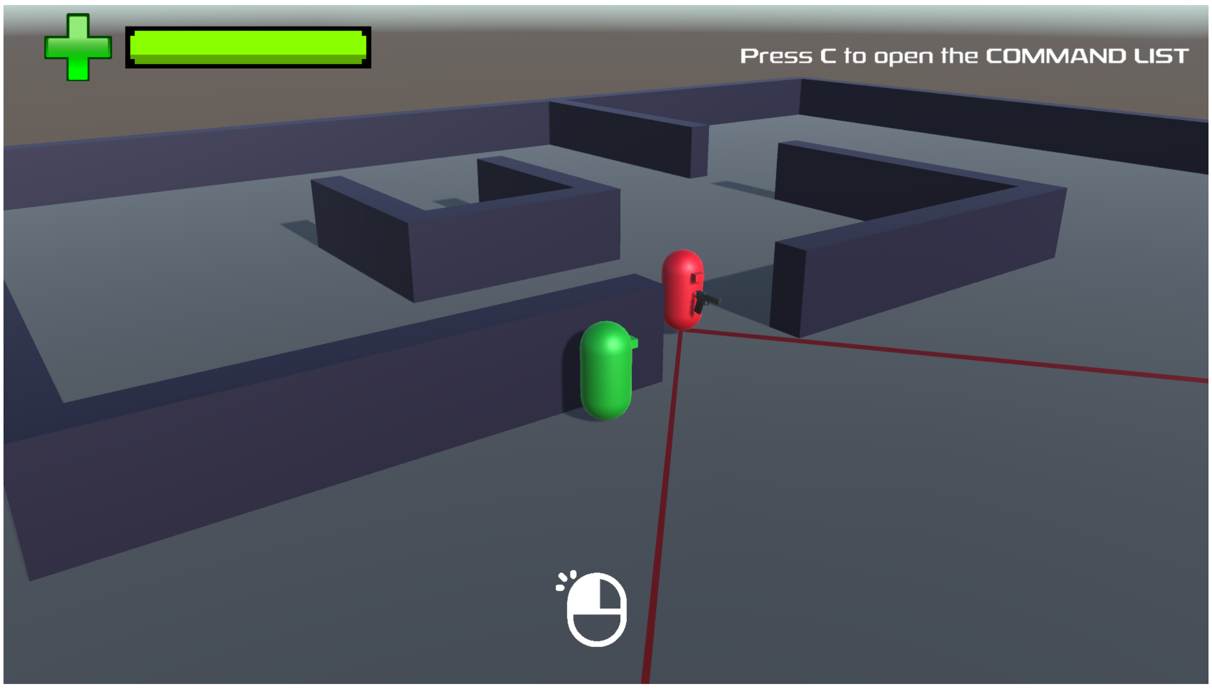
Università degli studi di Milano

Dipartimento di Informatica

AIVG Project



Developed by MARCO GRANDE
969819
marco.grande@studenti.unimi.it

CONTENTS

INTRODUCTION

This project includes the development of an Artificial Intelligence based on the behaviour of the guards in the Metal Gear series.

In normal conditions the guard follows a patrolling path in the map. In order to protect an object that is placed in the central section, he follows a path that surveils all the access points. From now on we will call this situation the “Normal state”.

The guard can be lured in a specific spot by the player if he decides to knock on a wall. After verifying the source of the sound he proceeds to change state and strictly surveils the object by going to the access point nearest to it. We will call the process of verifying the sound source “Sound state” and the process of strict surveillance “Warning state”.

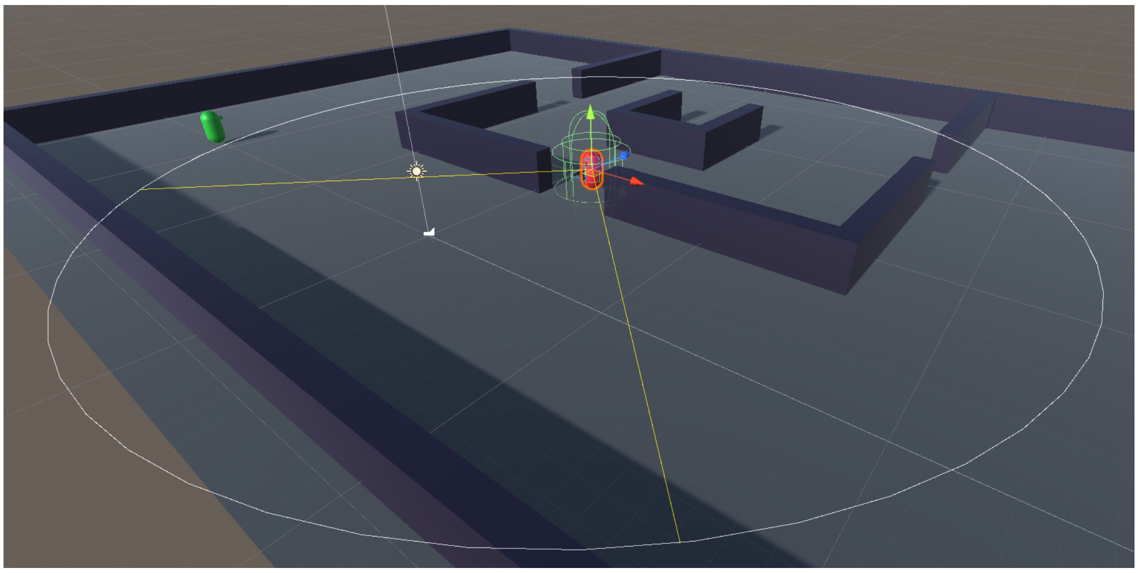
The guard has a field of view (FOV) that is used to detect the player and, depending on the player position, he can decide to shoot or chase him. The FOV takes into consideration the presence of obstruction objects and reacts to the player only if the guard can clearly “see” the intruder. If the player escapes, the guard goes into the Warning state. We will refer to the detection of the player as “Alarm state” and we will consider individually the “Shoot behaviour” and the “Chase behaviour”.

The decision making process of the AI is managed by a Finite State Machine that is combined with a decision tree when managing the Alarm State since, as we will see in further details later, the Shoot behaviour and the Chase behaviour have overlapping conditions. The movement and pathfinding is handled by Unity’s NavMesh agent but with necessary corrections in order to simulate a more believable behaviour.

FIELD OF VIEW

Function

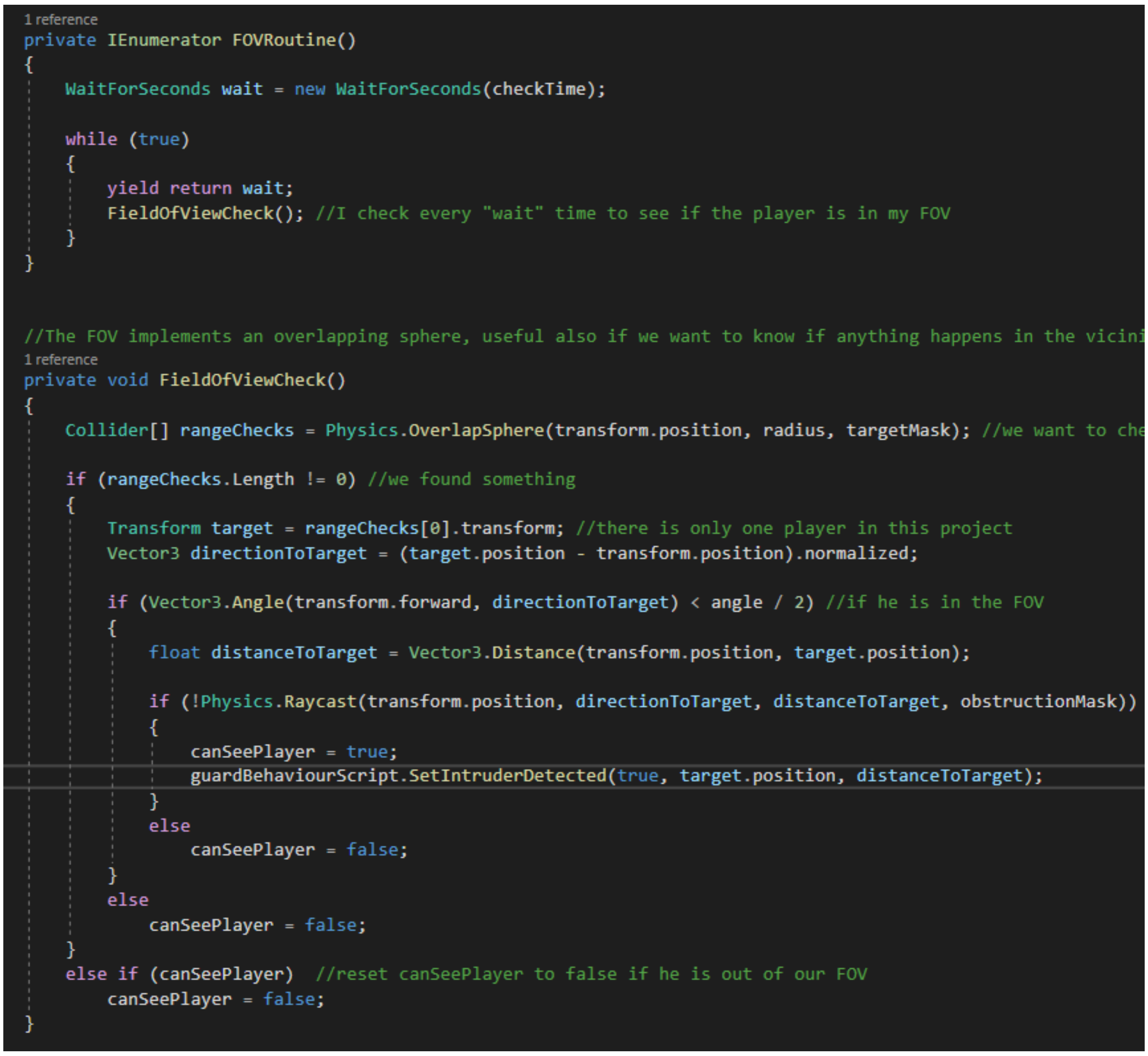
The Field of View represents the guard's vision and its output is used by the Finite State Machine to spot the player and change the internal state to the Alarm state.



Implementation

In order to implement the FOV I create a sphere around the guard that checks for objects in a specific layer (the player's layer). Then I proceed to find the versor that connects the guard to the target and I use it to check if the player is in the cone of vision.
After establishing the presence of the player I check if there aren't any obstacles (objects in the obstruction layer) and eventually send the information about the player to the guard behaviour in order to manage the Alarm state. All this process is executed every fixed amount of time by a coroutine.

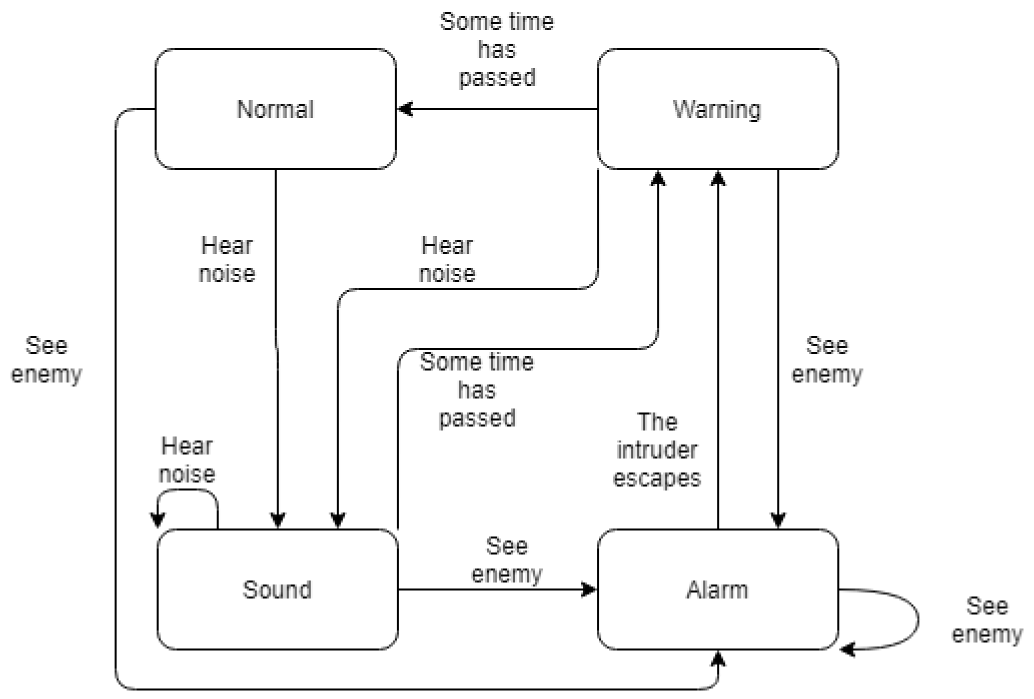
…



```
1 reference
private IEnumerator FOVRoutine()
{
    WaitForSeconds wait = new WaitForSeconds(checkTime);

    while (true)
    {
        yield return wait;
        FieldOfViewCheck(); //I check every "wait" time to see if the player is in my FOV
    }
}


//The FOV implements an overlapping sphere, useful also if we want to know if anything happens in the vicini
1 reference
private void FieldOfViewCheck()
{
    Collider[] rangeChecks = Physics.OverlapSphere(transform.position, radius, targetMask); //we want to che

    if (rangeChecks.Length != 0) //we found something
    {
        Transform target = rangeChecks[0].transform; //there is only one player in this project
        Vector3 directionToTarget = (target.position - transform.position).normalized;

        if (Vector3.Angle(transform.forward, directionToTarget) < angle / 2) //if he is in the FOV
        {
            float distanceToTarget = Vector3.Distance(transform.position, target.position);

            if (!Physics.Raycast(transform.position, directionToTarget, distanceToTarget, obstructionMask))
            {
                canSeePlayer = true;
                guardBehaviourScript.SetIntruderDetected(true, target.position, distanceToTarget);
            }
            else
                canSeePlayer = false;
        }
        else
            canSeePlayer = false;
    }
    else if (canSeePlayer)  //reset canSeePlayer to false if he is out of our FOV
        canSeePlayer = false;
}
```

FINITE STATE MACHINE

Architecture

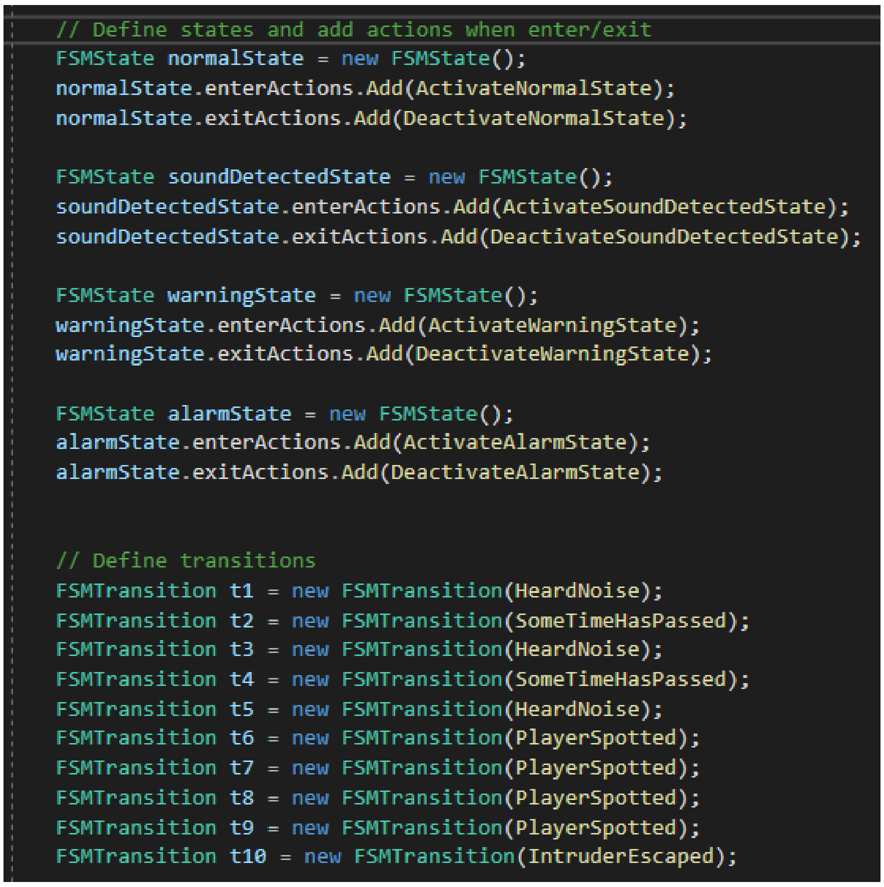


Even if we have four states from the point of view of the FSM structure, we still have five behaviours since the Alarm state contains the Shoot behaviour and the Chase behaviour (that will be chosen at runtime by the Decision Tree).

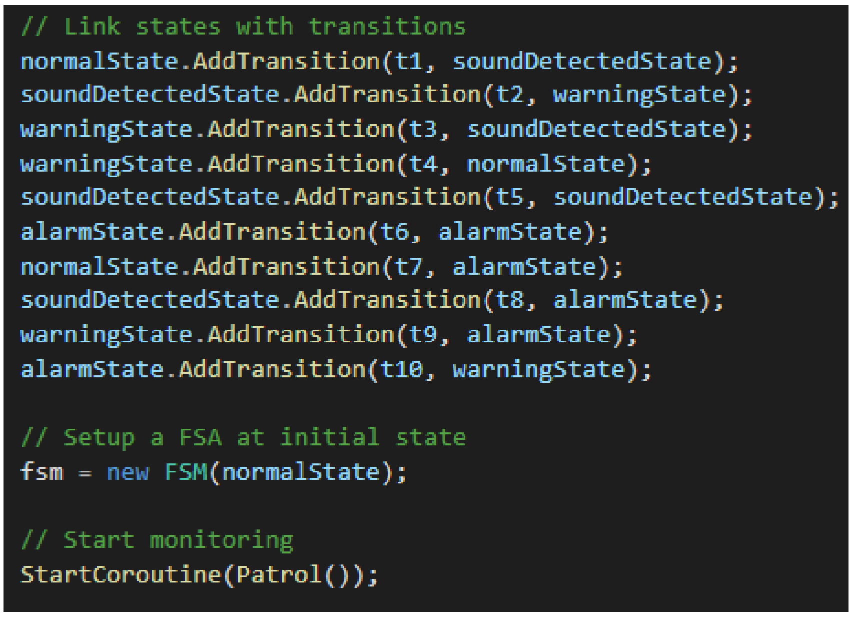
Implementation

For the development of this project I used the FSM data structure that was presented during the course and I adapted it considering the specific implementation.

I decided to divide different behaviours in different scripts, so the FSM job is to activate and deactivate the desired behaviour scripts by using the Enter and Exit actions.



```
// Define states and add actions when enter/exit
FSMState normalState = new FSMState();
normalState.enterActions.Add(ActivateNormalState);
normalState.exitActions.Add(DeactivateNormalState);

FSMState soundDetectedState = new FSMState();
soundDetectedState.enterActions.Add(ActivateSoundDetectedState);
soundDetectedState.exitActions.Add(DeactivateSoundDetectedState);

FSMState warningState = new FSMState();
warningState.enterActions.Add(ActivateWarningState);
warningState.exitActions.Add(DeactivateWarningState);

FSMState alarmState = new FSMState();
alarmState.enterActions.Add(ActivateAlarmState);
alarmState.exitActions.Add(DeactivateAlarmState);


// Define transitions
FSMTransition t1 = new FSMTransition(HeardNoise);
FSMTransition t2 = new FSMTransition(SomeTimeHasPassed);
FSMTransition t3 = new FSMTransition(HeardNoise);
FSMTransition t4 = new FSMTransition(SomeTimeHasPassed);
FSMTransition t5 = new FSMTransition(HeardNoise);
FSMTransition t6 = new FSMTransition(PlayerSpotted);
FSMTransition t7 = new FSMTransition(PlayerSpotted);
FSMTransition t8 = new FSMTransition(PlayerSpotted);
FSMTransition t9 = new FSMTransition(PlayerSpotted);
FSMTransition t10 = new FSMTransition(IntruderEscaped);
```

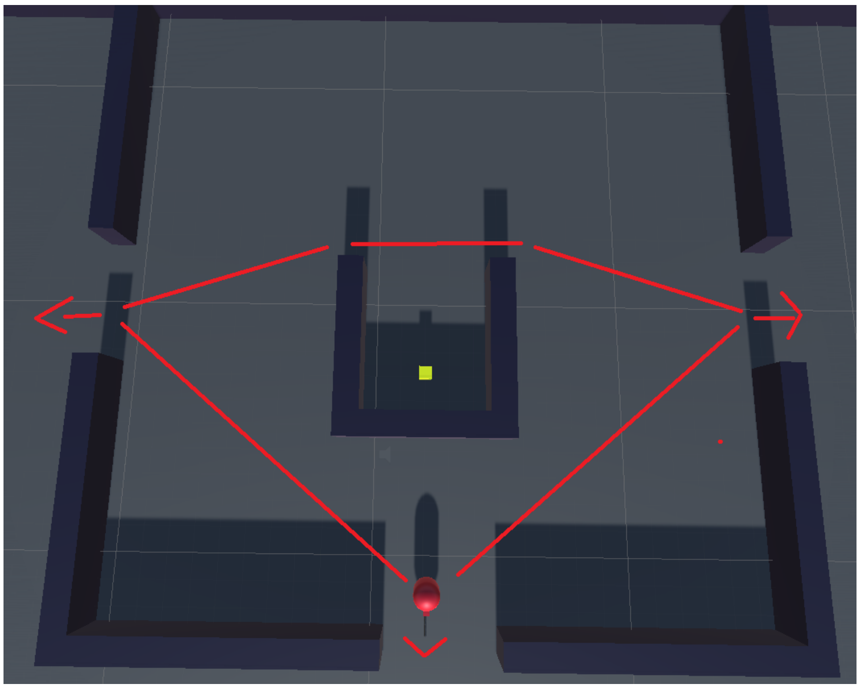


```
// Link states with transitions
normalState.AddTransition(t1, soundDetectedState);
soundDetectedState.AddTransition(t2, warningState);
warningState.AddTransition(t3, soundDetectedState);
warningState.AddTransition(t4, normalState);
soundDetectedState.AddTransition(t5, soundDetectedState);
alarmState.AddTransition(t6, alarmState);
normalState.AddTransition(t7, alarmState);
soundDetectedState.AddTransition(t8, alarmState);
warningState.AddTransition(t9, alarmState);
alarmState.AddTransition(t10, warningState);

// Setup a FSA at initial state
fsm = new FSM(normalState);

// Start monitoring
StartCoroutine(Patrol());
```

NORMAL STATE

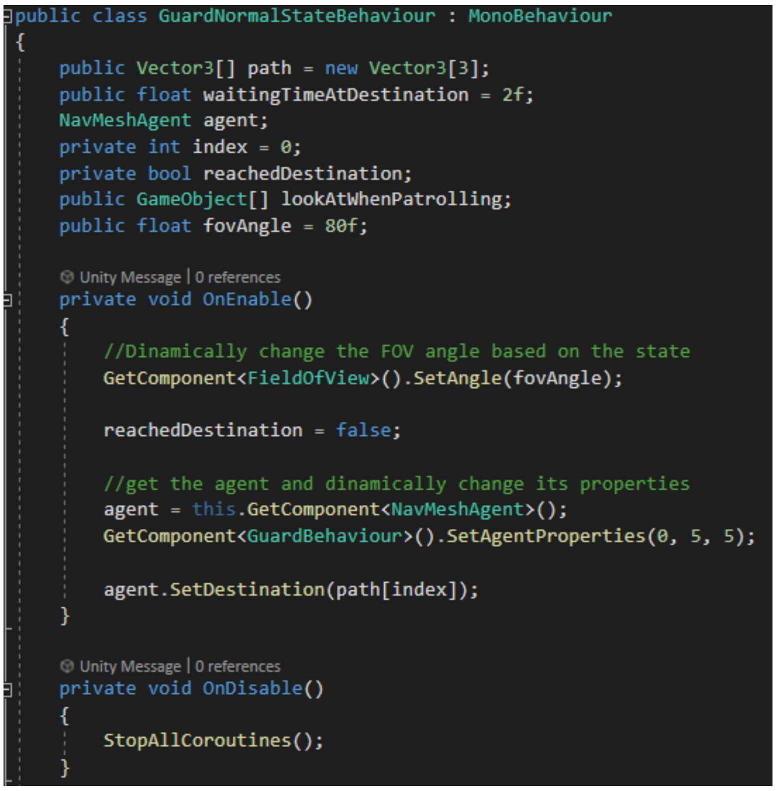
Function

The Normal state behaviour consists in visiting three points in the map using the NavMesh and rotating to a specific direction after arriving at each destination. The points to look at when arriving are defined by three empty game objects.

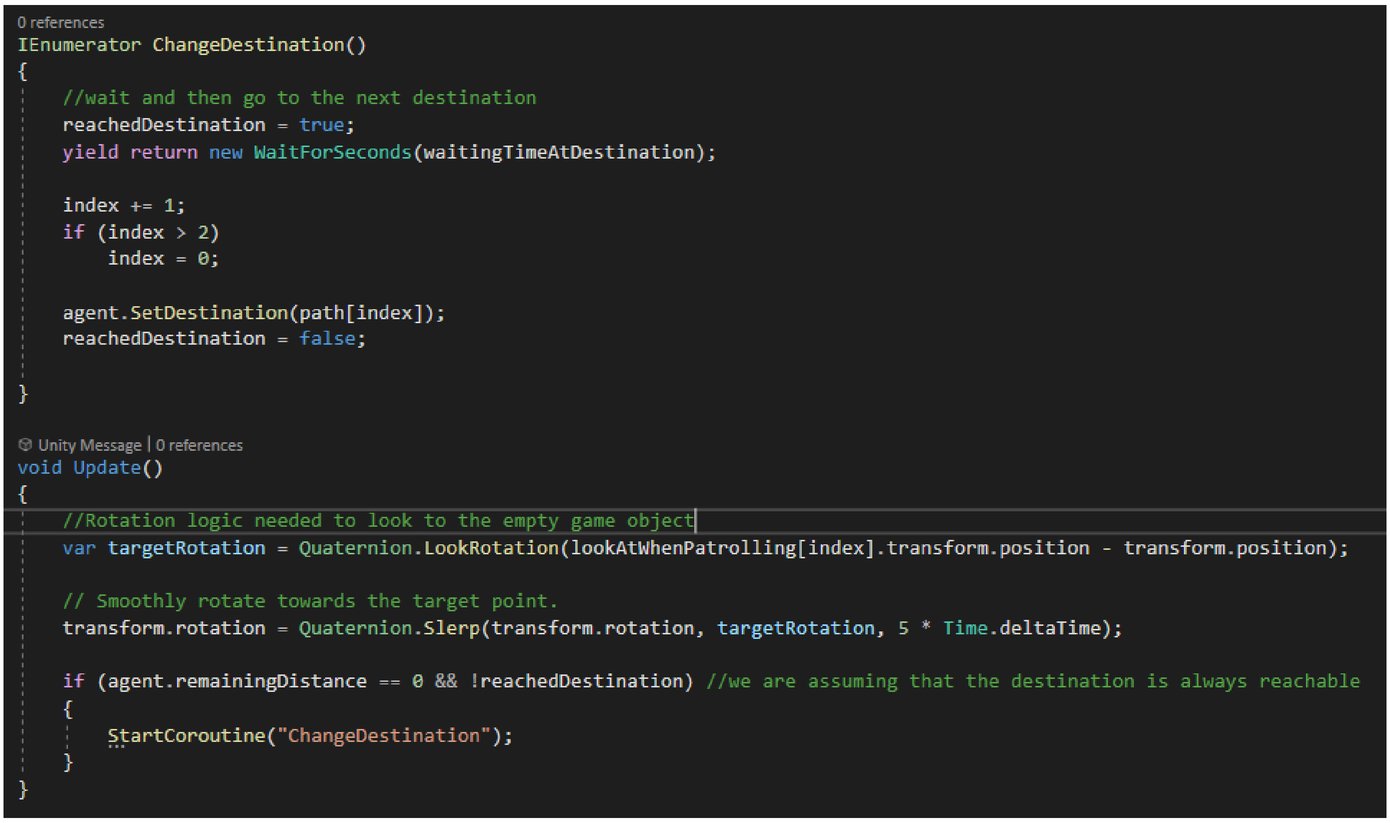


Implementation

The implementation sets a destination for the agent that is taken from an array that contains the three targets. A coroutine establishes how much time the agent needs to wait at every destination and increments the index in order to take the next target. The rotation is set by using a spherical interpolation between the quaternion of the guard rotation and the rotation of the vector connecting the guard and the empty game object. We don’t just interpolate after arrival since we want the guard to look near that direction even while he moves, otherwise the guard would suddenly turn when arrived at the destination.



```
public class GuardNormalStateBehaviour : MonoBehaviour
{
    public Vector3[] path = new Vector3[3];
    public float waitingTimeAtDestination = 2f;
    NavMeshAgent agent;
    private int index = 0;
    private bool reachedDestination;
    public GameObject[] lookAtWhenPatrolling;
    public float fovAngle = 80f;

    Unity Message | 0 references
    private void OnEnable()
    {
        //Dinamically change the FOV angle based on the state
        GetComponent<FieldOfView>().SetAngle(fovAngle);

        reachedDestination = false;

        //get the agent and dinamically change its properties
        agent = this.GetComponent<NavMeshAgent>();
        GetComponent<GuardBehaviour>().SetAgentProperties(0, 5, 5);

        agent.SetDestination(path[index]);
    }

    Unity Message | 0 references
    private void OnDisable()
    {
        StopAllCoroutines();
    }
```



```
0 references
IEnumerator ChangeDestination()
{
    //wait and then go to the next destination
    reachedDestination = true;
    yield return new WaitForSeconds(waitingTimeAtDestination);

    index += 1;
    if (index > 2)
        index = 0;

    agent.SetDestination(path[index]);
    reachedDestination = false;

}

Unity Message | 0 references
void Update()
{
    //Rotation logic needed to look to the empty game object
    var targetRotation = Quaternion.LookRotation(lookAtWhenPatrolling[index].transform.position - transform.position);

    // Smoothly rotate towards the target point.
    transform.rotation = Quaternion.Slerp(transform.rotation, targetRotation, 5 * Time.deltaTime);

    if (agent.remainingDistance == 0 && !reachedDestination) //we are assuming that the destination is always reachable
    {
        StartCoroutine("ChangeDestination");
    }
}
```

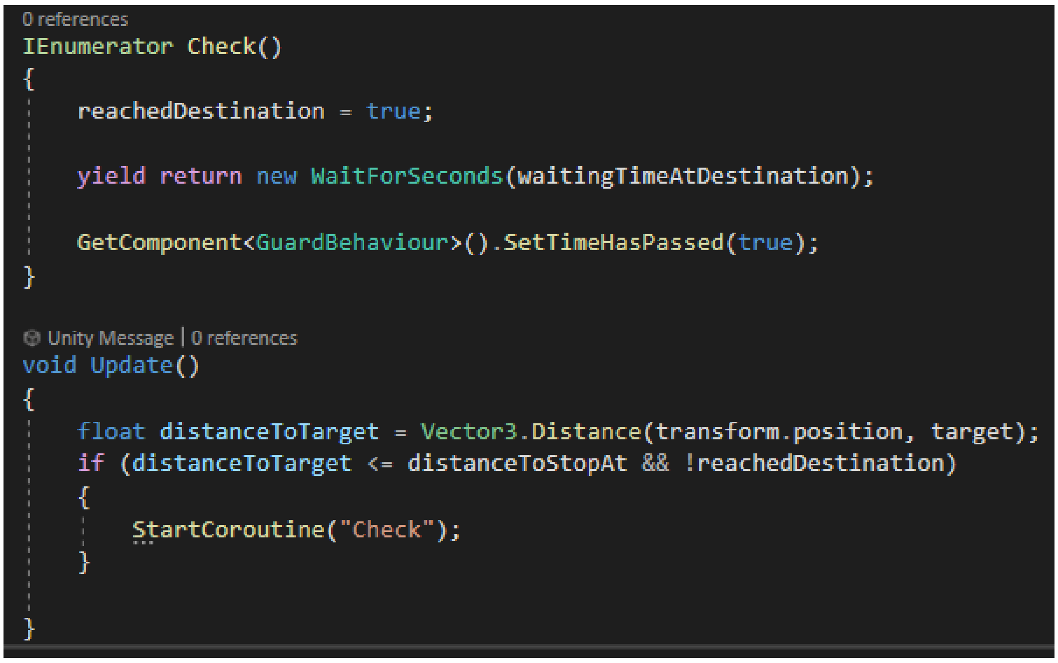
SOUND STATE

Function

This state consists in the guard visiting the sound source point, waiting a bit and then changing state. This state can loop into itself since, if the guard hears a new sound, he needs to become interested in the new source.

Implementation

The sound is generated by the player knocking on the wall: when the player is near the wall he can interact with it. The interaction generates a sphere collider that checks if the guard is in range, if that's the case then the guard goes into the Sound state.
The Sound state simply changes the guard properties and then sets a new target for the agent. A coroutine calls a function that enables the transition to the Warning state after some time has passed.

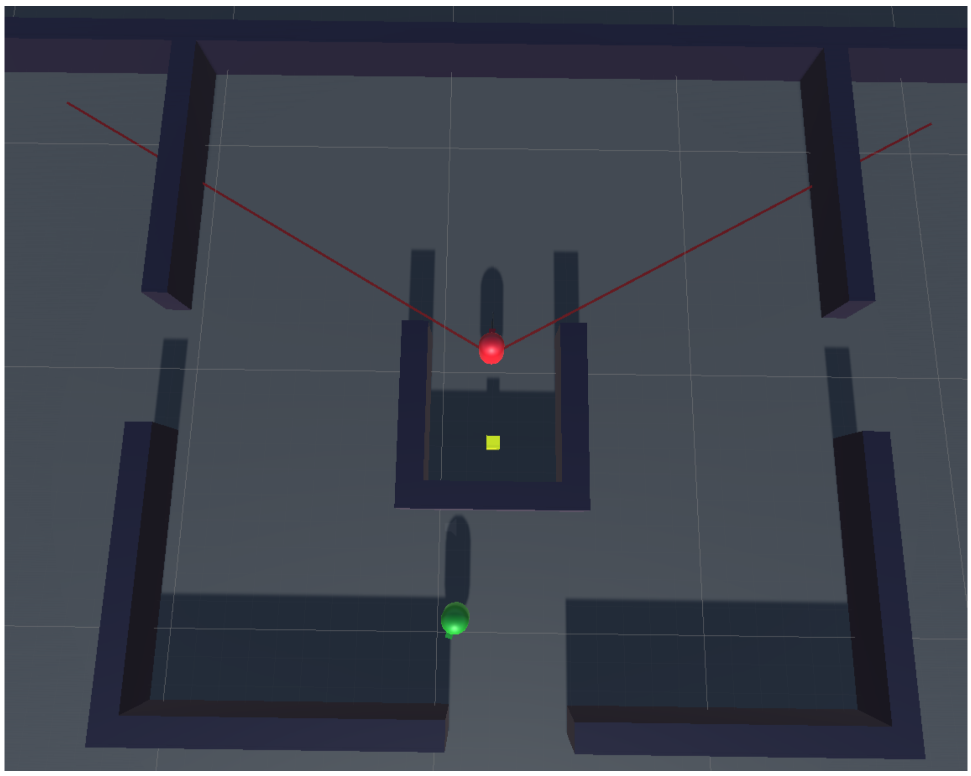


```
0 references
IEnumerator Check()
{
    reachedDestination = true;

    yield return new WaitForSeconds(waitingTimeAtDestination);

    GetComponent<GuardBehaviour>().SetTimeHasPassed(true);
}

Unity Message | 0 references
void Update()
{
    float distanceToTarget = Vector3.Distance(transform.position, target);
    if (distanceToTarget <= distanceToStopAt && !reachedDestination)
    {
        StartCoroutine("Check");
    }

}
```

WARNING STATE

Function

This state behaviour is similar to the Sound state. The Guard goes to a specific point, turns to watch in the desired direction (like in the Normal state), waits some time and then goes to the Normal State.


.

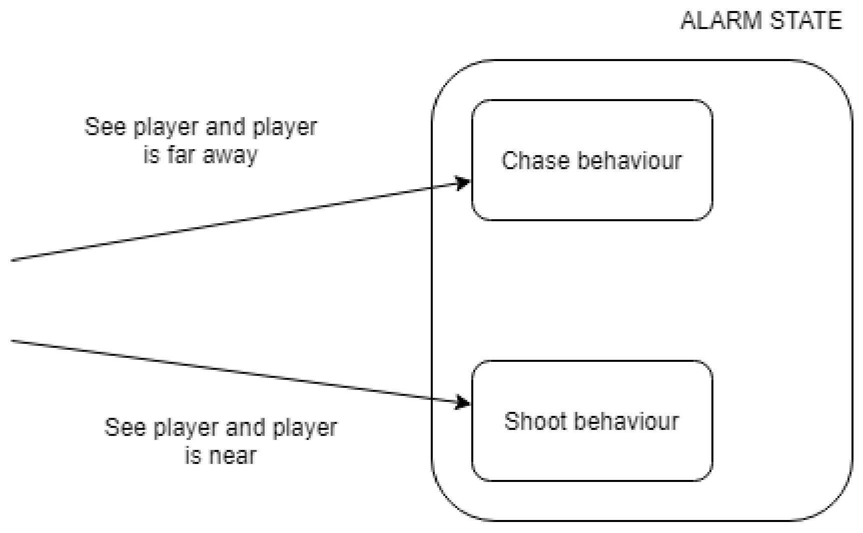
Implementation

The implementation is a mix of the Normal state and the Sound state. After setting the state specific properties for the agent and the FOV, we set the agent's destination, then we turn towards an empty game object and with a coroutine we make the guard wait for a fixed amount of time.

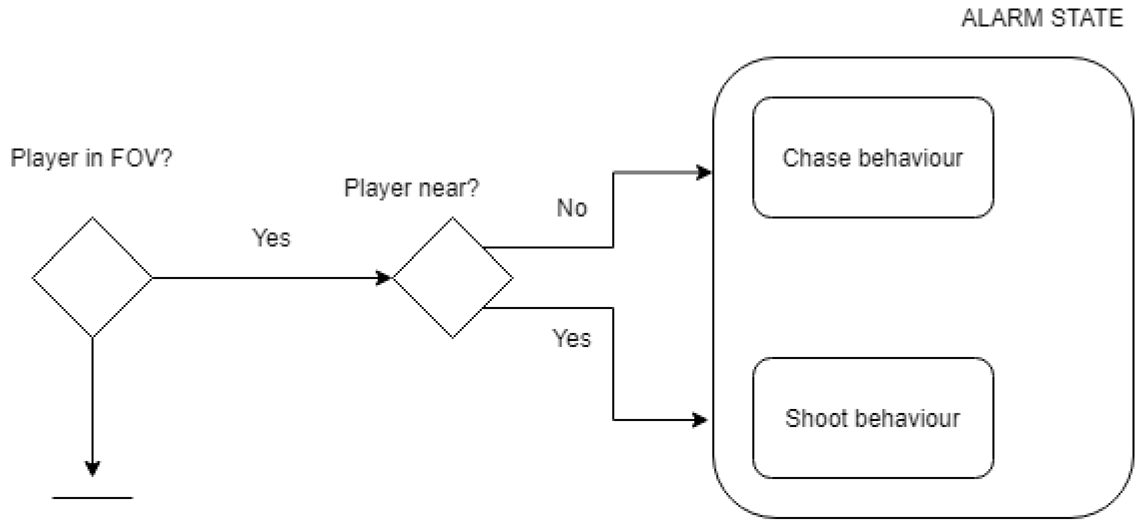
ALARM STATE AND DECISION TREE

Architecture

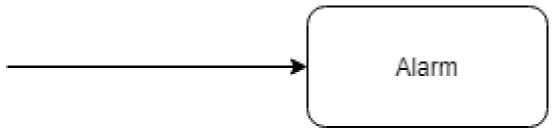
The Alarm state contains the Chase behaviour and the Shoot behaviour.
The Chase behaviour is activated when the player is in the cone of vision but is far from the guard, while the Shoot behaviour is activated when the player is in the cone of vision and is near the guard.


ALARM STATE
See player and player is far away
Chase behaviour
See player and player is near
Shoot behaviour

Since the conditions overlap I decided to use a decision tree to replace the transitions.

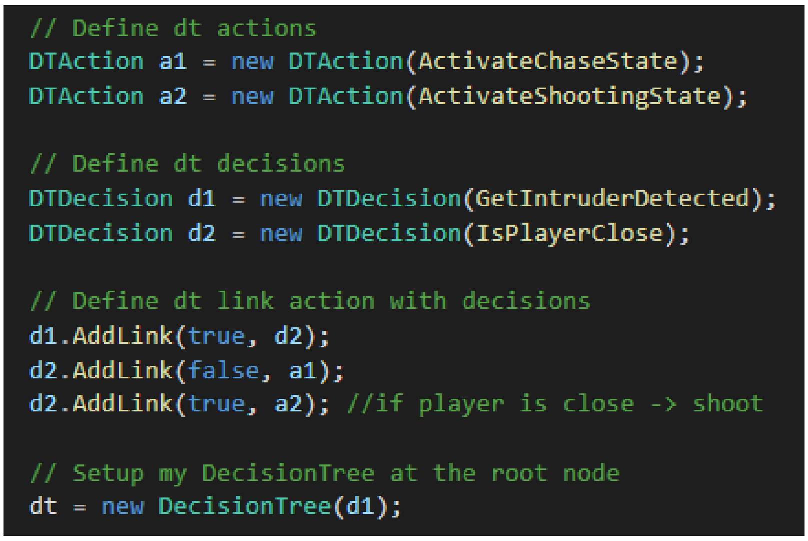


While this is the internal representation, from the Finite State Machine point of view we just have a transition that runs the decision tree and then it's on the DT to decide which behaviour to activate.



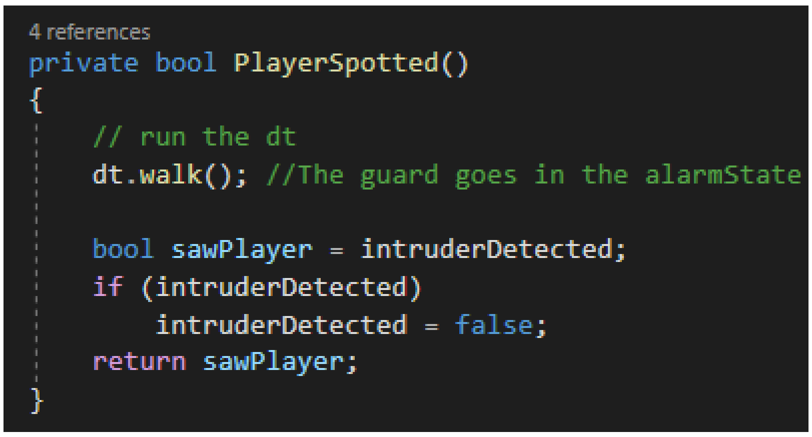
Implementation

The data structure for the decision tree is based on what we have seen during the lessons and it is implemented in conjunction with the FSM.



```
// Define dt actions
DTAction a1 = new DTAction(ActivateChaseState);
DTAction a2 = new DTAction(ActivateShootingState);

// Define dt decisions
DTDecision d1 = new DTDecision(GetIntruderDetected);
DTDecision d2 = new DTDecision(IsPlayerClose);

// Define dt link action with decisions
d1.AddLink(true, d2);
d2.AddLink(false, a1);
d2.AddLink(true, a2); //if player is close -> shoot

// Setup my DecisionTree at the root node
dt = new DecisionTree(d1);
```

PlayerSpotted() is the transition that goes into the Alarm state and runs the DT.



```
4 references
private bool PlayerSpotted()
{
    // run the dt
    dt.walk(); //The guard goes in the alarmState

    bool sawPlayer = intruderDetected;
    if (intruderDetected)
        intruderDetected = false;
    return sawPlayer;
}
```

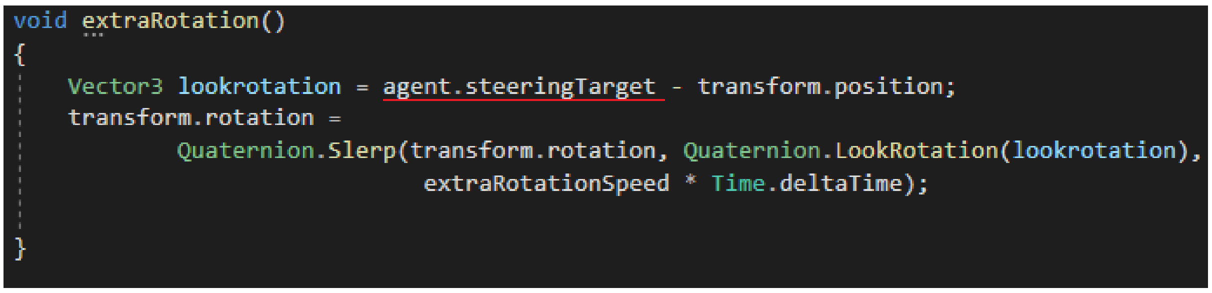
CHASE BEHAVIOUR

Function

In the Chase behaviour it is required to have faster movement because we want to keep the player in the cone of vision otherwise he will escape. In this state the guard moves faster, has a bigger cone of vision and has extra rotation speed since the geometry of the cone would let the player easily escape if he goes in the guard direction

Implementation

The implementation follows the player by setting the Navmesh target similarly to the Sound state and adds the extra rotation in a similar way to how we rotated to look at the empty objects in the Normal state. However this time the extra rotation considers the target.

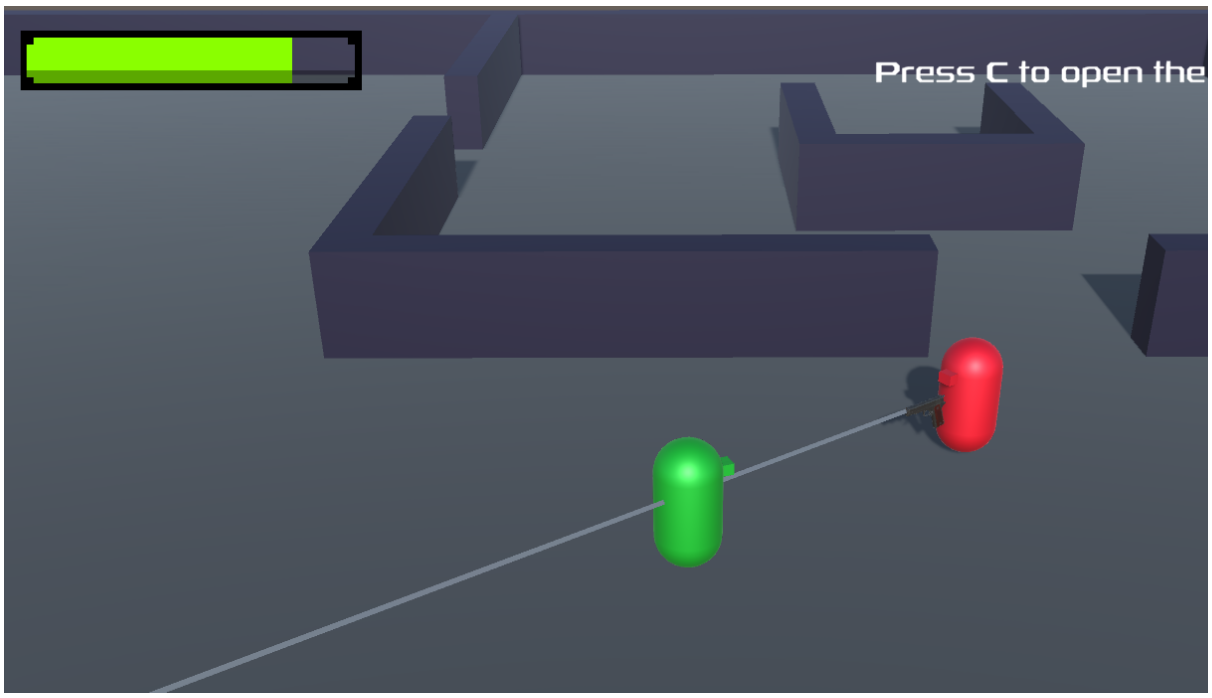


```
void extraRotation()
{
    Vector3 lookrotation = agent.steeringTarget - transform.position;
    transform.rotation =
            Quaternion.Slerp(transform.rotation, Quaternion.LookRotation(lookrotation),
                             extraRotationSpeed * Time.deltaTime);

}
```

SHOOT BEHAVIOUR

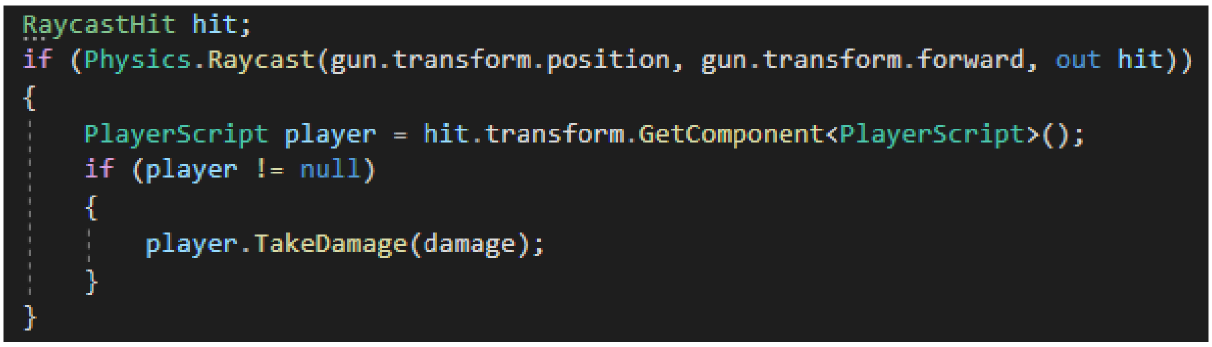
Function

When the player is in the cone of vision and in a certain range from the guard we switch to the Shoot behaviour. In this behaviour the guard aims at the player if he is still, or predicts his trajectory if he is moving, and then he shoots.



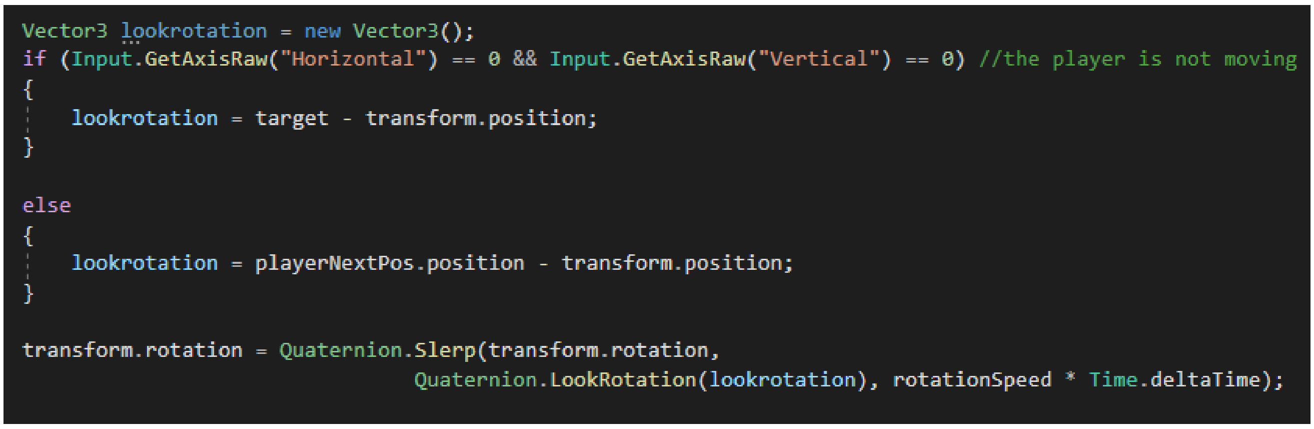
Implementation

In order to simulate the shooting we project a raycast and check if it hits the player and, eventually, change the player's health.



```
RaycastHit hit;
if (Physics.Raycast(gun.transform.position, gun.transform.forward, out hit))
{
    PlayerScript player = hit.transform.GetComponent<PlayerScript>();
    if (player != null)
    {
        player.TakeDamage(damage);
    }
}
```

For the prediction part I put an empty object in front of the player as a child. If the player is not moving the guard aims at the player, otherwise he aims at the game object.



```
Vector3 lookrotation = new Vector3();
if (Input.GetAxisRaw("Horizontal") == 0 && Input.GetAxisRaw("Vertical") == 0) //the player is not moving
{
    lookrotation = target - transform.position;
}

else
{
    lookrotation = playerNextPos.position - transform.position;
}

transform.rotation = Quaternion.Slerp(transform.rotation,
                                      Quaternion.LookRotation(lookrotation), rotationSpeed * Time.deltaTime);
```