using System;

using UnityEngine;

namespace UnityStandardAssets.Vehicles.Car

{

internal enum CarDriveType

{

FrontWheelDrive,

RearWheelDrive,

FourWheelDrive

}

internal enum SpeedType

{

MPH,

KPH

}

public class CarController : MonoBehaviour

{

[SerializeField] private CarDriveType m\_CarDriveType = CarDriveType.FourWheelDrive;

[SerializeField] private WheelCollider[] m\_WheelColliders = new WheelCollider[4];

[SerializeField] private GameObject[] m\_WheelMeshes = new GameObject[4];

[SerializeField] private WheelEffects[] m\_WheelEffects = new WheelEffects[4];

[SerializeField] private Vector3 m\_CentreOfMassOffset;

[SerializeField] private float m\_MaximumSteerAngle;

[Range(0, 1)] [SerializeField] private float m\_SteerHelper; // 0 is raw physics , 1 the car will grip in the direction it is facing

[Range(0, 1)] [SerializeField] private float m\_TractionControl; // 0 is no traction control, 1 is full interference

[SerializeField] private float m\_FullTorqueOverAllWheels;

[SerializeField] private float m\_ReverseTorque;

[SerializeField] private float m\_MaxHandbrakeTorque;

[SerializeField] private float m\_Downforce = 100f;

[SerializeField] private SpeedType m\_SpeedType;

[SerializeField] private float m\_Topspeed = 200;

[SerializeField] private static int NoOfGears = 5;

[SerializeField] private float m\_RevRangeBoundary = 1f;

[SerializeField] private float m\_SlipLimit;

[SerializeField] private float m\_BrakeTorque;

private Quaternion[] m\_WheelMeshLocalRotations;

private Vector3 m\_Prevpos, m\_Pos;

private float m\_SteerAngle;

private int m\_GearNum;

private float m\_GearFactor;

private float m\_OldRotation;

private float m\_CurrentTorque;

private Rigidbody m\_Rigidbody;

private const float k\_ReversingThreshold = 0.01f;

public bool Skidding { get; private set; }

public float BrakeInput { get; private set; }

public float CurrentSteerAngle{ get { return m\_SteerAngle; }}

public float CurrentSpeed{ get { return m\_Rigidbody.velocity.magnitude\*2.23693629f; }}

public float MaxSpeed{get { return m\_Topspeed; }}

public float Revs { get; private set; }

public float AccelInput { get; private set; }

// Use this for initialization

private void Start()

{

m\_WheelMeshLocalRotations = new Quaternion[4];

for (int i = 0; i < 4; i++)

{

m\_WheelMeshLocalRotations[i] = m\_WheelMeshes[i].transform.localRotation;

}

m\_WheelColliders[0].attachedRigidbody.centerOfMass = m\_CentreOfMassOffset;

m\_MaxHandbrakeTorque = float.MaxValue;

m\_Rigidbody = GetComponent<Rigidbody>();

m\_CurrentTorque = m\_FullTorqueOverAllWheels - (m\_TractionControl\*m\_FullTorqueOverAllWheels);

}

private void GearChanging()

{

float f = Mathf.Abs(CurrentSpeed/MaxSpeed);

float upgearlimit = (1/(float) NoOfGears)\*(m\_GearNum + 1);

float downgearlimit = (1/(float) NoOfGears)\*m\_GearNum;

if (m\_GearNum > 0 && f < downgearlimit)

{

m\_GearNum--;

}

if (f > upgearlimit && (m\_GearNum < (NoOfGears - 1)))

{

m\_GearNum++;

}

}

// simple function to add a curved bias towards 1 for a value in the 0-1 range

private static float CurveFactor(float factor)

{

return 1 - (1 - factor)\*(1 - factor);

}

// unclamped version of Lerp, to allow value to exceed the from-to range

private static float ULerp(float from, float to, float value)

{

return (1.0f - value)\*from + value\*to;

}

private void CalculateGearFactor()

{

float f = (1/(float) NoOfGears);

// gear factor is a normalised representation of the current speed within the current gear's range of speeds.

// We smooth towards the 'target' gear factor, so that revs don't instantly snap up or down when changing gear.

var targetGearFactor = Mathf.InverseLerp(f\*m\_GearNum, f\*(m\_GearNum + 1), Mathf.Abs(CurrentSpeed/MaxSpeed));

m\_GearFactor = Mathf.Lerp(m\_GearFactor, targetGearFactor, Time.deltaTime\*5f);

}

private void CalculateRevs()

{

// calculate engine revs (for display / sound)

// (this is done in retrospect - revs are not used in force/power calculations)

CalculateGearFactor();

var gearNumFactor = m\_GearNum/(float) NoOfGears;

var revsRangeMin = ULerp(0f, m\_RevRangeBoundary, CurveFactor(gearNumFactor));

var revsRangeMax = ULerp(m\_RevRangeBoundary, 1f, gearNumFactor);

Revs = ULerp(revsRangeMin, revsRangeMax, m\_GearFactor);

}

public void Move(float steering, float accel, float footbrake, float handbrake)

{

for (int i = 0; i < 4; i++)

{

Quaternion quat;

Vector3 position;

m\_WheelColliders[i].GetWorldPose(out position, out quat);

m\_WheelMeshes[i].transform.position = position;

m\_WheelMeshes[i].transform.rotation = quat;

}

//clamp input values

steering = Mathf.Clamp(steering, -1, 1);

AccelInput = accel = Mathf.Clamp(accel, 0, 1);

BrakeInput = footbrake = -1\*Mathf.Clamp(footbrake, -1, 0);

handbrake = Mathf.Clamp(handbrake, 0, 1);

//Set the steer on the front wheels.

//Assuming that wheels 0 and 1 are the front wheels.

m\_SteerAngle = steering\*m\_MaximumSteerAngle;

m\_WheelColliders[0].steerAngle = m\_SteerAngle;

m\_WheelColliders[1].steerAngle = m\_SteerAngle;

SteerHelper();

ApplyDrive(accel, footbrake);

CapSpeed();

//Set the handbrake.

//Assuming that wheels 2 and 3 are the rear wheels.

if (handbrake > 0f)

{

var hbTorque = handbrake\*m\_MaxHandbrakeTorque;

m\_WheelColliders[2].brakeTorque = hbTorque;

m\_WheelColliders[3].brakeTorque = hbTorque;

}

CalculateRevs();

GearChanging();

AddDownForce();

CheckForWheelSpin();

TractionControl();

}

private void CapSpeed()

{

float speed = m\_Rigidbody.velocity.magnitude;

switch (m\_SpeedType)

{

case SpeedType.MPH:

speed \*= 2.23693629f;

if (speed > m\_Topspeed)

m\_Rigidbody.velocity = (m\_Topspeed/2.23693629f) \* m\_Rigidbody.velocity.normalized;

break;

case SpeedType.KPH:

speed \*= 3.6f;

if (speed > m\_Topspeed)

m\_Rigidbody.velocity = (m\_Topspeed/3.6f) \* m\_Rigidbody.velocity.normalized;

break;

}

}

private void ApplyDrive(float accel, float footbrake)

{

float thrustTorque;

switch (m\_CarDriveType)

{

case CarDriveType.FourWheelDrive:

thrustTorque = accel \* (m\_CurrentTorque / 4f);

for (int i = 0; i < 4; i++)

{

m\_WheelColliders[i].motorTorque = thrustTorque;

}

break;

case CarDriveType.FrontWheelDrive:

thrustTorque = accel \* (m\_CurrentTorque / 2f);

m\_WheelColliders[0].motorTorque = m\_WheelColliders[1].motorTorque = thrustTorque;

break;

case CarDriveType.RearWheelDrive:

thrustTorque = accel \* (m\_CurrentTorque / 2f);

m\_WheelColliders[2].motorTorque = m\_WheelColliders[3].motorTorque = thrustTorque;

break;

}

for (int i = 0; i < 4; i++)

{

if (CurrentSpeed > 5 && Vector3.Angle(transform.forward, m\_Rigidbody.velocity) < 50f)

{

m\_WheelColliders[i].brakeTorque = m\_BrakeTorque\*footbrake;

}

else if (footbrake > 0)

{

m\_WheelColliders[i].brakeTorque = 0f;

m\_WheelColliders[i].motorTorque = -m\_ReverseTorque\*footbrake;

}

}

}

private void SteerHelper()

{

for (int i = 0; i < 4; i++)

{

WheelHit wheelhit;

m\_WheelColliders[i].GetGroundHit(out wheelhit);

if (wheelhit.normal == Vector3.zero)

return; // wheels arent on the ground so dont realign the rigidbody velocity

}

// this if is needed to avoid gimbal lock problems that will make the car suddenly shift direction

if (Mathf.Abs(m\_OldRotation - transform.eulerAngles.y) < 10f)

{

var turnadjust = (transform.eulerAngles.y - m\_OldRotation) \* m\_SteerHelper;

Quaternion velRotation = Quaternion.AngleAxis(turnadjust, Vector3.up);

m\_Rigidbody.velocity = velRotation \* m\_Rigidbody.velocity;

}

m\_OldRotation = transform.eulerAngles.y;

}

// this is used to add more grip in relation to speed

private void AddDownForce()

{

m\_WheelColliders[0].attachedRigidbody.AddForce(-transform.up\*m\_Downforce\*

m\_WheelColliders[0].attachedRigidbody.velocity.magnitude);

}

// checks if the wheels are spinning and is so does three things

// 1) emits particles

// 2) plays tiure skidding sounds

// 3) leaves skidmarks on the ground

// these effects are controlled through the WheelEffects class

private void CheckForWheelSpin()

{

// loop through all wheels

for (int i = 0; i < 4; i++)

{

WheelHit wheelHit;

m\_WheelColliders[i].GetGroundHit(out wheelHit);

// is the tire slipping above the given threshhold

if (Mathf.Abs(wheelHit.forwardSlip) >= m\_SlipLimit || Mathf.Abs(wheelHit.sidewaysSlip) >= m\_SlipLimit)

{

m\_WheelEffects[i].EmitTyreSmoke();

// avoiding all four tires screeching at the same time

// if they do it can lead to some strange audio artefacts

if (!AnySkidSoundPlaying())

{

m\_WheelEffects[i].PlayAudio();

}

continue;

}

// if it wasnt slipping stop all the audio

if (m\_WheelEffects[i].PlayingAudio)

{

m\_WheelEffects[i].StopAudio();

}

// end the trail generation

m\_WheelEffects[i].EndSkidTrail();

}

}

// crude traction control that reduces the power to wheel if the car is wheel spinning too much

private void TractionControl()

{

WheelHit wheelHit;

switch (m\_CarDriveType)

{

case CarDriveType.FourWheelDrive:

// loop through all wheels

for (int i = 0; i < 4; i++)

{

m\_WheelColliders[i].GetGroundHit(out wheelHit);

AdjustTorque(wheelHit.forwardSlip);

}

break;

case CarDriveType.RearWheelDrive:

m\_WheelColliders[2].GetGroundHit(out wheelHit);

AdjustTorque(wheelHit.forwardSlip);

m\_WheelColliders[3].GetGroundHit(out wheelHit);

AdjustTorque(wheelHit.forwardSlip);

break;

case CarDriveType.FrontWheelDrive:

m\_WheelColliders[0].GetGroundHit(out wheelHit);

AdjustTorque(wheelHit.forwardSlip);

m\_WheelColliders[1].GetGroundHit(out wheelHit);

AdjustTorque(wheelHit.forwardSlip);

break;

}

}

private void AdjustTorque(float forwardSlip)

{

if (forwardSlip >= m\_SlipLimit && m\_CurrentTorque >= 0)

{

m\_CurrentTorque -= 10 \* m\_TractionControl;

}

else

{

m\_CurrentTorque += 10 \* m\_TractionControl;

if (m\_CurrentTorque > m\_FullTorqueOverAllWheels)

{

m\_CurrentTorque = m\_FullTorqueOverAllWheels;

}

}

}

private bool AnySkidSoundPlaying()

{

for (int i = 0; i < 4; i++)

{

if (m\_WheelEffects[i].PlayingAudio)

{

return true;

}

}

return false;

}

}

}