AI for Videogames Project Documentation

Valota Andrea - 982544

February 2022

1 Introduction

The goal of this project is to create an application that simulate cars moving in a track. The circuit is modeled based on the *National Motor Racetrack of Monza* and cars move using two variations of path following algorithms: the *Chase the Rabbit* approach and *Predictive Path Following* approach. The application is realized using *Unity version 2020.3.20f1* and the C# language.

2 Scenes

The application has two Scenes that can be found in the scene folder. In the first scene the path used to make the cars move is a more accurate representation of the track structure but the cars are moving on a plane without any 3D structure to constrain them. In the second scene I created an approximated 3D version of the track to test how the agents would perform in an environment where they could crash into walls. Both versions contains at least two cars. Each one uses the same parameters to manage speed, braking, drag and steering. Each agents use a different path following approach and each approach has different parameters, to obtain good results on the specific track.

2.1 Accurate track

This scene is composed mainly by:

- a plane
- the path
- *PPFCar*
- *CTRCar*

In this scene the path is exactly the same as the track, with more sharp turns that make more visible the cutting corner behaviour of the different approaches.

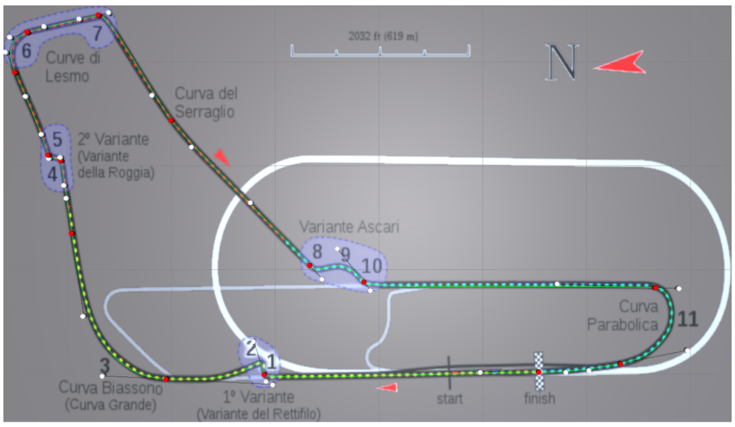

Figure 1: First scene

2.2 Approximated 3D track

This scene is composed mainly by:

- the 3D track
- the path
- *PPFCar*
- *CTRCar*

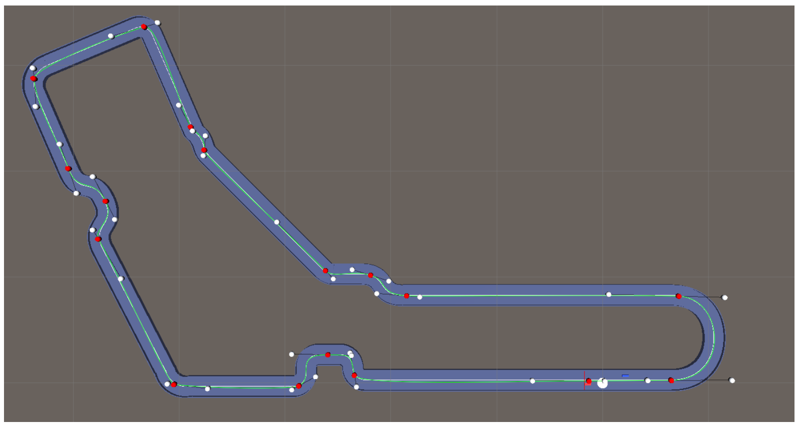

Figure 2: Second scene

In this scene I recreated the track using 3D models to check if the cars could complete the track even if the behaviours made them cut corners consistently. To create the 3D version I had to approximate the track, making some turn less sharp. Also having a basic 3D track makes the simulation more clear and understandable.

3 Path creation and scripts

All the scripts related to the creation of the path can be found in the PathScripts folder. The path is created by connecting various Bezier curves.

3.1 Path Creator

To create a new path in the scene we have to create an empty GameObject and add to it the *PathCreator* script. This script creates the new path and saves it. It also defines some parameters, like points color and diameter, useful to make the process of managing the path in the scene easier.

```
    public class PathCreator : MonoBehaviour {

    [HideInInspector]
    public Path path;

    public Color anchorCol = Color.red;
    public Color controlCol = Color.white;
    public Color segmentCol = Color.green;
    public Color selectedSegmentCol = Color.yellow;
    public float anchorDiameter = .1f;
    public float controlDiameter = .075f;
    public bool displayControlPoints = true;

    public void CreatePath()
    {
        path = new Path(transform.position);
    }

    void Reset()
    {
        CreatePath();
    }
}
```

3.2 Path

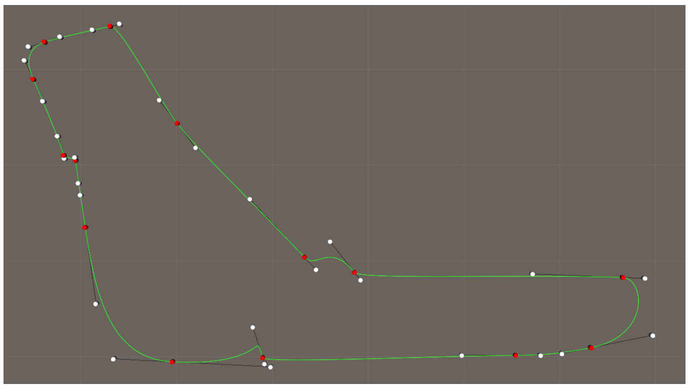

Figure 3: Path

The *Path* script contains all the attributes and methods needed to manage the path and saves all the points needed to create it. We consider two different types of points we can use: anchor points and control points. The anchor points are the initial and final point of each Bezier curve and the control points allow us to change the shape of the curve. The path also has a parameter isClosed to determine if the first and last point should be connected by a curve too, closing the path. The points are Vector3 points, so they exist in a 3D space but everything in the scene is placed on an XZ plane and the path needs to be placed on it too. To solve the issue we can use the SetToXZPlane method that place all the points on the XZ plane setting their Y to 1.

```
    //Set all the points on the XZ plane (set the Y value to 1)
    public void SetToXZPlane()
    {
        for (int i = 0; i < points.Count; i++)
        {
            points[i] = new Vector3(points[i].x, 1, points[i].z);
        }
    }
```

The script also contains a method that computes evenly spaced points on the path. This specific method is needed because the the various curve linked to create the path may not have the same length and that would cause major problems in the PathFollowing behaviours. To create the points an estimation of the length of a single curve is made and points are placed along the path. The spacing between the points is a parameter so it is easy to control from the GUI.

```
    //Calculate points on the path evenly spaced based on the
    spacing parameter
    public Vector3[] CalculateEvenlySpacedPoints(float spacing,
    float resolution = 1)
    {
        List<Vector3> evenlySpacedPoints = new List<Vector3>();
        evenlySpacedPoints.Add(points[0]);
        Vector3 previousPoint = points[0];
        float dstSinceLastEvenPoint = 0;

        for (int segmentIndex = 0; segmentIndex < NumSegments;
    segmentIndex++)
        {
            //Estimate curve lenght and computer the number of
            Vector3[] p = GetPointsInSegment(segmentIndex);
            float controlNetLength = Vector3.Distance(p[0], p[1]) +
     Vector3.Distance(p[1], p[2]) + Vector3.Distance(p[2], p[3]);
            float estimatedCurveLength = Vector3.Distance(p[0], p
    [3]) + controlNetLength / 2f;
            int divisions = Mathf.CeilToInt(estimatedCurveLength *
    resolution * 10);
            float t = 0;
            while (t <= 1)
            {
                t += 1f/divisions;
                Vector3 pointOnCurve = Bezier.EvaluateCubic(p[0], p
    [1], p[2], p[3], t);
                dstSinceLastEvenPoint += Vector3.Distance(
    previousPoint, pointOnCurve);

                while (dstSinceLastEvenPoint >= spacing)
                {
                    float overshootDst = dstSinceLastEvenPoint -
    spacing;
```

```
                    Vector3 newEvenlySpacedPoint = pointOnCurve + (
previousPoint - pointOnCurve).normalized * overshootDst;
                    evenlySpacedPoints.Add(newEvenlySpacedPoint);
                    dstSinceLastEvenPoint = overshootDst;
                    previousPoint = newEvenlySpacedPoint;
                }

                previousPoint = pointOnCurve;
            }
        }

        return evenlySpacedPoints.ToArray();
    }
```

3.3 Path Editor

Also, in the editor folder the *PathEditor* script can be found. it is used to make the path editable in a comfortable way directly form the scene. The main controls are:

- SHIFT + Left click to create a new anchor point
- Right click to delete the last created anchor point
- CTRL + Left click to call the SetToXZPlane method

All the control and anchor points can be moved just by using the Left click and dragging them around with the mouse.

4 Movement

The *DelegatedSteering* script is in charge to manage the movement of the cars. It computes the destination point based on the type of path following algorithm used and then obtain the forces to apply to the agent from all the movement behaviours associated with the current GameObject. At last it blends all the forces and makes the agent move.
The script is based on two abstract classes found in the *DelegatedUtilities* script: MovementBehaviour and PathFollowing. The *PathFollowing* class defines the abstract class that the different path following classes have to inherit from. These classes have to implement the GetDestination method that return the destination point the agent has to follow. To make sure that a single GameObject doesn't have multiple path follwing scripts I used the *DisallowMultipleComponent* option that Unity provides.

```
//To be extended by the classes that choose the destination point
[DisallowMultipleComponent]
public abstract class PathFollowing : MonoBehaviour
{
   public abstract Vector3 GetDestination(MovementStatus status);
}
```

The *MovementBehaviour* class defines the abstract class that the diffent movement behaviour classes have to inherit from. These classes has to implement the GetAcceleration method that returns the acceleration that a single component gives to the agent. All the acceleration will be blended by the DelegatedSteering script.

```
// To be extended by all movement behaviours
public abstract class MovementBehaviour : MonoBehaviour {
   public abstract Vector3 GetAcceleration (MovementStatus status, Vector3 destination);
}
```

4.1 Predictive Path Following

The first path following approach implemented is the Predictive Path Following. This approach predicts where the agent will be in a short time and then maps this to the nearest point on the path. If this candidate target is not placed farther along the path, then is changed so that it is. This approach can appear smooth for complex path with sudden changes of direction but has the downside of cutting corners when two path come close together.

```
public override Vector3 GetDestination(MovementStatus status)
{
    //Compute the future position of the agent
    Vector3 prediction = transform.position + (status.linearSpeed * status.movementDirection * predictionTime);

    //Find the point on the path closest to the prediction
    int min = 0;
    for (int i = 0; i < pathPoints.Length; i++)
    {
        if ((pathPoints[i] - prediction).magnitude < (pathPoints[min] - prediction).magnitude)
        {
            min = i;
        }

    }

    //If the prediction is off the path (+ pathRadius) the new destination is put back on the path plus an offset otherwise the current prediction is a valid destination
    if ((pathPoints[min] - prediction).magnitude < pathRadius && !Mathf.Approximately(status.linearSpeed,0f))
    {
        destination = prediction;
    }
    else
    {
        destination = pathPoints[(min + predictionOffset) % pathPoints.Length];
    }

    return destination;
}
```

In my implementation I compute the predicted position and map it to the closest point on the path. The mapping happens computing the distance between the prediction and points on the path. These points are the ones computed by the EvenlySpacedPoint method. Then, if the predicted position is inside the path radius, that point is set as the current destination, otherwise the destination becomes a point on the path farther than the mapped point computed.

4.2 Chase the rabbit

The second approach implemented is the Chase The Rabbit. This approach considers the current position of the agent and maps it on the closest point to the path. Then, the destination point is selected further along the path than the mapped point.

```
    public override Vector3 GetDestination(MovementStatus status)
    {
        //Find the point on the path closest to the current position
        int min = 0;
        for (int i = 0; i < pathPoints.Length; i++)
        {
            if ((pathPoints[i] - transform.position).magnitude < (pathPoints[min] - transform.position).magnitude)
            {
                min = i;
            }

        }

        //Set the destination further along the path
        destination = pathPoints[(min + predictionOffset) % pathPoints.Length];
        return destination;
    }
```

In my implementation I computed the mapping of the current position on the path and set the destination as a point further in the path.

4.3 Seek

The Seek behaviour is used to compute the cars acceleration given the destination to reach obtained by the scripts mentioned above. This steering behaviour tries to match the character position with the destination. Usually this behaviour tends to make the agent spiral around the destination point but in this specific case, since the destination is updated continuously and is always ahead with respect to the agent, this problem is avoided. In my implementation I also modified the seek behaviour in order to make the agents break when taking sharp turns to make the movement more suited for a car.

```
    public override Vector3 GetAcceleration (MovementStatus status, Vector3 destination) {

      if (destination != null) {
         //Vertical adjustment
         Vector3 verticalAdj = new Vector3 (destination.x, transform.position.y, destination.z);
         Vector3 toDestination = (verticalAdj - transform.position);

         if (toDestination.magnitude > stopAt)
         {
            Vector3 tangentComponent = Vector3.Project(toDestination.normalized, status.movementDirection);
            Vector3 normalComponent = (toDestination.normalized - tangentComponent);

            //If the destination is at BreakAngle degree from the movement direction it means we are doing a sharp turn and we need to break
            bool brakeCheck = (Vector3.Angle(status.movementDirection, toDestination) < brakeAtAngle);

            return (tangentComponent * (brakeCheck ? gas : -brake)) + (normalComponent * steer);
        }
        else {
            return Vector3.zero;
        }

    } else {
        return Vector3.zero;
    }
}
```

4.4 Drag

To make the agents more believable a Drag component was also added to them. This component simulate friction, a force that slows down the cars, and make them more physically accurate.

```
 public override Vector3 GetAcceleration (MovementStatus status, Vector3 destination) {
    return - (status.movementDirection.normalized * status.linearSpeed / linearDrag)
            - ((Quaternion.Euler (0f, 90f, 0f) * status.movementDirection.normalized) * status.angularSpeed / angularDrag);
}
```

4.5 Parameters

All the scripts have various parameters that influences how believable the cars are and how well they perform on the track. The main parameter of the Delegated Steering script are:

- Min linear speed
- Max linear speed
- Max angular speed

These parameters define the car minimum and maximum speed and his maximum angular speed. The parameters of the path following scripts are:

- Path name
- Spacing
- Prediction offset
- Prediction time

- Path radius

These parameter specify respectively: the name of the GameObject that has the path to follow, the spacing between the points computed on the path (the lower the number the more precise the path approximation), the number of points to skip when computing the destination point, the time at which the predicted position will be considered, the radius of the path (to avoid all the cars to align on a single line). The parameters of the Seek script are:

- Gas
- Steer
- Break
- Break at angle

These parameter specify: acceleration of the car, how fast the car can turn, how strong are the breaks, at which angle the turn is considered a sharp turn (used to make the car break if the turn is considered sharp). The parameters of the Drag script are:

- Linear drag
- Angular drag

These parameters define how strong the drag should be. The parameter indicates the number of seconds needed for the car to stop if no acceleration is applied to it.
All the parameters used are the ones that produced the best results in terms of completing the track regularly and smoothly.

4.6 Comparison between the two approaches

The two approaches used to compute the destination point create slightly different behaviours. The *Chase the Rabbit* approach follows the path more accurately and usually make sharper turns. The *Predictive Path Following* makes cars cut corners more but creates a more smooth and believable behaviour overall. For these reasons I would say that the *Predictive Path Following* can have better results but requires more tuning of parameters while the *Chase the Rabbit* approach is usually more robust in terms of following the path but can create some odd looking movements.

5 Conclusion

The goal of the project was to create a simulation of cars racing on a track using different path following algorithms. The application implemented the *National Motor Racetrack of Monza* as a path and two different version of path following: *Predictive Path Following* and *Chse the Rabbit*. The implemented cars AI work as intended and this simulation allows the user to analyze the two approaches, pointing out particular problems and advantages of both.