UNIVERSITÀ
DEGLI STUDI
DI MILANO

LA STATALE

PONG

Playlab fOr inNovation in Games

Artificial Intelligence for Videogames a.y. 2018/2019

Andrea Carrarini (927539) – andrea.carrarini@studenti.unimi.it

CONTENT INDEX

1. INTRODUCTION

This is the university project I worked on during the second semester of the 2018/2019 academic year for the "**Artificial Intelligence for Video Games**" exam.

This project represents the natural evolution of "Hypogeum" , a multiplayer driving / shooting / battle car game made with Unity for PC & Mac.

In the game, four teams of two players face each other in an enormous arena (called, in fact, Hypogeum), trying to defeat their enemies and be the last standing. Each team belongs to one of the four existing factions and is composed of two players from the same species, that represent the champions. The battles are fought on cars, with one player as driver and one as shooter, each of them equipped with a faction-specific weapon.

The teams, in addition to the other players, have to pay attention to the surrounding environment: different traps and NPCs could interfere with the battle and increase the difficulty of the match. However, the heroes are not alone: each team has its supporters in the audience and, through thrilling actions, can increase their excitement up to push them to help their favorites with useful power-ups.

Hypogeum participated to the **New Game Designer 2019** and won the "Best Multiplayer Game" and "EDI Special Award" awards.

2. CHALLENGE & PURPOSE

The challenge and the purpose of this project are to develop a believable AI for enemies' cars for several reasons:

- Fill the lobbies if there aren't enough players;
- Substitute a disconnected driver player in order to let the shooter one continue his match;
- Play in a training mode with "bots" where players can explore the map and develop new skills and strategies;

3. IMPLEMENTATION

3.1. MOVEMENT

To develop the movement of the cars I used the solutions presented by **Professor Dario Maggiorini** during the Artificial Intelligence for Video Games course, in particular the classes used are (slightly modified to fit at best with the underlying game):

- **MovementStatus**

  Simple class representing the actual movement status of the object.

- **MovementBehaviour**

  Abstract class that works as an interface for the other behaviours.

- **DragBehaviour**

  Adds linear and angular drag.
- **FleeBehaviour**

  Responsible of the flee from an enemy.
- **SeekBehaviour**

  Responsible of chasing down something (an Object or a Player).
- **AvoidBehaviourVolume**

  Takes care to avoid incoming obstacles. (I modified the sight range to be equal to the actual speed of the car, in order to enable turns even with very near obstacles when the car is almost stationary)
- **DDelegatedSteering**

  Blends everything together (Modification explained later).

3.2. DECISION MAKING

In order to develop a simple, effective and believable Decision Making system I adopted and combined 2 different techniques: **Finite States Machines** and **Behaviour Trees**.

For both I used again the code provided by professor Dario Maggiorini during the course.

3.2.1. FINITE STATES MACHINE

Since the actions a driver player can make are:

- Chase / Flee from the opponent car
- Pick an Instinct or Reason Coin
- Execute a Jump to increase the Audience Hype
- Moving around the map

I thought that a FSM was the technique that fitted best to describe the status of a car.

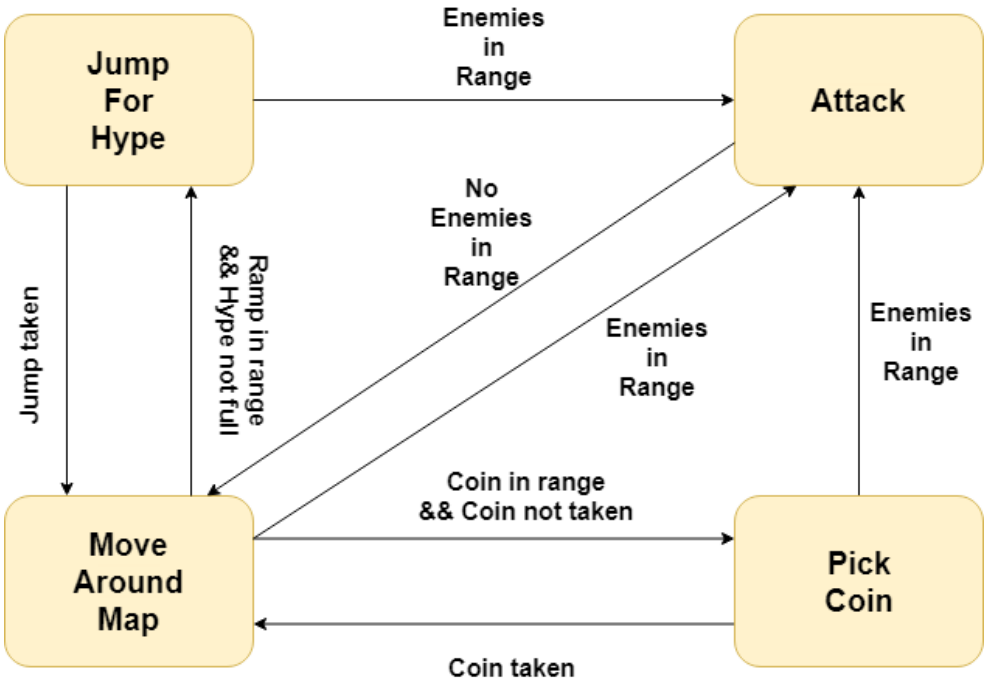

Figure 1: Finite States Machine

The FSM I built is very simple: there are 4 states and 8 transitions, with 2 states that are more common than the others, in particular **Move around map** is the base state where the FSM starts, and the natural return state when the actions in the other states are completed.

Attack, on the contrary, has the highest priority, in fact if the car detects an enemy in its range, no matter what it's doing, it leaves immediately the current state and enters the Attack state.

Pick Coin refers to the game mechanic that allows players to choose between a Reason and an Instinct coin, placed at the top of the ramps, that when taken boost different statistics of the car.

Jump For Hype is another game mechanic that is about doing stylish moves, such as jumps and drifts, to raise the Hype of the Audience, and when the Hype bar is full, it will throw a power-up to the team.

To cycle between the states I used a coroutine that awakes every 0.5 seconds and checks if in the current state there are Transitions to be fired.

3.2.2. BEHAVIOUR TREES

Each state of the FSM has an associated BT describing the actions the car has to do to reach its goal and to act in a believable way.

Since the code provided from professor Dario Maggiorini is coroutine compliant and BTs keep track of the "state" they arrived, the solution FSM + BTs required some work to actually run correctly.

In fact just stopping and restarting the specific BT's coroutine wasn't enough, so I figured out that the whenever the car leaves a state and enters in a new one in the FSM, the old

state's BT must be rebuilt to erase the memory and let the car executing it again, when it's needed, from the beginning.

```
public CRBT.BehaviorTree PickCoinBTBuilder()
{
    CRBT.BTAction a3 = new CRBT.BTAction( GetRampPads );
    CRBT.BTAction a4 = new CRBT.BTAction( MoveToRamp );
    CRBT.BTAction a5 = new CRBT.BTAction( MoveToMidPad );
    CRBT.BTAction a6 = new CRBT.BTAction( MoveToCoin );

    CRBT.BTCondition c4 = new CRBT.BTCondition( NearestPadFound );
    CRBT.BTCondition c2 = new CRBT.BTCondition( DistanceFromBasePad );
    CRBT.BTCondition c6 = new CRBT.BTCondition( DistanceFromMidPad );

    CRBT.BTSelector sel2 = new CRBT.BTSelector( new CRBT.IBTTask[] { c4, a3 } );

    CRBT.BTDecoratorUntilFail uf1 = new CRBT.BTDecoratorUntilFail( c2 );
    CRBT.BTDecoratorUntilFail uf2 = new CRBT.BTDecoratorUntilFail( c6 );

    CRBT.BTSequence seq2 = new CRBT.BTSequence( new CRBT.IBTTask[] { sel2, a4, uf1, a5, uf2, a6 } );

    return new CRBT.BehaviorTree( seq2 );
}
```

Figure 2: example of the creation of a Behaviour Tree

```
public void PickCoinStartCoroutine()
{
    if ( resetPickCoinBT )
    {
        PickCoinBT = PickCoinBTBuilder();
        resetPickCoinBT = false;
    }

    pickCoinCR = StartCoroutine( PickCoinLauncherCR() );
}
```

Figure 3: method starting the coroutine specific to the BT

```
public void StopPickCoinBT()
{
    StopCoroutine( pickCoinCR );
    pickCoinCR = null;
    seekBehaviour.destination = null;
    resetPickCoinBT = true;

    foreach ( GameObject go in GameObject.FindGameObjectsWithTag( "ramp" ) )
    {
        // Re-enabling raycast detection
        go.layer = LayerMask.NameToLayer( "Default" );
    }
}
```

Figure 4: method stopping the coroutine and setting the boolean value resetPickCoinBT to true, in order to build it back again

3.2.2.1. MOVE AROUND THE MAP

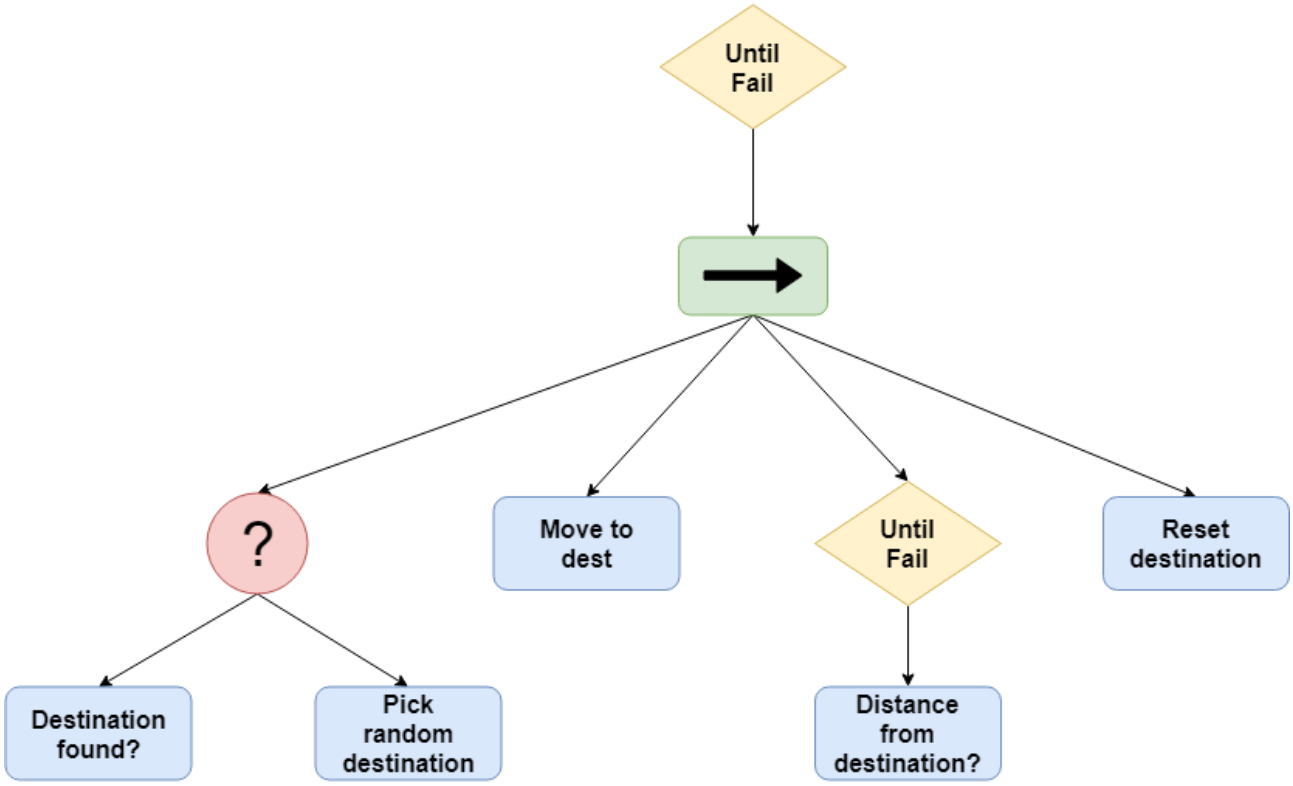

Figure 5: Move around the map BT scheme

Since all the BTs are launched in the FSM states entry actions and Move around the map must be repeated until another state is entered, except for **Destination found?** condition, the other tasks cannot return false, making the BT a loop, thanks to the Until Fail at the top.

Basically I created a prefab named **MoveAroundMapDestination** and I instantiate the GameObject inside the **PickRandomDestination** at the beginning of the match, each time when entering the state and each time it is reached by the car.

Destroys happen when the state is left and when the previous is reached, before instantiating the new one.

Move to dest is just about setting the destination in **SeekBehaviour**, while **Reset destination** just set a Boolean to true, telling the system that another random destination is needed.

Destination found? simply checks if a field is null, while **Distance from destination?** checks if distance from MoveAroundMapDestination is greater than a fixed value, returning false, and so satisfying the Until Fail, only if the car is really near the GameObject.

The purpose of this state is to simulate a player behavior when he/she doesn't see enemies nearby, so tries to explore the map and search for them.

```
public bool PickRandomDestination()
{
    if ( !destination )
    {
        float randomX, randomZ;

        if ( Random.value < 0.5 )
        {
            randomX = Random.value * -250;
        }
        else randomX = Random.value * 250;

        if ( Random.value < 0.5 )
        {
            randomZ = Random.value * -230;
        }
        else randomZ = Random.value * 230;

        GameObject go = GB.LoadDestinationPlaceholder();
        destination = Instantiate( go, new Vector3( randomX, -6, randomZ ), new Quaternion() );
    }
    return true;
}
```

Figure 6: PickRandomDestination() method

3.2.2.2. ATTACK

Attack is the simplest one:

My resistance >= his? Just checks if the AI car has more resistance, the stat that regulates the collision damage between the cars, that the enemy one:

if yes the AI car will **Chase** the enemy (just setting the enemy car transform as the destination in **SeekBehaviour**);

if no it will **Keep distance** that sets the enemy car transform as the destination in **FleeBehaviour**.

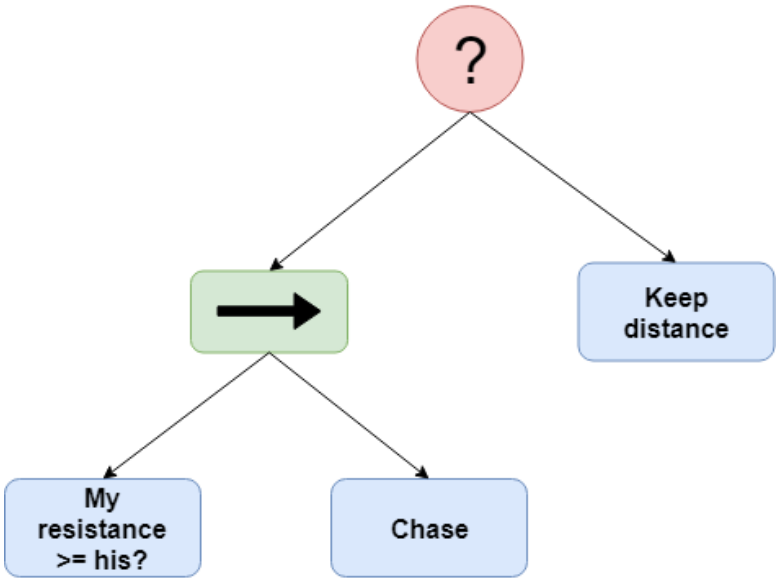

Figure 7: Attack BT

3.2.2.3. PICK COIN

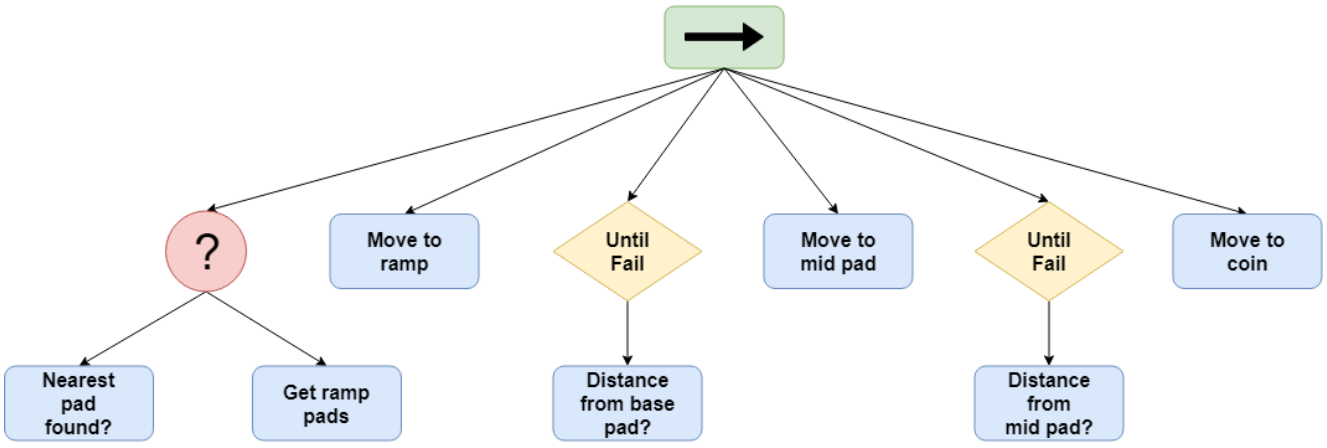

Figure 8: Pick Coin BT

Pick Coin is the most complicated one, so I'll explain step by step.

Nearest pad found? Simply checks if a field is null.

In order to guide the car over the ramp to the coin at the top of them, I positioned 3 pads on each ramp:

- **BasePad** at the base of the ramp;
- **MidPad** approximatively at the mid of the ramp;
- **JumpPad** at the top of the ramp, in the same spots of the coins.

Get ramp pads perform a search for all the GameObjects with the tag "**basePad**" and then extracts the nearest one and then get the relative mid and jump pads.

```
// To know from where the coin is accessible and to move to the base of the correct ramp
public bool GetRampPads()
{
    float minDistance = 100000f;

    foreach ( GameObject go in GameObject.FindGameObjectsWithTag( baseTag ) )
    {
        if ( (go.transform.position - gameObject.transform.position).magnitude < minDistance )
        {
            minDistance = (go.transform.position - gameObject.transform.position).magnitude;
            nearestBasePad = go;
        }
    }

    switch ( nearestBasePad.name )
    {
        case "DragonBasePad":
            nearestMidPad = GameObject.Find( "DragonMidPad" );
            nearestJumpPad = GameObject.Find( "DragonJumpPad" );
            break;
        case "WoodBasePad":
            nearestMidPad = GameObject.Find( "WoodMidPad" );
            nearestJumpPad = GameObject.Find( "WoodJumpPad" );
            break;
        case "StoneBasePad":
            nearestMidPad = GameObject.Find( "StoneMidPad" );
            nearestJumpPad = GameObject.Find( "StoneJumpPad" );
            break;
        case "BigBasePad":
            nearestMidPad = GameObject.Find( "BigMidPad" );
            nearestJumpPad = GameObject.Find( "BigJumpPad" );
            break;
    }

    if ( nearestJumpPad && nearestMidPad )
        return true;
    return false;
}
```

Figure 9: GetRampPads() method showing the names of the pads

Move to ramp just put the basePad transform into SeekBehaviour destination.

On the contrary **Move to mid pad** is a bit more complex:

```
public bool MoveToMidPad()
{
    CarOnRamp = true;
    IgnoreRampRaycast();
    seekBehaviour.destination = nearestMidPad.transform;

    seekBehaviour.brake *= 0.1f;
    seekBehaviour.brakeAt *= 0.1f;

    // To ensure the car doesn't try to reach the mid pad from the ground
    StartCoroutine( WaitForMidPadReached() );

    return true;
}
```

Figure 10: MoveToMidPad() method

CarOnRamp is needed in **DDelegatedBehaviour** to make a distinction in the movement on ramps or not, in fact on ramps, only from base to mid, the car moves forward only if there is at least a wheel colliding with the ramp, avoiding so to fly like a plane.

```
    // Used to adapt the sight range in AvoidBehaviour
    AvoidBehaviourVolume avoidBehaviourVolume = gameObject.GetComponent<AvoidBehaviourVolume>();
    avoidBehaviourVolume.actualSpeed = status.linearSpeed;

    Rigidbody rb = GetComponent<Rigidbody>();

    // To stop moving also in the air if the car is on a ramp
    if ( gameObject.GetComponent<FSMBehaviour>().CarOnRamp )
    {
        // Apply movement only if at least 3 of 4 wheels are on the ground
        int wheelsOnTheGround = 0;

        foreach ( WheelCollider wheelCollider in wheels )
        {
            if ( wheelCollider.isGrounded )
                wheelsOnTheGround += 1;
        }

        if ( wheelsOnTheGround >= 1 )
        {
            rb.MovePosition( rb.position + transform.forward * tangentDelta );
            rb.MoveRotation( rb.rotation * Quaternion.Euler( 0f, rotationDelta, 0f ) );
        }
    }
    else
    {
        Vector3 forwardOnGround = transform.forward;
        forwardOnGround.y = -6.47f;
        rb.MovePosition( rb.position + transform.forward * tangentDelta );
        rb.MoveRotation( rb.rotation * Quaternion.Euler( 0f, rotationDelta, 0f ) );
    }

    status.movementDirection = transform.forward;
}
```

Figure 11: DDelegatedSteering modified

In Figure 11 we can see that when the car is on a ramp a check on the number of wheels on the ground is performed, and the movement is actually updated only if at least 1 wheel is on the ground, to stop the car to fly.

Another little update to the script is in the else block, in fact the Y component of the Vector3 **forwardOnGround** is put always to a number that is near to the real number of the height of the ground (it's in the else clause, so it refers to all the cases that don't involve a ramp), in order to keep it the most possible on the ground.

IgnoreRampRaycast() is mandatory to make the car get on the ramps, otherwise it will avoid them thank to AvoidBehaviour. It simply change the **layer** of the GameObjects with the tag "**ramp**" from **Default** to **Ignore Raycast**.

Brake and **brakeAt** are multiplied by 0.1 not letting the car brake when reaching midPad.

WaitForMidPadReached() is used to be sure that the car didn't fall from the ramp when trying to reach midPad, in that case in fact the car would go in the projection on the ground of midPad position.

```
public IEnumerator WaitForMidPadReached()
{
    yield return new WaitForSeconds( 4f );

    if ( gameObject.transform.position.y < -5f )
    {
        // Putting back brake and brakeAt at default values
        seekBehaviour.brake *= 10f;
        seekBehaviour.brakeAt *= 10f;

        StopPickCoinBT();

        CarOnRamp = false;

        PickCoinStartCoroutine();
    }
}
```

Figure 12: WaitForMidPadReached() method

If the car falls from the ramp the BT is stopped and started again from the beginning.

Move to coin is very similar to Move to mid pad and so it is **WaitForCoinTaken()**.

```
public bool MoveToCoin()
{
    CarOnRamp = false;

    seekBehaviour.destination = nearestJumpPad.transform;

    // To avoid to steer all to right nefore taking the coin
    //gameObject.GetComponent<AvoidBehaviourVolume>().steer = 10;
    gameObject.GetComponent<AvoidBehaviourVolume>().steer = 3;

    gameObject.GetComponent<SeekBehaviour>().gas *= 2f;

    // To check if the coin has been taken, if not, do it again
    StartCoroutine( WaitForCoinTaken() );

    return true;
}
```

Figure 13: MoveToCoin() method

3.2.2.4. JUMP FOR HYPE

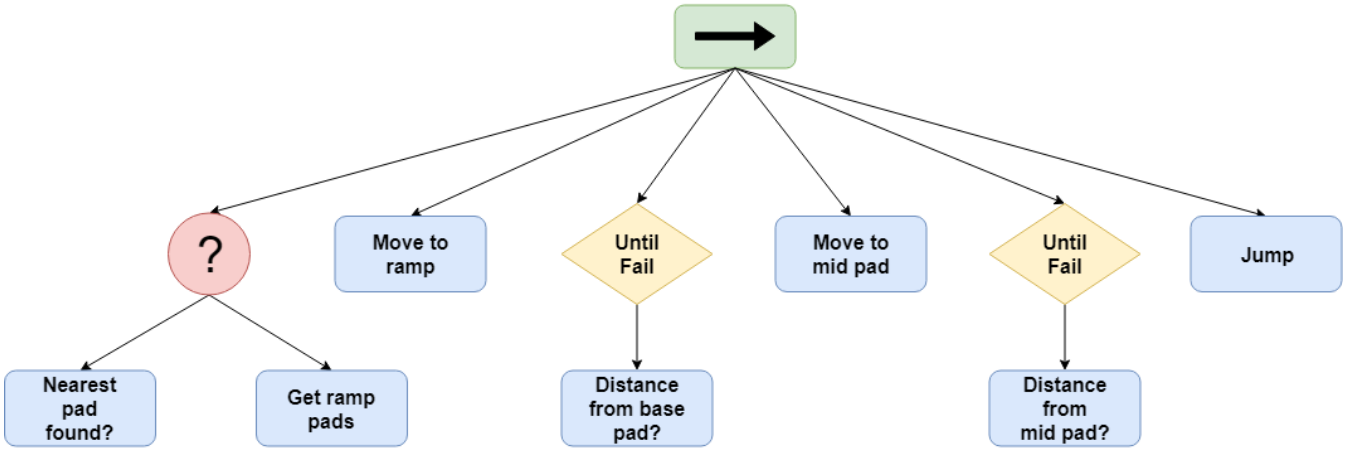

Figure 14: Jump for Hype BT schema

Jump for hype is the same of PickCoin, the only difference is in the last action, which in this case is **Jump**:

In this method I decreased the **steer** to avoid make the car steer and not taking the jump and increased the **gas** value to make the jump more spectacular and credible.

```
public bool Jump()
{
    seekBehaviour.destination = nearestJumpPad.transform;

    CarOnRamp = false;

    // To avoid to steer all to right nefore taking the coin
    gameObject.GetComponent<AvoidBehaviourVolume>().steer = 3;

    // To give the car more speed to take the jump
    gameObject.GetComponent<SeekBehaviour>().gas *= 2f;

    StartCoroutine( WaitForJump() );

    return true;
}
```

Figure 15: Jump() method

And the **WaitForJump()** method, in which I put back to default the values changed in jump() method.

```
public IEnumerator WaitForJump()
{
    yield return new WaitForSeconds( 5f );

    // Putting back brake and brakeAt at default values
    seekBehaviour.brake *= 10f;
    seekBehaviour.brakeAt *= 10f;

    CarOnRamp = false;

    jumpTaken = true;

    // Putting back the steer value to its "default" value
    gameObject.GetComponent<AvoidBehaviourVolume>().steer = 50;

    // Putting back the gas value to its "default" value
    gameObject.GetComponent<SeekBehaviour>().gas *= 0.5f;
}
```

Figure 16: WaitForJump() method

3.3. QUALITY OF LIFE ADDITIONS

In order to avoid the car to being stuck upside down or fall indefinitely beneath the map, I added a script that solve this problems whenever they happen, just like the respawn mechanic already present in the game for the players.

```
public class Reposition : MonoBehaviour
{

    private Transform AICarTransform;

    // a modulo b
    static int MathMod( int a, int b )
    {
        return (Mathf.Abs( a * b ) + a) % b;
    }

    void Start()
    {
        AICarTransform = gameObject.transform;
        StartCoroutine( Respawn() );
    }

    public IEnumerator Respawn()
    {
        while ( true )
        {
            yield return new WaitForSeconds( 4 );

            int zRotation = ( int ) Mathf.Ceil( AICarTransform.rotation.eulerAngles.z );

            if ( MathMod( zRotation, 360 ) < 190 && (MathMod( zRotation, 360 ) > 155) )
            {
                Vector3 respawnPosition = AICarTransform.position;
                AICarTransform.SetPositionAndRotation( new Vector3( respawnPosition.x, 0, respawnPosition.z ), new Quaternion( 0, 0, 0, 0 ) );
            }
            else if ( MathMod( zRotation, 360 ) < 280 && (MathMod( zRotation, 360 ) > 80) )
            {
                Vector3 respawnPosition = AICarTransform.position;
                AICarTransform.SetPositionAndRotation( new Vector3( respawnPosition.x, 0, respawnPosition.z ), new Quaternion( 0, 0, 0, 0 ) );
            }

            // If car falls beneath the map
            else if ( AICarTransform.position.y <= -15 )
            {
                Vector3 respawnPosition = AICarTransform.position;
                AICarTransform.SetPositionAndRotation( new Vector3( respawnPosition.x, 0, respawnPosition.z ), new Quaternion( 0, 0, 0, 0 ) );
            }
        }
    }
}
```

Figure 17: Reposition script

Since the angle rotation is periodic, I needed a way to get back to the base cases of clockwise and counterclockwise rotations.

To do that I quickly created **MathMod(int a, int b)**, which given two int (a and b), returns the rest of the **subtraction in $\mathbb{N}$** between a and b.

The rest are just checks, every 4 seconds, for the car to be in some wrong ranges of rotations and if yes, it just respawns the car in the same position but with the correct rotation around Z axis.