using System;

using UnityEngine;

using Random = UnityEngine.Random;

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 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

{

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.

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 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>();

if (otherAI != 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;

}

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;

}

}

}