Contents

Links

Video:

https://www.youtube.com/watch?v=dNsFmNwuI0M

Games & AI Coursework 2

Operation All-In

Introduction

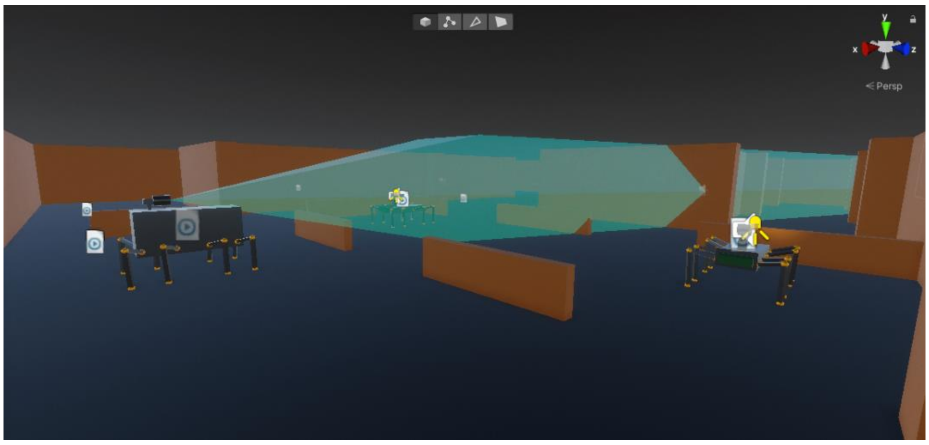
Operation All-In (Op. AI) is a first-person shooter set against a number of drone opponents. There are two types; a ‘dumb’ assault drone with a top-mounted machinegun and short ranged visual scanner, and a ‘smart’ commander drone equipped with radar. Alone, the assault drones can be easily neutralised by abusing their short view range and lack of movement. However, the smart drone keeps track of the player’s position using its radar and can command the assault drones depending on the situation.

AI Techniques

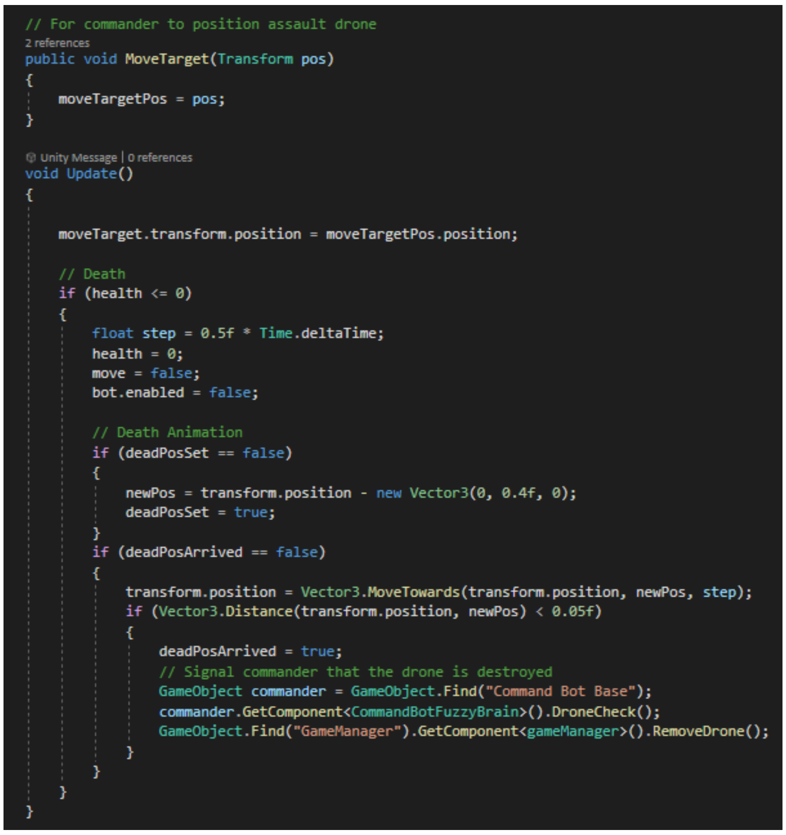
The two AI techniques I utilized are a finite state machine for the dumb drones, and fuzzy logic for the command drone. The finite state machine is suitable for the dumb drones, as whether they are being commanded or not, they are only performing a certain action at a time and that action is very clear and predictable. This fits their design, as they are meant to be simple turrets that follow the orders given to them by the command drone. They react only to whether they can see the player or not, and their response of whether or not to shoot is binary.

The command drone, on the other hand uses fuzzy logic to decide its actions depending on several factors such as aggression, protection and distance to the player. These factors change depending on how many drones there are alive, how much health the player has, and once again the distance of the player. If the player is doing well, the commander may decide to hide far away from the player and bring a drone to it in order to help it defend. Meanwhile, when the player’s health is low and a lot of drones are alive, it may decide to stay put to ensure it can catch the player in its radar and send a drone in to finish the player off. This technique makes the drone appear ‘smarter’, choosing its actions depending on the situation and thus providing varying gameplay in each run.

An alternative technique that could have been used for the commander drone is machine learning. This is where a neural network is used to ‘teach’ the AI to react to any situation it is put in using whatever inputs it has available. For instance, if given the ability to control the position of any drone, it could scan the environment and place them in a way that makes it difficult for the player to get through. This technique would have made the AI much smarter and a lot less predictable, however would have required a lot of training data to get it to a playable state. When putting AI against a simple task it is easy to speed up the time frame and create several copies to speed up the training time, however against a player opponent the game would have to be actually played by someone many times to teach it properly. This is why I opted to use fuzzy logic instead, although ML would have definitely been possible given enough time and might have provided a more interesting result.

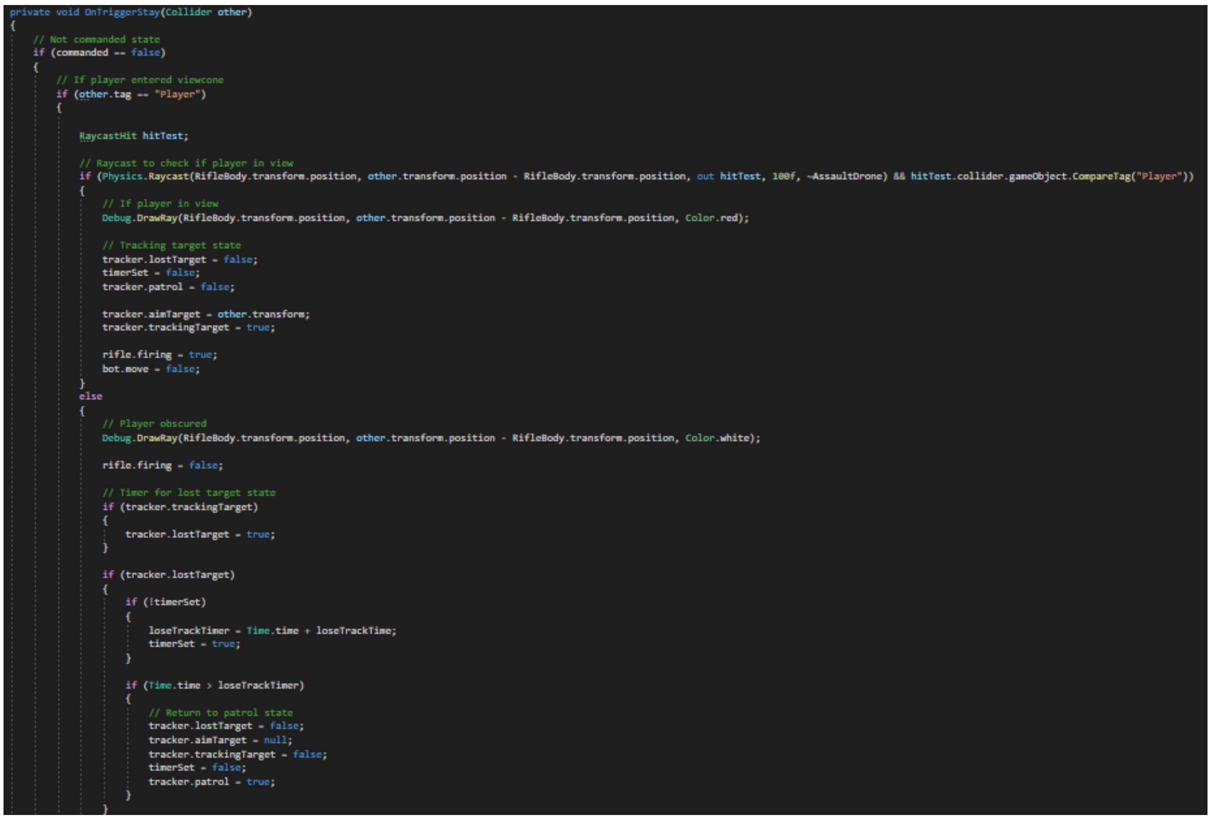


Method



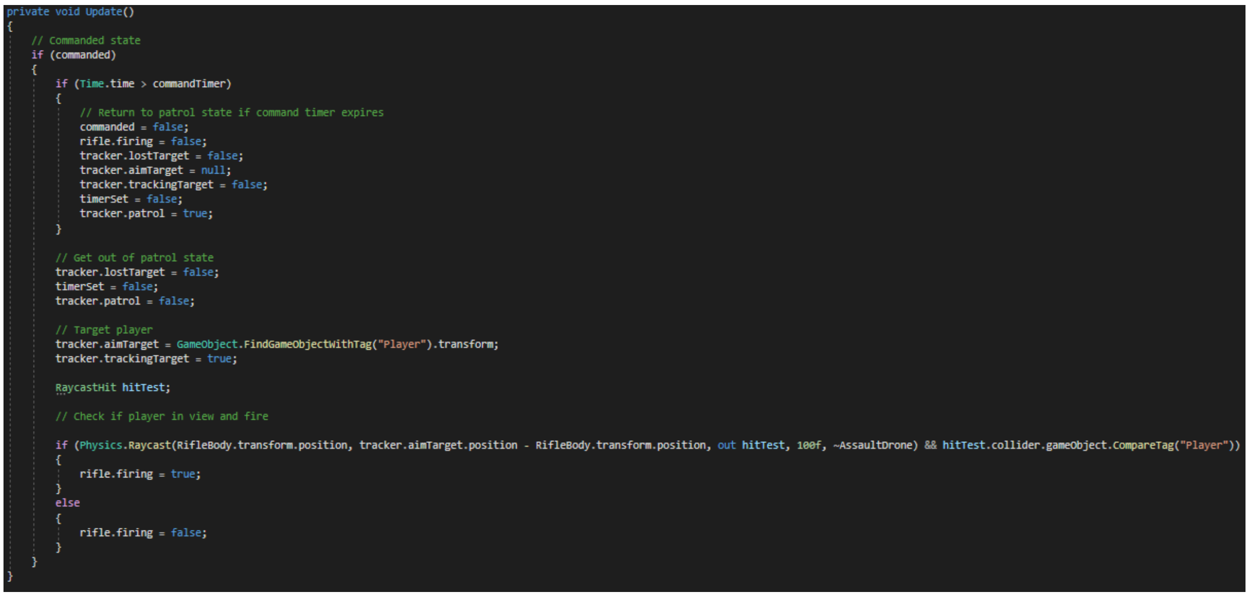
```
// For commander to position assault drone
2 references
public void MoveTarget(Transform pos)
{
    moveTargetPos = pos;
}

Unity Message | 0 references
void Update()
{

    moveTarget.transform.position = moveTargetPos.position;

    // Death
    if (health <= 0)
    {
        float step = 0.5f * Time.deltaTime;
        health = 0;
        move = false;
        bot.enabled = false;

        // Death Animation
        if (deadPosSet == false)
        {
            newPos = transform.position - new Vector3(0, 0.4f, 0);
            deadPosSet = true;
        }
        if (deadPosArrived == false)
        {
            transform.position = Vector3.MoveTowards(transform.position, newPos, step);
            if (Vector3.Distance(transform.position, newPos) < 0.05f)
            {
                deadPosArrived = true;
                // Signal commander that the drone is destroyed
                GameObject commander = GameObject.Find("Command Bot Base");
                commander.GetComponent<CommandBotFuzzyBrain>().DroneCheck();
                GameObject.Find("GameManager").GetComponent<gameManager>().RemoveDrone();
            }
        }
    }
}
```

The assault drones have a move target that they path find towards using Unity's pathfinding system as long as they are alive. By default, the drones stay in position, however the commander is able to access the "MoveTarget" function to order them to move elsewhere such as to protect or attack. The animation of the drone dying is done by manipulating the transform position, as my goal was to animate the drones semi-procedurally wherein, they are not randomly generated but also do not use the unity animation system.



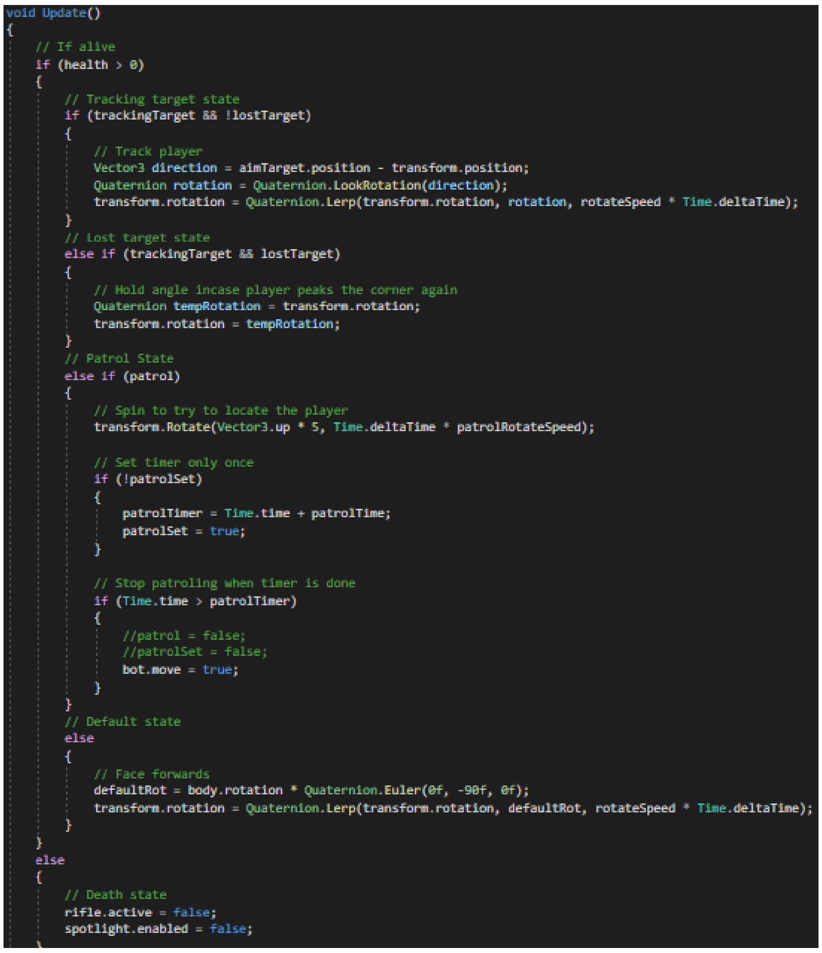
```
private void OnTriggerStay(Collider other)
{
    // Not commanded state
    if (commanded == false)
    {
        // If player entered viewcone
        if (other.tag == "Player")
        {

            RaycastHit hitTest;

            // Raycast to check if player in view
            if (Physics.Raycast(RifleBody.transform.position, other.transform.position - RifleBody.transform.position, out hitTest, 100f, ~AssaultDrone) && hitTest.collider.gameObject.CompareTag("Player"))
            {
                // If player in view
                Debug.DrawRay(RifleBody.transform.position, other.transform.position - RifleBody.transform.position, Color.red);

                // Tracking target state
                tracker.lostTarget = false;
                timerSet = false;
                tracker.patrol = false;

                tracker.aimTarget = other.transform;
                tracker.trackingTarget = true;

                rifle.firing = true;
                bot.move = false;
            }
            else
            {
                // Player obscured
                Debug.DrawRay(RifleBody.transform.position, other.transform.position - RifleBody.transform.position, Color.white);

                rifle.firing = false;

                // Timer for lost target state
                if (tracker.trackingTarget)
                {
                    tracker.lostTarget = true;
                }

                if (tracker.lostTarget)
                {
                    if (!timerSet)
                    {
                        loseTrackTimer = Time.time + loseTrackTime;
                        timerSet = true;
                    }

                    if (Time.time > loseTrackTimer)
                    {
                        // Return to patrol state
                        tracker.lostTarget = false;
                        tracker.aimTarget = null;
                        tracker.trackingTarget = false;
                        timerSet = false;
                        tracker.patrol = true;
                    }
                }
```

This is the camera script which takes care of the assault drone’s view of the player. When the drone is in its dumb state where it is not commanded, the drone checks for collisions in its view-cone and casts a ray to determine whether the player is obscured. If it spots the player, it begins it’s firing state. If it then loses track of the player, it swaps to the lost target state to keep watching where the player was last spotted and then finally the patrol state to attempt to locate the player again. This essentially makes it a motion tracked turret, which is quite easy to get around or kill.



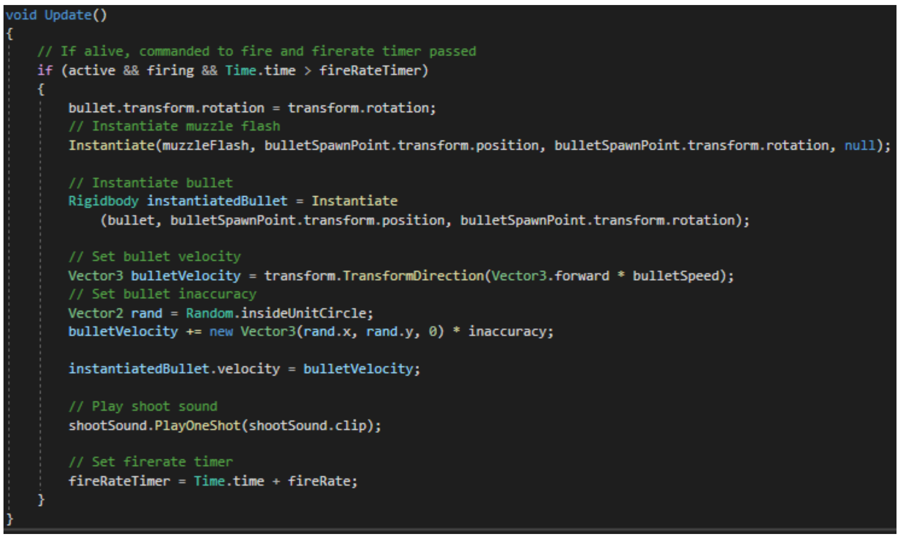
```
private void Update()
{
    // Commanded state
    if (commanded)
    {
        if (Time.time > commandTimer)
        {
            // Return to patrol state if command timer expires
            commanded = false;
            rifle.firing = false;
            tracker.lostTarget = false;
            tracker.aimTarget = null;
            tracker.trackingTarget = false;
            timerSet = false;
            tracker.patrol = true;
        }

        // Get out of patrol state
        tracker.lostTarget = false;
        timerSet = false;
        tracker.patrol = false;

        // Target player
        tracker.aimTarget = GameObject.FindGameObjectWithTag("Player").transform;
        tracker.trackingTarget = true;

        RaycastHit hitTest;

        // Check if player in view and fire

        if (Physics.Raycast(RifleBody.transform.position, tracker.aimTarget.position - RifleBody.transform.position, out hitTest, 100f, ~AssaultDrone) && hitTest.collider.gameObject.CompareTag("Player"))
        {
            rifle.firing = true;
        }
        else
        {
            rifle.firing = false;
        }
    }
}
```

Meanwhile, if the drone is in the commanded state, they are fed the player’s position no matter the range and cast a ray-cast to check if the player is obscured causing them to either fire or keep tracking through the obstruction. This means every time the player comes into their view; they are ready to fire and do so as long as the player is detected by radar. This makes them a much greater threat compared to their non-commanded state.



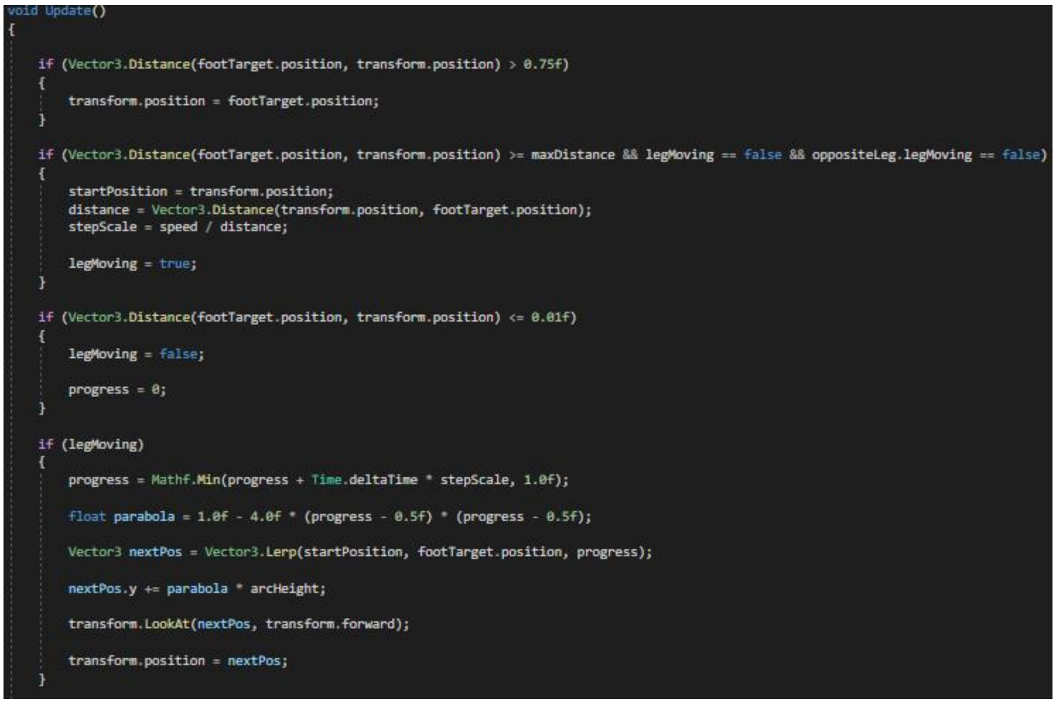
```
void Update()
{
    // If alive
    if (health > 0)
    {
        // Tracking target state
        if (trackingTarget && !lostTarget)
        {
            // Track player
            Vector3 direction = aimTarget.position - transform.position;
            Quaternion rotation = Quaternion.LookRotation(direction);
            transform.rotation = Quaternion.Lerp(transform.rotation, rotation, rotateSpeed * Time.deltaTime);
        }
        // Lost target state
        else if (trackingTarget && lostTarget)
        {
            // Hold angle incase player peaks the corner again
            Quaternion tempRotation = transform.rotation;
            transform.rotation = tempRotation;
        }
        // Patrol State
        else if (patrol)
        {
            // Spin to try to locate the player
            transform.Rotate(Vector3.up * 5, Time.deltaTime * patrolRotateSpeed);

            // Set timer only once
            if (!patrolSet)
            {
                patrolTimer = Time.time + patrolTime;
                patrolSet = true;
            }

            // Stop patroling when timer is done
            if (Time.time > patrolTimer)
            {
                //patrol = false;
                //patrolSet = false;
                bot.move = true;
            }
        }
        // Default state
        else
        {
            // Face forwards
            defaultRot = body.rotation * Quaternion.Euler(0f, -90f, 0f);
            transform.rotation = Quaternion.Lerp(transform.rotation, defaultRot, rotateSpeed * Time.deltaTime);
        }
    }
    else
    {
        // Death state
        rifle.active = false;
        spotlight.enabled = false;
```

This script ties into the track script, which contains the large sum of the state machine. This script determines what the drone should be doing when it is in a particular state, such as the track state to follow the player, lost target state to maintain the angle where the player was last seen, and the patrol state where the turret spins in order to locate the player. Personally, I am not too proud of this script, as instead of using several bools I could have used a single Enum to determine which state the drone is in and possibly even tied it to the Unity animator to visualise the different states and transitions. However, it still fulfils its purpose well as when fed with a particular state by another script, the drone does exactly what it is instructed to do. The tracking state linearly interpolates the rotation of the turret towards the player at a set speed that allows the player to outrun it if they're going at maximum speed but without being too forgiving. The lost target state simply holds the rotation, which stops the player from being able to quickly peak in and out of a corner as the drone will be aiming directly at that corner. Finally, the patrol state rotates the drone around its vertical axis at a configurable speed. When killed, the spotlight goes off to show the player that the drone will not be able to shoot them anymore but will still be capable of walking.



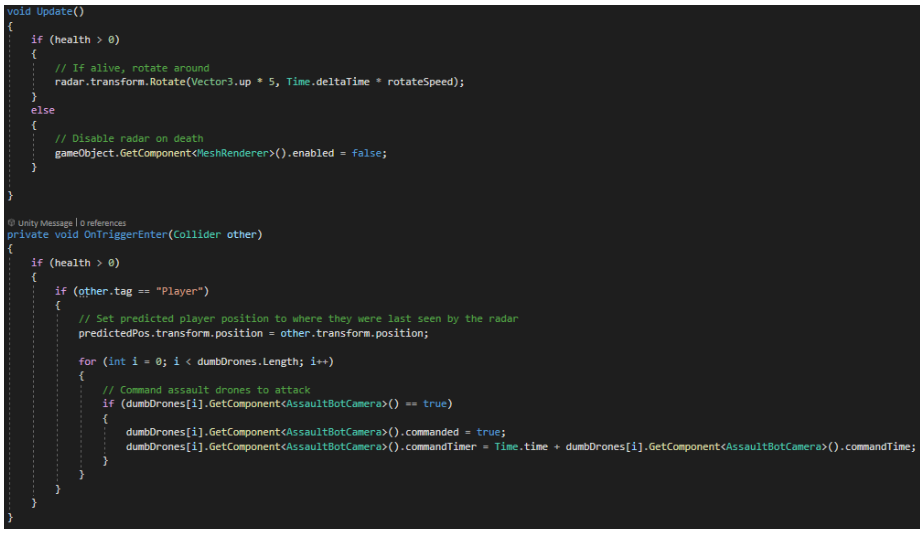
```
void Update()
{
    // If alive, commanded to fire and firerate timer passed
    if (active && firing && Time.time > fireRateTimer)
    {
        bullet.transform.rotation = transform.rotation;
        // Instantiate muzzle flash
        Instantiate(muzzleFlash, bulletSpawnPoint.transform.position, bulletSpawnPoint.transform.rotation, null);

        // Instantiate bullet
        Rigidbody instantiatedBullet = Instantiate
            (bullet, bulletSpawnPoint.transform.position, bulletSpawnPoint.transform.rotation);

        // Set bullet velocity
        Vector3 bulletVelocity = transform.TransformDirection(Vector3.forward * bulletSpeed);
        // Set bullet inaccuracy
        Vector2 rand = Random.insideUnitCircle;
        bulletVelocity += new Vector3(rand.x, rand.y, 0) * inaccuracy;

        instantiatedBullet.velocity = bulletVelocity;

        // Play shoot sound
        shootSound.PlayOneShot(shootSound.clip);

        // Set firerate timer
        fireRateTimer = Time.time + fireRate;
    }
}
```

Finally, when commanded to fire, the drone instantiates a bullet prefab, which is an actual physical bullet using the unity particle system. For this reason, it is assigned a muzzle velocity value to determine how fast the bullet should be launched, and an inaccuracy value which assigns a small variation to the launch vector causing it to have a random amount of spread, thus not being fully accurate every time. It also instantiates a muzzle flash prefab and plays a sound clip of a firing sound.



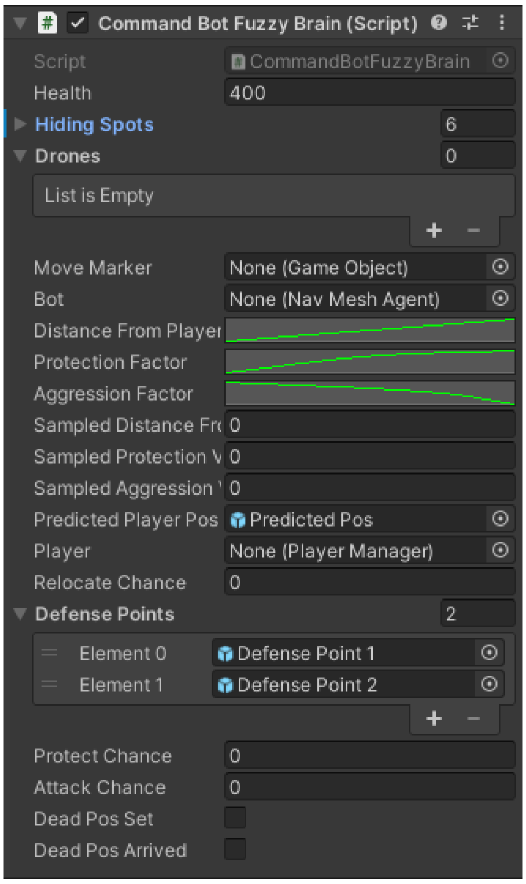
```
void Update()
{
    if (Vector3.Distance(footTarget.position, transform.position) > 0.75f)
    {
        transform.position = footTarget.position;
    }

    if (Vector3.Distance(footTarget.position, transform.position) >= maxDistance && legMoving == false && oppositeLeg.legMoving == false)
    {
        startPosition = transform.position;
        distance = Vector3.Distance(transform.position, footTarget.position);
        stepScale = speed / distance;

        legMoving = true;
    }

    if (Vector3.Distance(footTarget.position, transform.position) <= 0.01f)
    {
        legMoving = false;

        progress = 0;
    }

    if (legMoving)
    {
        progress = Mathf.Min(progress + Time.deltaTime * stepScale, 1.0f);

        float parabola = 1.0f - 4.0f * (progress - 0.5f) * (progress - 0.5f);

        Vector3 nextPos = Vector3.Lerp(startPosition, footTarget.position, progress);

        nextPos.y += parabola * arcHeight;

        transform.LookAt(nextPos, transform.forward);

        transform.position = nextPos;
    }
```

One last part that both the assault and command drones share is the leg movement. As mentioned before, I wanted the drones to be procedurally animated, and thus I used an IK system for the legs which controls the placement of the feet. There are invisible markers attached to the robot's body, and when the feet get too far away from these markers, they begin to move towards them using a Lerp function. To imitate the movement of the feet, I implemented a parabola equation to give the feet a U-shaped movement. Whereas this has no effect on the gameplay, I believe it helps make the drone movement appear more realistic.



```
void Update()
{
    if (health > 0)
    {
        // If alive, rotate around
        radar.transform.Rotate(Vector3.up * 5, Time.deltaTime * rotateSpeed);
    }
    else
    {
        // Disable radar on death
        gameObject.GetComponent<MeshRenderer>().enabled = false;
    }

}

Unity Message | 0 references
private void OnTriggerEnter(Collider other)
{
    if (health > 0)
    {
        if (other.tag == "Player")
        {
            // Set predicted player position to where they were last seen by the radar
            predictedPos.transform.position = other.transform.position;

            for (int i = 0; i < dumbDrones.Length; i++)
            {
                // Command assault drones to attack
                if (dumbDrones[i].GetComponent<AssaultBotCamera>() == true)
                {
                    dumbDrones[i].GetComponent<AssaultBotCamera>().commanded = true;
                    dumbDrones[i].GetComponent<AssaultBotCamera>().commandTimer = Time.time + dumbDrones[i].GetComponent<AssaultBotCamera>().commandTime;
                }
            }
        }
    }
}
```

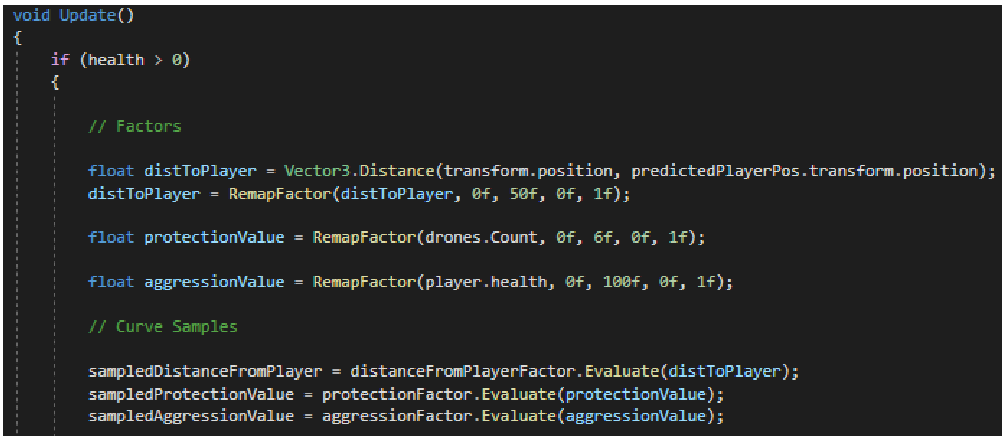
The command drone radar is very similar to the individual assault drone view-cones, however when the player is detected inside of the view-cone collider, it loops through every assault drone and puts them into the commanded state, causing them to fire at the player.



Command Bot Fuzzy Brain (Script)

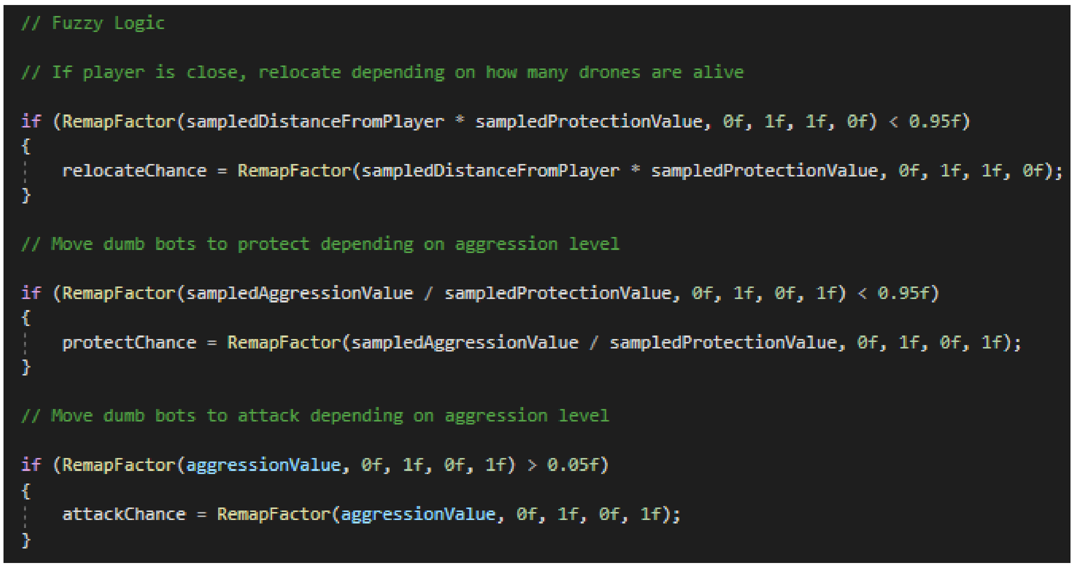
Script	CommandBotFuzzyBrain
Health	400
Hiding Spots	6
Drones	0
List is Empty	
Move Marker	None (Game Object)
Bot	None (Nav Mesh Agent)
Distance From Player	
Protection Factor	
Aggression Factor	
Sampled Distance Fr	0
Sampled Protection V	0
Sampled Aggression '	0
Predicted Player Pos	Predicted Pos
Player	None (Player Manager)
Relocate Chance	0
Defense Points	2
Element 0	Defense Point 1
Element 1	Defense Point 2
Protect Chance	0
Attack Chance	0
Dead Pos Set	☐
Dead Pos Arrived	☐

The command bot itself, however, is controlled by a fuzzy logic script. This script takes several factors, processes them, and then gives the drone several actions it could perform. This allows it to react to changes in its environment, but not always perform the same actions when put into similar situations. This makes the AI appear more 'intelligent' and sometimes unpredictable, although over time the player will be able to figure out it's algorithm and get used to its actions.



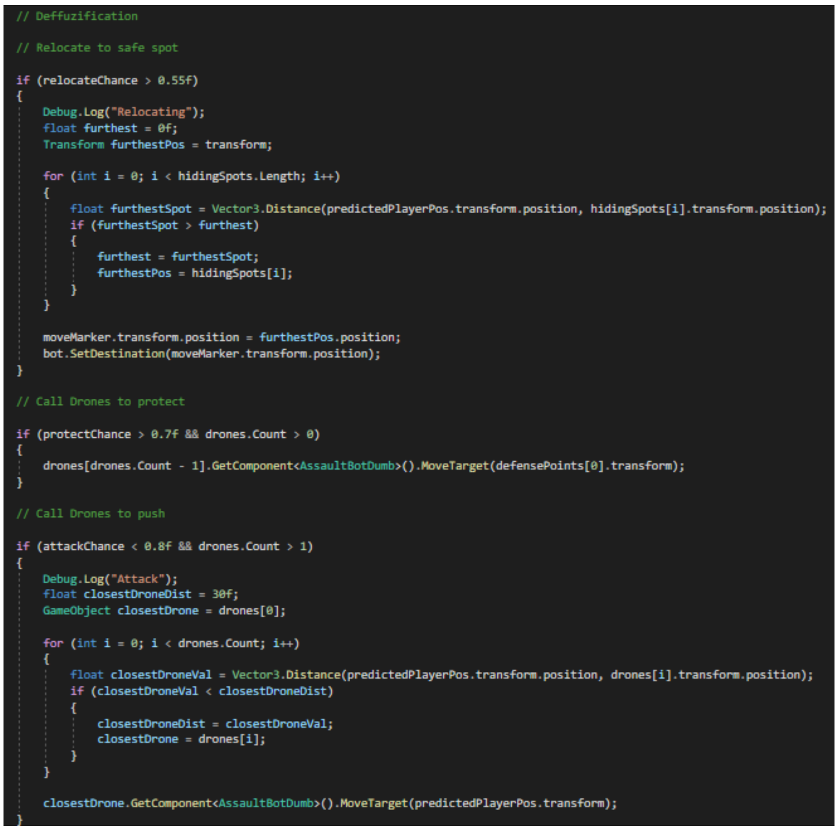
```
void Update()
{
    if (health > 0)
    {
        // Factors

        float distToPlayer = Vector3.Distance(transform.position, predictedPlayerPos.transform.position);
        distToPlayer = RemapFactor(distToPlayer, 0f, 50f, 0f, 1f);

        float protectionValue = RemapFactor(drones.Count, 0f, 6f, 0f, 1f);

        float aggressionValue = RemapFactor(player.health, 0f, 100f, 0f, 1f);

        // Curve Samples

        sampledDistanceFromPlayer = distanceFromPlayerFactor.Evaluate(distToPlayer);
        sampledProtectionValue = protectionFactor.Evaluate(protectionValue);
        sampledAggressionValue = aggressionFactor.Evaluate(aggressionValue);
```

First, the script takes in data such as how far away the player is, how many drones there are, and the player health. This is run through a remap function which converts the values onto a scale of 0 to 1 based on a minimum and maximum value. Next, these factors are placed on the animation curves using the Evaluate function, which allows me to tweak the effect they have on the AI's actions by editing the curve.



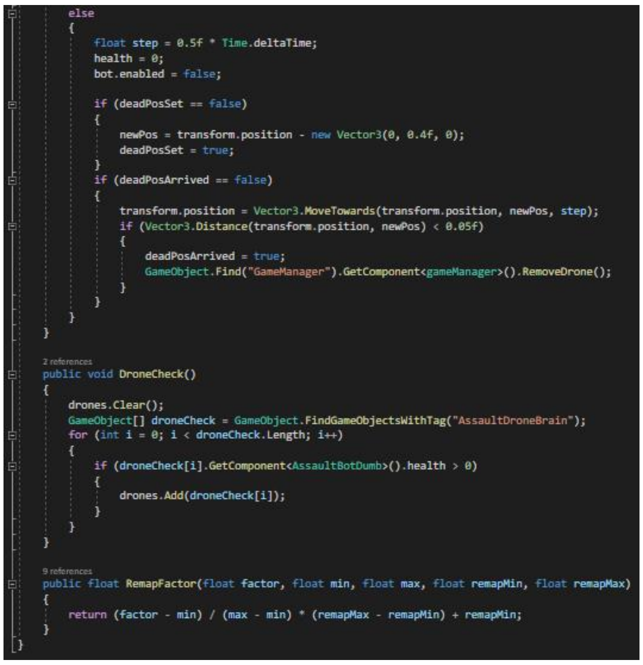
```
// Fuzzy Logic

// If player is close, relocate depending on how many drones are alive

if (RemapFactor(sampledDistanceFromPlayer * sampledProtectionValue, 0f, 1f, 1f, 0f) < 0.95f)
{
    relocateChance = RemapFactor(sampledDistanceFromPlayer * sampledProtectionValue, 0f, 1f, 1f, 0f);
}

// Move dumb bots to protect depending on aggression level

if (RemapFactor(sampledAggressionValue / sampledProtectionValue, 0f, 1f, 0f, 1f) < 0.95f)
{
    protectChance = RemapFactor(sampledAggressionValue / sampledProtectionValue, 0f, 1f, 0f, 1f);
}

// Move dumb bots to attack depending on aggression level

if (RemapFactor(aggressionValue, 0f, 1f, 0f, 1f) > 0.05f)
{
    attackChance = RemapFactor(aggressionValue, 0f, 1f, 0f, 1f);
}
```

Next, the values are fuzzified further, such as by the relocate chance sampling both the player distance and protection value multiplied together or the protect chance taking the aggression value divided by the protection value. The attack chance is based on the aggression value, which is dependent on the player's health. This makes the AI make its choice of actions based on how much of a danger the player is based on their health and distance compared to how protected the drone is based on how many drones are currently alive.



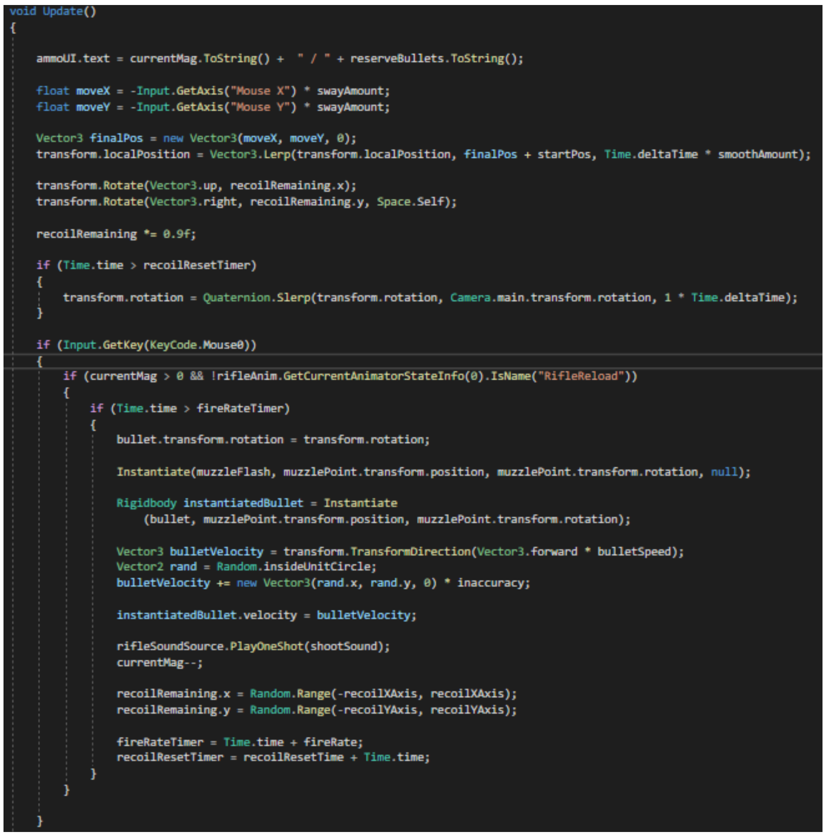
```csharp
// Deffuzification

// Relocate to safe spot

if (relocateChance > 0.55f)
{
    Debug.Log("Relocating");
    float furthest = 0f;
    Transform furthestPos = transform;

    for (int i = 0; i < hidingSpots.Length; i++)
    {
        float furthestSpot = Vector3.Distance(predictedPlayerPos.transform.position, hidingSpots[i].transform.position);
        if (furthestSpot > furthest)
        {
            furthest = furthestSpot;
            furthestPos = hidingSpots[i];
        }
    }

    moveMarker.transform.position = furthestPos.position;
    bot.SetDestination(moveMarker.transform.position);
}

// Call Drones to protect

if (protectChance > 0.7f && drones.Count > 0)
{
    drones[drones.Count - 1].GetComponent<AssaultBotDumb>().MoveTarget(defensePoints[0].transform);
}

// Call Drones to push

if (attackChance < 0.8f && drones.Count > 1)
{
    Debug.Log("Attack");
    float closestDroneDist = 30f;
    GameObject closestDrone = drones[0];

    for (int i = 0; i < drones.Count; i++)
    {
        float closestDroneVal = Vector3.Distance(predictedPlayerPos.transform.position, drones[i].transform.position);
        if (closestDroneVal < closestDroneDist)
        {
            closestDroneDist = closestDroneVal;
            closestDrone = drones[i];
        }
    }

    closestDrone.GetComponent<AssaultBotDumb>().MoveTarget(predictedPlayerPos.transform);
}
```

Finally, the script then puts the fuzzy values into context, applying different actions depending on the situation. If the bot decides to relocate, it looks through a number of pre-set hiding spots and determines which one is furthest away from the player before moving to that one. If it decides it is in danger, it can call one of the drones to protect it, or if it is feeling well protected and decides the player is in danger, it can call the closest drone to push their last known position. This gives the player a different challenge in each run of the game, such as being surprised by a drone sneaking up behind them or being forced fight back every drone coming to defend the commander.



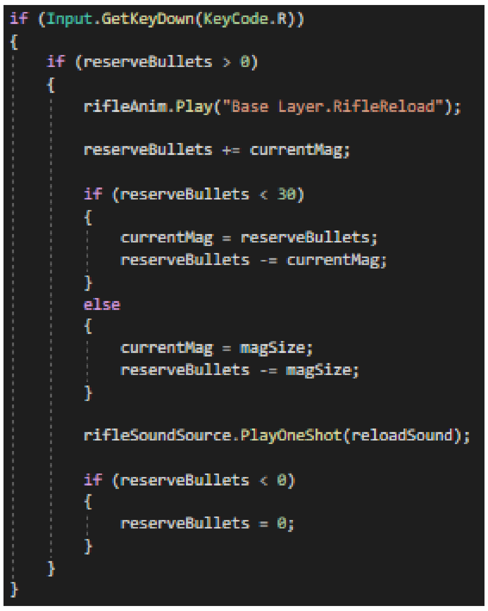
```csharp
        else
        {
            float step = 0.5f * Time.deltaTime;
            health = 0;
            bot.enabled = false;

            if (deadPosSet == false)
            {
                newPos = transform.position - new Vector3(0, 0.4f, 0);
                deadPosSet = true;
            }
            if (deadPosArrived == false)
            {
                transform.position = Vector3.MoveTowards(transform.position, newPos, step);
                if (Vector3.Distance(transform.position, newPos) < 0.05f)
                {
                    deadPosArrived = true;
                    GameObject.Find("GameManager").GetComponent<gameManager>().RemoveDrone();
                }
            }
        }
    }

    2 references
    public void DroneCheck()
    {
        drones.Clear();
        GameObject[] droneCheck = GameObject.FindGameObjectsWithTag("AssaultDroneBrain");
        for (int i = 0; i < droneCheck.Length; i++)
        {
            if (droneCheck[i].GetComponent<AssaultBotDumb>().health > 0)
            {
                drones.Add(droneCheck[i]);
            }
        }
    }

    9 references
    public float RemapFactor(float factor, float min, float max, float remapMin, float remapMax)
    {
        return (factor - min) / (max - min) * (remapMax - remapMin) + remapMin;
    }
}
```

The drone has a function that allows it to check which drones are still alive, therefore it would be possible to expand the gameplay further by adding more drones throughout the game and the commander would be able to control them without any issues. The “RemapFactor” function simply changes an input value onto a new scale depending on remapMin and remapMax, which for the purpose of fuzzy logic is between zero and one.



```
void Update()
{

    ammoUI.text = currentMag.ToString() +  " / " + reserveBullets.ToString();

    float moveX = -Input.GetAxis("Mouse X") * swayAmount;
    float moveY = -Input.GetAxis("Mouse Y") * swayAmount;

    Vector3 finalPos = new Vector3(moveX, moveY, 0);
    transform.localPosition = Vector3.Lerp(transform.localPosition, finalPos + startPos, Time.deltaTime * smoothAmount);

    transform.Rotate(Vector3.up, recoilRemaining.x);
    transform.Rotate(Vector3.right, recoilRemaining.y, Space.Self);

    recoilRemaining *= 0.9f;

    if (Time.time > recoilResetTimer)
    {
        transform.rotation = Quaternion.Slerp(transform.rotation, Camera.main.transform.rotation, 1 * Time.deltaTime);
    }

    if (Input.GetKey(KeyCode.Mouse0))
    {
        if (currentMag > 0 && !rifleAnim.GetCurrentAnimatorStateInfo(0).IsName("RifleReload"))
        {
            if (Time.time > fireRateTimer)
            {
                bullet.transform.rotation = transform.rotation;

                Instantiate(muzzleFlash, muzzlePoint.transform.position, muzzlePoint.transform.rotation, null);

                Rigidbody instantiatedBullet = Instantiate
                    (bullet, muzzlePoint.transform.position, muzzlePoint.transform.rotation);

                Vector3 bulletVelocity = transform.TransformDirection(Vector3.forward * bulletSpeed);
                Vector2 rand = Random.insideUnitCircle;
                bulletVelocity += new Vector3(rand.x, rand.y, 0) * inaccuracy;

                instantiatedBullet.velocity = bulletVelocity;

                rifleSoundSource.PlayOneShot(shootSound);
                currentMag--;

                recoilRemaining.x = Random.Range(-recoilXAxis, recoilXAxis);
                recoilRemaining.y = Random.Range(-recoilYAxis, recoilYAxis);

                fireRateTimer = Time.time + fireRate;
                recoilResetTimer = recoilResetTime + Time.time;
            }
        }

    }
```

The player’s rifle is script is highly configurable, allowing any number of new weapons to be added or tweaked. The sway on camera movement is achieved by linearly interpolating the local position of the gun based on the mouse input, creating a smooth feeling without using any animation. The fire-rate can be tweaked with a float value, and the recoil is physically added to the rotation of the gun when it is fired, increasing by a random amount within a certain range whilst it is being fired and reducing on update to once again achieve satisfying firing feedback without any animation. The only animation in the whole game is the reload animation, which is also used as a check to ensure the gun cannot be fired whilst reloading.



```
if (Input.GetKeyDown(KeyCode.R))
{
    if (reserveBullets > 0)
    {
        rifleAnim.Play("Base Layer.RifleReload");

        reserveBullets += currentMag;

        if (reserveBullets < 30)
        {
            currentMag = reserveBullets;
            reserveBullets -= currentMag;
        }
        else
        {
            currentMag = magSize;
            reserveBullets -= magSize;
        }

        rifleSoundSource.PlayOneShot(reloadSound);

        if (reserveBullets < 0)
        {
            reserveBullets = 0;
        }
    }
}
```

The reload uses a simple tilting animation and basic equations to give the player back the ammo they had left over in the magazine and to ensure they do not get a full magazine if there is not enough ammo in reserve.

Other scripts, such as for the crosshair, game manager and bullet impacts are simple collision checks and if statements and thus are not worth showcasing.

Critical Evaluation

Overall, I believe my game prototype has achieved the goals I set out in my presentation, as it features both ‘smart’ and ‘dumb’ AI opponents using procedural animation and different AI techniques which are capable of communicating together to increase the difficulty when the commander is alive and reward the player for neutralising that commander by making the remaining AI easier to dispatch. For this purpose, I think I selected the right AI techniques as a finite state machine works similarly to how the dumb AI would work, performing actions based on a binary parameter such as ‘can or cannot see the player’, and fuzzy logic expands on that by making the smart AI less predictable and more reactive to its environment. I believe both implementations could have been improved further, such as by using a single Enum for the state machine and making use of the animator to transition between different states, and giving the fuzzy logic AI many more inputs and actions to make it even less predictable and give it more ways to defend itself and attack the player, thus creating more gameplay opportunities. Finally, it would be an interesting task to attempt to use machine learning for the commander, possibly by teaching it against a ‘simulation’ of a player attacking in combination of actual gameplay tests. This would make the AI truly ‘smart’ as it would be capable of actually learning how to react to threats and act accordingly instead of using pre-programmed responses.

External Assets

Standard Assets (FPS Controller):

https://assetstore.unity.com/packages/essentials/asset-packs/standard-assets-for-unity-2018-4-32351

Animation Rigging:

https://docs.unity3d.com/Packages/com.unity.animation.rigging@0.2/manual/index.html

Tracers & Impact Effects:

https://www.davidstenfors.com/#!/tutorials/unity-tutorials/tracer-rounds-tutorial

Pro Builder (3D Modelling within Unity):

https://unity3d.com/unity/features/worldbuilding/probuilder

Gun Sounds:

https://assetstore.unity.com/packages/audio/sound-fx/weapons/weapon-soldier-sounds-pack-29662