```
using System;
using System.Collections;
using UnityEngine;
using Random = UnityEngine.Random;

#pragma warning disable 649
namespace UnityStandardAssets.Vehicles.Car
{
    [RequireComponent(typeof (CarController))]
    public class CarAIControl : MonoBehaviour
    {
        public enum BrakeCondition
        {
            NeverBrake,                 // the car simply accelerates at full throttle all the time.
            TargetDirectionDifference,  // the car will brake according to the upcoming change in direction of the target. Useful for route-based AI, slowing for corners.
            TargetDistance,             // the car will brake as it approaches its target, regardless of the target's direction. Useful if you want the car to
                                        // head for a stationary target and come to rest when it arrives there.
        }

        // This script provides input to the car controller in the same way that the user control script does.
        // As such, it is really 'driving' the car, with no special physics or animation tricks to make the car behave properly.

        // "wandering" is used to give the cars a more human, less robotic feel. They can waver slightly
        // in speed and direction while driving towards their target.

        [SerializeField] [Range(0, 1)] private float m_CautiousSpeedFactor = 0.05f;               // percentage of max speed to use when being maximally cautious
        [SerializeField] [Range(0, 180)] private float m_CautiousMaxAngle = 50f;                  // angle of approaching corner to treat as warranting maximum caution
        [SerializeField] private float m_CautiousMaxDistance = 100f;                              // distance at which distance-based cautiousness begins
        [SerializeField] private float m_CautiousAngularVelocityFactor = 30f;                     // how cautious the AI
```

```
          should be when considering its own current angular velocity (i.e. easing off acceleration if spinning!)
        [SerializeField] private float m_SteerSensitivity = 0.05f;                                // how sensitively the AI uses steering input to turn to the desired direction
        [SerializeField] private float m_AccelSensitivity = 0.04f;                                // How sensitively the AI uses the accelerator to reach the current desired speed
        [SerializeField] private float m_BrakeSensitivity = 1f;                                   // How sensitively the AI uses the brake to reach the current desired speed
        [SerializeField] private float m_LateralWanderDistance = 3f;                              // how far the car will wander laterally towards its target
        [SerializeField] private float m_LateralWanderSpeed = 0.1f;                               // how fast the lateral wandering will fluctuate
        [SerializeField] [Range(0, 1)] private float m_AccelWanderAmount = 0.1f;                  // how much the cars acceleration will wander
        [SerializeField] private float m_AccelWanderSpeed = 0.1f;                                 // how fast the cars acceleration wandering will fluctuate
        [SerializeField] private BrakeCondition m_BrakeCondition = BrakeCondition.TargetDistance; // what should the AI consider when accelerating/braking?
        [SerializeField] private bool m_Driving;                                                  // whether the AI is currently actively driving or stopped.
        [SerializeField] private Transform m_Target;                                              // 'target' the target object to aim for.
        [SerializeField] private bool m_StopWhenTargetReached;                                    // should we stop driving when we reach the target?
        [SerializeField] private float m_ReachTargetThreshold = 2;                                // proximity to target to consider we 'reached' it, and stop driving.

        private float m_RandomPerlin;             // A random value for the car to base its wander on (so that AI cars don't all wander in the same pattern)
        private CarController m_CarController;    // Reference to actual car controller we are controlling
        private float m_AvoidOtherCarTime;        // time until which to avoid the car we recently collided with
        private float m_AvoidOtherCarSlowdown;    // how much to slow down due to colliding with another car, whilst avoiding
        private float m_AvoidPathOffset;          // direction (-1 or 1) in which to offset path to avoid other car, whilst avoiding
        private Rigidbody m_Rigidbody;

        private bool BarrierStop = false;

        private bool RaceCompleted = false;

        public bool AI1;
        public bool AI2;
        public bool AI3;
        public bool AI4;
        public bool AI5;
        public bool AI6;
        public bool AI7;

        private void Awake()
        {
            // get the car controller reference
            m_CarController = GetComponent<CarController>();

            // give the random perlin a random value
            m_RandomPerlin = Random.value*100;

            m_Rigidbody = GetComponent<Rigidbody>();
        }


        private void FixedUpdate()
        {
            if (m_Target == null || !m_Driving)
            {
                // Car should not be moving,
                // use handbrake to stop
                m_CarController.Move(0, 0, -1f, 1f);
            }
            else
            {
                if (SaveScript.RaceStart == true && RaceCompleted == false)
                {
                    Vector3 fwd = transform.forward;
                   if (m_Rigidbody.velocity.magnitude > m_CarController.MaxSpeed * 0.1f)
                    {
                        fwd = m_Rigidbody.velocity;
                    }

                    float desiredSpeed = m_CarController.MaxSpeed;

                    // now it's time to decide if we should be slowing down...
                    switch (m_BrakeCondition)
                    {
                        case BrakeCondition.TargetDirectionDifference:
                            {
                                // the car will brake according to the upcoming change in direction of the target. Useful for route-based AI, slowing for corners.

                                // check out the angle of our target compared to the current direction of the car
                                float approachingCornerAngle = Vector3.Angle(m_Target.forward, fwd);

                                // also consider the current amount we're turning, multiplied up and then compared in the same way as an upcoming corner angle
                                float spinningAngle = m_Rigidbody.angularVelocity.magnitude * m_CautiousAngularVelocityFactor;

                                // if it's different to our current angle, we need to be cautious (i.e. slow down) a certain amount
                                float cautiousnessRequired = Mathf.InverseLerp(0, m_CautiousMaxAngle,
                                                                               Mathf.Max(spinningAngle,
                                                                                         approachingCornerAngle));
                                desiredSpeed = Mathf.Lerp(m_CarController.MaxSpeed, m_CarController.MaxSpeed * m_CautiousSpeedFactor,
                                                          cautiousnessRequired);
                                break;
                            }

                        case BrakeCondition.TargetDistance:
                            {
                                // the car will brake as it approaches its target, regardless of the target's direction. Useful if you want the car to
                                // head for a stationary target and come to rest when it arrives there.

                                // check out the distance to target
                                Vector3 delta = m_Target.position - transform.position;
                                float distanceCautiousFactor = Mathf.InverseLerp(m_CautiousMaxDistance, 0, delta.magnitude);

                                // also consider the current amount we're turning, multiplied up and then compared in the same way as an upcoming corner angle
                                float spinningAngle = m_Rigidbody.angularVelocity.magnitude * m_CautiousAngularVelocityFactor;

                                // if it's different to our current angle, we need to be cautious (i.e. slow down) a certain amount
                                float cautiousnessRequired = Mathf.Max(
                                    Mathf.InverseLerp(0, m_CautiousMaxAngle, spinningAngle), distanceCautiousFactor);
                                desiredSpeed = Mathf.Lerp(m_CarController.MaxSpeed, m_CarController.MaxSpeed * m_CautiousSpeedFactor,
                                                          cautiousnessRequired);
                                break;
                            }

                        case BrakeCondition.NeverBrake:
                            break;

                    }

                    // Evasive action due to collision with other cars:

                    // our target position starts off as the 'real' target position
                    Vector3 offsetTargetPos = m_Target.position;

                    // if are we currently taking evasive action to prevent being stuck against another car:
                    if (Time.time < m_AvoidOtherCarTime)
                    {
                        // slow down if necessary (if we were behind the other car when collision occured)
                        desiredSpeed *= m_AvoidOtherCarSlowdown;

                        // and veer towards the side of our path-to-target that is away from the other car
                        offsetTargetPos += m_Target.right * m_AvoidPathOffset;
                    }
                    else
                    {
                        // no need for evasive action, we can just wander across the path-to-target in a random way,
                        // which can help prevent AI from seeming too uniform and robotic in their driving
                        offsetTargetPos += m_Target.right *
                                           (Mathf.PerlinNoise(Time.time * m_LateralWanderSpeed, m_RandomPerlin) * 2 - 1) *
                                           m_LateralWanderDistance;
                    }

                    // use different sensitivity depending on whether accelerating or braking:
                    float accelBrakeSensitivity = (desiredSpeed < m_CarController.CurrentSpeed)
                                                      ? m_BrakeSensitivity
                                                      : m_AccelSensitivity;

                    // decide the actual amount of accel/brake input to achieve desired speed.
                    float accel = Mathf.Clamp((desiredSpeed - m_CarController.CurrentSpeed) * accelBrakeSensitivity, -1, 1);

                    // add acceleration 'wander', which also prevents AI from seeming too uniform and robotic in their driving
                    // i.e. increasing the accel wander amount can introduce jostling and bumps between AI cars in a race
                    accel *= (1 - m_AccelWanderAmount) +
                             (Mathf.PerlinNoise(Time.time * m_AccelWanderSpeed, m_RandomPerlin) * m_AccelWanderAmount);

                    // calculate the local-relative position of the target, to steer towards
                    Vector3 localTarget = transform.InverseTransformPoint(offsetTargetPos);
```

```

                    // work out the local angle towards the target
                    float targetAngle = Mathf.Atan2(localTarget.x, localTarget.z) * Mathf.Rad2Deg;

                    // get the amount of steering needed to aim the car towards the target
                    float steer = Mathf.Clamp(targetAngle * m_SteerSensitivity, -1, 1) * Mathf.Sign(m_CarController.CurrentSpeed);

                    // feed input to the car controller.
                    if (BarrierStop == false)
                    {
                        m_CarController.Move(steer, accel, accel, 0f);
                    }
                    if (BarrierStop == true)
                    {
                        m_CarController.Move(steer, -accel, -accel, 0f);
                    }

                    // if appropriate, stop driving when we're close enough to the target.
                    if (m_StopWhenTargetReached && localTarget.magnitude < m_ReachTargetThreshold)
                    {
                        m_Driving = false;
                    }
                }
            }
        }

        private void OnTriggerEnter(Collider other)
        {
            if(other.gameObject.CompareTag("Barrier"))
            {
                BarrierStop = true;
            }
            if (AI1 == true)
            {
                if (other.gameObject.CompareTag("AI1"))
                {
                    StartCoroutine(StopDriving());
                }
            }
            if (AI2 == true)
            {
                if (other.gameObject.CompareTag("AI2"))
                {
                    StartCoroutine(StopDriving());
                }
            }
            if (AI3 == true)
            {
                if (other.gameObject.CompareTag("AI3"))
                {
                    StartCoroutine(StopDriving());
                }
            }
            if (AI4 == true)
            {
                if (other.gameObject.CompareTag("AI4"))
                {
                    StartCoroutine(StopDriving());
                }
            }
            if (AI5 == true)
            {
                if (other.gameObject.CompareTag("AI5"))
                {
                    StartCoroutine(StopDriving());
                }
            }
            if (AI6 == true)
            {
                if (other.gameObject.CompareTag("AI6"))
                {
                    StartCoroutine(StopDriving());
                }
            }
            if (AI7 == true)
            {
                if (other.gameObject.CompareTag("AI7"))
                {
                    StartCoroutine(StopDriving());
                }
            }
        }

        private void OnTriggerExit(Collider other)
        {
            if (other.gameObject.CompareTag("Barrier"))
            {
                StartCoroutine(BarrierReset());
            }
        }


        private void OnCollisionStay(Collision col)
        {
            // detect collision against other cars, so that we can take evasive action
            if (col.rigidbody != null)
            {
                var otherAI = col.rigidbody.GetComponent<CarAIControl>();
                var PlayerCar = col.rigidbody.GetComponent<CarController>();
                if (otherAI != null || PlayerCar != null)
                {
                    // we'll take evasive action for 1 second
                    m_AvoidOtherCarTime = Time.time + 1;

                    // but who's in front?...
                    if (Vector3.Angle(transform.forward, otherAI.transform.position - transform.position) < 90)
                    {
                        // the other ai is in front, so it is only good manners that we ought to brake...
                        m_AvoidOtherCarSlowdown = 0.5f;
                    }
                    if (Vector3.Angle(transform.forward, PlayerCar.transform.position - transform.position) < 90)
                    {
                        // the other ai is in front, so it is only good manners that we ought to brake...
                        m_AvoidOtherCarSlowdown = 0.5f;
                    }
                    else
                    {
                        // we're in front! ain't slowing down for anybody...
                        m_AvoidOtherCarSlowdown = 1;
                    }

                    // both cars should take evasive action by driving along an offset from the path centre,
                    // away from the other car
                    var otherCarLocalDelta = transform.InverseTransformPoint(otherAI.transform.position);
                    float otherCarAngle = Mathf.Atan2(otherCarLocalDelta.x, otherCarLocalDelta.z);
                    m_AvoidPathOffset = m_LateralWanderDistance*-Mathf.Sign(otherCarAngle);
                }
            }
        }


        public void SetTarget(Transform target)
        {
            m_Target = target;
            m_Driving = true;
        }

        IEnumerator BarrierReset()
        {
            yield return new WaitForSeconds(Random.Range(1f, 5f));
            BarrierStop = false;
        }

        IEnumerator StopDriving()
        {
            yield return new WaitForSeconds(Random.Range(1f, 9f));
            RaceCompleted = true;
            m_CarController.Move(0, 0, -1f, 1f);
        }
    }
}
```