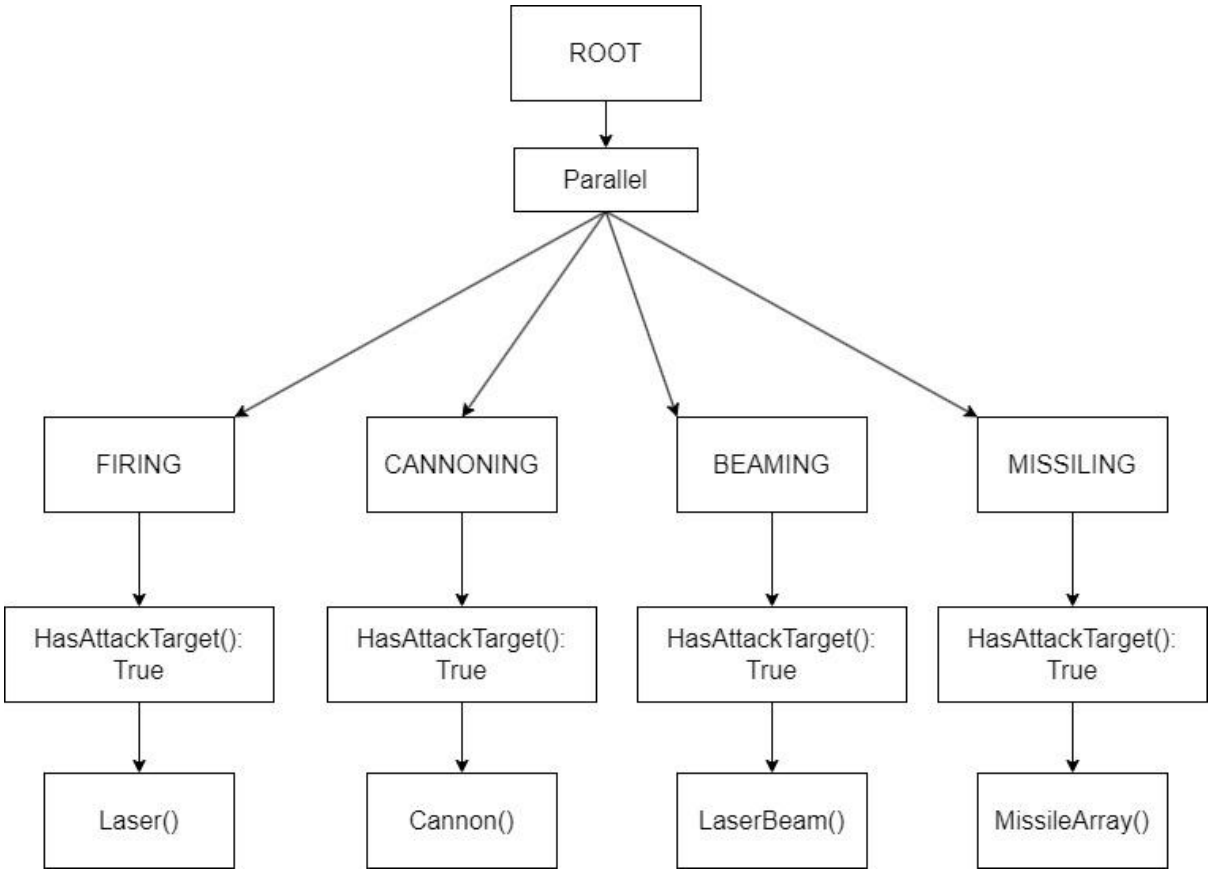

Figure 3: Behaviour Tree for Firing Weapons

The parallel join is a key word specfic to Panda BT (the software used to make the behaviour trees in the project). Using "parallel" before the four different weapon branches means that all the weapons fire as long as the requirements for the leaf nodes are met. So in this case, as long as HasAttackTarget() = true then all the methods calls to the weapons will activate. Within these methods, the weapons may or may not fire depending on other conditionals within the functions.

As for adjusting the firing activities to be action driven and to consider energy levels and environmental factors, some changes were made. For the Laser Beam, it would only fire when the bot is not fleeing so that it conserved resources.

```
[Task]
private void LaserBeam()
{
    //Laser Beam - Strict range, plenty of energy and very tight firing angle
    if (Vector3.Distance(transform.position, attackTarget.transform.position) > 20
        && Vector3.Distance(transform.position, attackTarget.transform.position) < 50
        && mechSystem.energy >= 300 && mechAIAiming.FireAngle(10)
        && (!StatusCheck() || !HasAttackTarget())) //check if the mech is currently fleeing
        mechAIWeapons.laserBeamAI = true;
    else
        mechAIWeapons.laserBeamAI = false;
}
```

Check in LaserBeam() if bot is fleeing

And in the status check, the bot is more likely to attack when they are further away.

```
298         if (attackTarget)
299         {
300             status += Vector3.Distance(transform.position, attackTarget.transform.position); //more likely to attack when further away
301             if(attackTarget.GetComponent<MechSystems>().health < 1000) // more likely to attack when the attackTarget is low on health
302             {
303                 status *= 1.5f;
304             }
305             //attacked = 1.5f; //the threshold to start attacking is higher if there is an attack target -> fear factor
306         }
```

Line 300 puts more priority to attacking if the bot is further away from the attackTarget

NPC Behaviours

Functionally, the behaviour tree is very similar to the FSM. However, some adjustments were made so that the Mech made more intelligent decisions around Roaming and Fleeing, and also the Mech exhibited more player-like behaviours when Roaming, Fleeing and Attacking.

For the roaming behaviour, instead of simply travelling between random patrol points, additional logic was added to the method. When the Mech was within line of sight of the patrol point they were walking towards, the Mech would check if the patrol point had a resource pack. If the pack was there, the Mech would continue to move towards the patrol point, otherwise, the Mech would choose another random patrol point. This adjusted behaviour means that the Mech is not wasting time going to a patrol point that does not have any resources and this behaviour is more player-like as, to a human, it does not make sense to go to an empty resource area. Furthermore, the Mech only checks if the patrol point has a resource pack if the point is in line of sight. This is more player-like as humans cannot see through walls.

```
[Task]
//FSM Behaviour: Roam - Roam between random patrol points
HalleyG123
private void Roam() {
    //Move towards random patrol point
    if (Vector3.Distance( a: transform.position,  b: patrolPoints[patrolIndex].transform.position) <= 2.0f) {
        patrolIndex = Random.Range(0, patrolPoints.Length - 1);
    }
    //if in line of sight check if patrol point has resource, otherwise choose new point
    else if (mechAIAiming.LineOfSight(patrolPoints[patrolIndex]))
    {
        RaycastHit[] hits = Physics.RaycastAll( origin: mechAIAiming.rayCastPoint.transform.position,
            direction: -(mechAIAiming.rayCastPoint.transform.position - patrolPoints[patrolIndex].transform.position).normalized,
            //make sure raycast only goes as far as the patrol point of interest
            Vector3.Distance( a: mechAIAiming.rayCastPoint.transform.position,  b: patrolPoints[patrolIndex].transform.position));

        if (hits.Length > 0)
        {
            for (int i = 0; i < hits.Length; i++)
            {
                //if there is a resource pack
                if (hits[i].transform.GetComponent<Pickup>())
                {
                    mechAIMovement.Movement(patrolPoints[patrolIndex].transform.position,  stopDistance: 1);
                    //Look at random patrol points
                    mechAIAiming.RandomAimTarget(patrolPoints);
                    return;
                }
            }
        }
        //choose a different point if there is no resource pack
        patrolIndex = Random.Range(0, patrolPoints.Length - 1);
    }
    else {
        mechAIMovement.Movement(patrolPoints[patrolIndex].transform.position,  stopDistance: 1);
    }
    //Look at random patrol points
    mechAIAiming.RandomAimTarget(patrolPoints);
}
```

Figure 4: Adjusted Roaming Behaviour

For the attacking behaviour, to have the Mech behaviour more player-like, there was a 25% chance that the mech just runs away from combat. Using a random float between 0 and 1, if the float was like than 0.25, the mech's state would change to flee. This was implemented because as a player, I personally may get scared during combat and just run away even if I am "fine" in terms of health and resources. I wanted to emulate that in this Mech. Furthermore, the Mech's aim was made less accurate by adding a "wonky" variable to the mechAi movement. So, it would appear that the Mech is not just beelining towards their target, making their movement more player-like.

```
[Task]
//FSM Behaviour: Attack
HalleyG123
private void Attack() {

    //If there is a target, set it as the aimTarget
    if (attackTarget && mechAIAiming.LineOfSight(attackTarget)) {
        //There is a 25% chance the mech just runs away
        float rnd = Random.Range(0, 1);
        if (rnd < 0.25)
        {
            mechState = MechStates.Flee;
        }
        else
        {
            //Child object correction - wonky pivot point
            mechAIAiming.aimTarget = attackTarget.transform.GetChild(0).gameObject;

            //Move Towards attack Target
            Vector3 wonky = new Vector3( x: Random.Range(-1, 1),  y: 0,  z: Random.Range(-1, 1)); //variation in movement
            if (Vector3.Distance( a: transform.position,  b: attackTarget.transform.position) >= 45.0f) {
                mechAIMovement.Movement( position: attackTarget.transform.position + wonky,  stopDistance: 45);
            }
            //Otherwise "strafe" - move towards random patrol points at intervals
            else if (Vector3.Distance( a: transform.position,  b: attackTarget.transform.position) < 45.0f && Time.time > attackTimer) {
                patrolIndex = Random.Range(0, patrolPoints.Length - 1);
                mechAIMovement.Movement(patrolPoints[patrolIndex].transform.position,  stopDistance: 2);
                attackTimer = Time.time + attackTime + Random.Range(-0.5f, 0.5f);
            }

            //Track position of current target to pursue if lost
            pursuePoint = attackTarget.transform.position;
        }
    }
}
```

Figure 5: Adjusted Attacking Behaviour

The pursuing behaviour was left as in the base project but the fleeing was adjusted. Instead of fleeing to a random patrol point, the Mech's goal instead was to move out of line of sight as quickly as possible. In doing so, the Mech reduces the amount of time they are in the line of fire from an attacking target. This strategic retreat is done by finding the closest patrol point that is out of sight of the attack target. The method utilises an adjusted LineOfSight method call that determines if a gameobject is within the mech's line of sight. See figures below:

```
void Flee() {

    //If there is an attack target, set it as the aimTarget
    if (attackTarget && mechAIAiming.LineOfSight(attackTarget)) {
        //Child object correction - wonky pivot point
        mechAIAiming.aimTarget = attackTarget.transform.GetChild(0).gameObject;
    } else {
        //Look at random patrol points
        mechAIAiming.RandomAimTarget(patrolPoints);
    }

    if (Vector3.Distance(a: transform.position, b: patrolPoints[patrolIndex].transform.position) <= 2.0f)
    {
        //choose closest patrol point that is out of sight of the attackTarget
        if (attackTarget)
        {
            Array.Sort(patrolPoints, comparison: (a:GameObject , b:GameObject ) =>
            {
                return (int)(Vector3.Distance(a: b.transform.position, b: transform.position) - Vector3.Distance(a: a.transform.position, b: transform.position));
            });

            for (int i = 0; i < patrolPoints.Length; i++)
            {
                if (LineOfSight(patrolPoints[i].transform.position, attackTarget))
                {
                    patrolIndex = i;
                    break;
                }
            }
        }
        else
        {
            patrolIndex = Random.Range(0, patrolPoints.Length - 1);
        }
    }
    else
    {
        mechAIMovement.Movement(patrolPoints[patrolIndex].transform.position, stopDistance: 1);
    }
```

Figure 6: Adjusted Fleeing Behaviour with Strategic Retreat logic implemented

```
//Method to determine if object is within LOS of Mech
1 usage  HaileyG123
private bool LineOfSight(Vector3 position, GameObject thisTarget) {

    //Need to correct for wonky pivot point - Mech model pivot at base instead of centre
    Vector3 correction = thisTarget.transform.GetChild(0).gameObject.transform.position;

    RaycastHit hit;
    if (Physics.Raycast(origin: position, direction: -(position - correction).normalized, out hit, maxDistance: 100.0f)) {

        Debug.DrawLine(start: position, end: hit.point, Color.red );
        //if the raycasthit travelled further than the proposed position, there is a clear line of sight
        if (hit.distance > Vector3.Distance(a: position, b: thisTarget.transform.position))
        {
            return true;
        }
        else
            return false;
    }
    else
        return false;
}
```

Figure 7: new LineOfSight method that determines if any gameobject is within the mech's line of sight

As such, the mech is making more intelligent decisions about fleeing and in turn this behaviour is more player-like since it makes more sense to get out of way when an opponenet is firing at you rather than just running away. This strategy is often used by more

experienced players. And thus, it is predicted that this mech will do better in the Bot tournament.

Finally, the StatusCheck() method was adjusted to better align with the kind of bot that I have constructed. Throughout this report it has been shown that this mech is cautious and strategic but because of this it may display fear-like behaviours such as running away sooner than necessary. I wanted this to be reflected in the status check as well.

The following variables were added to the status check heuristic. If the attackTarget is further away, the mech is more likely to attack. This lends well to a sniper bot. If the attackTarget is below half health, the mech is also more likely to attack it. This means that the mech is more likely to gang up on a target if another bot is attacking it. I also increased the threshold for the bot to choose to attack, if there is an attackTarget to emulate a “fear factor”. Finally, if the mech has already died, they are less likely to attack and conserve resources. But if they have a high score, they will be more likely to attack to keep on their winning streak.

```
[Task]
//Method for checking heuristic status of Mech to determine if Fleeing is necessary
HalleyG123 *
private bool StatusCheck()
{

    float status = 0;
    int attacked = 1;

    //number of deaths and score variables
    int score = gameManager.playerScores[mechSystem.ID];
    int deaths = gameManager.playerDeaths[mechSystem.ID];

    if (attackTarget)
    {
        status += Vector3.Distance(a: transform.position, b: attackTarget.transform.position); //more likely to attack when further away
        if(attackTarget.GetComponent<MechSystems>().health < 1000) // more likely to attack when the attackTarget is low on health
        {
            status *= 1.5f;
        }
        attacked = 2; //the threshold to start attacking is higher if there is an attack target -> fear factor
    }

    status += (mechSystem.health * 0.5f) + mechSystem.energy + (mechSystem.shells * 3) +
              (mechSystem.missiles * 5) - deaths + (score * 2); //less likely to attack if it has already died


    if (status > 1500 * attacked)
        return false;
    else
        return true;
}
```

Figure 8: Adjusted StatusCheck() method

Conclusion

In conclusion, through this project, I have converted the base finite state machine for the mech AI to a behaviour tree for NPC actions. I converted the FiringSystem() into another behaviour tree for weapons firing that can work in parallel with the NPC behaviours. By conducting research such as observing my own play style and investigating the Panda BT documentation, I have adjusted the leaf nodes for the two behaviour trees so that the mech displays player-like behaviours, uses weapons more efficiently and makes more intelligent decisions when Fleeing, Roaming and Attacking. Through this assignment, I have learnt about the benefits of using

behaviour trees over finite state machines and the methods to make game AI that players find challenging and realistic.