using System;

using UnityEngine;

namespace UnityStandardAssets.Vehicles.Aeroplane

{

[RequireComponent(typeof (Rigidbody))]

public class AeroplaneController : MonoBehaviour

{

[SerializeField] private float m\_MaxEnginePower = 40f; // The maximum output of the engine.

[SerializeField] private float m\_Lift = 0.002f; // The amount of lift generated by the aeroplane moving forwards.

[SerializeField] private float m\_ZeroLiftSpeed = 300; // The speed at which lift is no longer applied.

[SerializeField] private float m\_RollEffect = 1f; // The strength of effect for roll input.

[SerializeField] private float m\_PitchEffect = 1f; // The strength of effect for pitch input.

[SerializeField] private float m\_YawEffect = 0.2f; // The strength of effect for yaw input.

[SerializeField] private float m\_BankedTurnEffect = 0.5f; // The amount of turn from doing a banked turn.

[SerializeField] private float m\_AerodynamicEffect = 0.02f; // How much aerodynamics affect the speed of the aeroplane.

[SerializeField] private float m\_AutoTurnPitch = 0.5f; // How much the aeroplane automatically pitches when in a banked turn.

[SerializeField] private float m\_AutoRollLevel = 0.2f; // How much the aeroplane tries to level when not rolling.

[SerializeField] private float m\_AutoPitchLevel = 0.2f; // How much the aeroplane tries to level when not pitching.

[SerializeField] private float m\_AirBrakesEffect = 3f; // How much the air brakes effect the drag.

[SerializeField] private float m\_ThrottleChangeSpeed = 0.3f; // The speed with which the throttle changes.

[SerializeField] private float m\_DragIncreaseFactor = 0.001f; // how much drag should increase with speed.

public float Altitude { get; private set; } // The aeroplane's height above the ground.

public float Throttle { get; private set; } // The amount of throttle being used.

public bool AirBrakes { get; private set; } // Whether or not the air brakes are being applied.

public float ForwardSpeed { get; private set; } // How fast the aeroplane is traveling in it's forward direction.

public float EnginePower { get; private set; } // How much power the engine is being given.

public float MaxEnginePower{ get { return m\_MaxEnginePower; }} // The maximum output of the engine.

public float RollAngle { get; private set; }

public float PitchAngle { get; private set; }

public float RollInput { get; private set; }

public float PitchInput { get; private set; }

public float YawInput { get; private set; }

public float ThrottleInput { get; private set; }

private float m\_OriginalDrag; // The drag when the scene starts.

private float m\_OriginalAngularDrag; // The angular drag when the scene starts.

private float m\_AeroFactor;

private bool m\_Immobilized = false; // used for making the plane uncontrollable, i.e. if it has been hit or crashed.

private float m\_BankedTurnAmount;

private Rigidbody m\_Rigidbody;

WheelCollider[] m\_WheelColliders;

private void Start()

{

m\_Rigidbody = GetComponent<Rigidbody>();

// Store original drag settings, these are modified during flight.

m\_OriginalDrag = m\_Rigidbody.drag;

m\_OriginalAngularDrag = m\_Rigidbody.angularDrag;

for (int i = 0; i < transform.childCount; i++ )

{

foreach (var componentsInChild in transform.GetChild(i).GetComponentsInChildren<WheelCollider>())

{

componentsInChild.motorTorque = 0.18f;

}

}

}

public void Move(float rollInput, float pitchInput, float yawInput, float throttleInput, bool airBrakes)

{

// transfer input parameters into properties.s

RollInput = rollInput;

PitchInput = pitchInput;

YawInput = yawInput;

ThrottleInput = throttleInput;

AirBrakes = airBrakes;

ClampInputs();

CalculateRollAndPitchAngles();

AutoLevel();

CalculateForwardSpeed();

ControlThrottle();

CalculateDrag();

CaluclateAerodynamicEffect();

CalculateLinearForces();

CalculateTorque();

CalculateAltitude();

}

private void ClampInputs()

{

// clamp the inputs to -1 to 1 range

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

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

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

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

}

private void CalculateRollAndPitchAngles()

{

// Calculate roll & pitch angles

// Calculate the flat forward direction (with no y component).

var flatForward = transform.forward;

flatForward.y = 0;

// If the flat forward vector is non-zero (which would only happen if the plane was pointing exactly straight upwards)

if (flatForward.sqrMagnitude > 0)

{

flatForward.Normalize();

// calculate current pitch angle

var localFlatForward = transform.InverseTransformDirection(flatForward);

PitchAngle = Mathf.Atan2(localFlatForward.y, localFlatForward.z);

// calculate current roll angle

var flatRight = Vector3.Cross(Vector3.up, flatForward);

var localFlatRight = transform.InverseTransformDirection(flatRight);

RollAngle = Mathf.Atan2(localFlatRight.y, localFlatRight.x);

}

}

private void AutoLevel()

{

// The banked turn amount (between -1 and 1) is the sine of the roll angle.

// this is an amount applied to elevator input if the user is only using the banking controls,

// because that's what people expect to happen in games!

m\_BankedTurnAmount = Mathf.Sin(RollAngle);

// auto level roll, if there's no roll input:

if (RollInput == 0f)

{

RollInput = -RollAngle\*m\_AutoRollLevel;

}

// auto correct pitch, if no pitch input (but also apply the banked turn amount)

if (PitchInput == 0f)

{

PitchInput = -PitchAngle\*m\_AutoPitchLevel;

PitchInput -= Mathf.Abs(m\_BankedTurnAmount\*m\_BankedTurnAmount\*m\_AutoTurnPitch);

}

}

private void CalculateForwardSpeed()

{

// Forward speed is the speed in the planes's forward direction (not the same as its velocity, eg if falling in a stall)

var localVelocity = transform.InverseTransformDirection(m\_Rigidbody.velocity);

ForwardSpeed = Mathf.Max(0, localVelocity.z);

}

private void ControlThrottle()

{

// override throttle if immobilized

if (m\_Immobilized)

{

ThrottleInput = -0.5f;

}

// Adjust throttle based on throttle input (or immobilized state)

Throttle = Mathf.Clamp01(Throttle + ThrottleInput\*Time.deltaTime\*m\_ThrottleChangeSpeed);

// current engine power is just:

EnginePower = Throttle\*m\_MaxEnginePower;

}

private void CalculateDrag()

{

// increase the drag based on speed, since a constant drag doesn't seem "Real" (tm) enough

float extraDrag = m\_Rigidbody.velocity.magnitude\*m\_DragIncreaseFactor;

// Air brakes work by directly modifying drag. This part is actually pretty realistic!

m\_Rigidbody.drag = (AirBrakes ? (m\_OriginalDrag + extraDrag)\*m\_AirBrakesEffect : m\_OriginalDrag + extraDrag);

// Forward speed affects angular drag - at high forward speed, it's much harder for the plane to spin

m\_Rigidbody.angularDrag = m\_OriginalAngularDrag\*ForwardSpeed;

}

private void CaluclateAerodynamicEffect()

{

// "Aerodynamic" calculations. This is a very simple approximation of the effect that a plane

// will naturally try to align itself in the direction that it's facing when moving at speed.

// Without this, the plane would behave a bit like the asteroids spaceship!

if (m\_Rigidbody.velocity.magnitude > 0)

{

// compare the direction we're pointing with the direction we're moving:

m\_AeroFactor = Vector3.Dot(transform.forward, m\_Rigidbody.velocity.normalized);

// multipled by itself results in a desirable rolloff curve of the effect

m\_AeroFactor \*= m\_AeroFactor;

// Finally we calculate a new velocity by bending the current velocity direction towards

// the the direction the plane is facing, by an amount based on this aeroFactor

var newVelocity = Vector3.Lerp(m\_Rigidbody.velocity, transform.forward\*ForwardSpeed,

m\_AeroFactor\*ForwardSpeed\*m\_AerodynamicEffect\*Time.deltaTime);

m\_Rigidbody.velocity = newVelocity;

// also rotate the plane towards the direction of movement - this should be a very small effect, but means the plane ends up

// pointing downwards in a stall

m\_Rigidbody.rotation = Quaternion.Slerp(m\_Rigidbody.rotation,

Quaternion.LookRotation(m\_Rigidbody.velocity, transform.up),

m\_AerodynamicEffect\*Time.deltaTime);

}

}

private void CalculateLinearForces()

{

// Now calculate forces acting on the aeroplane:

// we accumulate forces into this variable:

var forces = Vector3.zero;

// Add the engine power in the forward direction

forces += EnginePower\*transform.forward;

// The direction that the lift force is applied is at right angles to the plane's velocity (usually, this is 'up'!)

var liftDirection = Vector3.Cross(m\_Rigidbody.velocity, transform.right).normalized;

// The amount of lift drops off as the plane increases speed - in reality this occurs as the pilot retracts the flaps

// shortly after takeoff, giving the plane less drag, but less lift. Because we don't simulate flaps, this is

// a simple way of doing it automatically:

var zeroLiftFactor = Mathf.InverseLerp(m\_ZeroLiftSpeed, 0, ForwardSpeed);

// Calculate and add the lift power

var liftPower = ForwardSpeed\*ForwardSpeed\*m\_Lift\*zeroLiftFactor\*m\_AeroFactor;

forces += liftPower\*liftDirection;

// Apply the calculated forces to the the Rigidbody

m\_Rigidbody.AddForce(forces);

}

private void CalculateTorque()

{

// We accumulate torque forces into this variable:

var torque = Vector3.zero;

// Add torque for the pitch based on the pitch input.

torque += PitchInput\*m\_PitchEffect\*transform.right;

// Add torque for the yaw based on the yaw input.

torque += YawInput\*m\_YawEffect\*transform.up;

// Add torque for the roll based on the roll input.

torque += -RollInput\*m\_RollEffect\*transform.forward;

// Add torque for banked turning.

torque += m\_BankedTurnAmount\*m\_BankedTurnEffect\*transform.up;

// The total torque is multiplied by the forward speed, so the controls have more effect at high speed,

// and little effect at low speed, or when not moving in the direction of the nose of the plane

// (i.e. falling while stalled)

m\_Rigidbody.AddTorque(torque\*ForwardSpeed\*m\_AeroFactor);

}

private void CalculateAltitude()

{

// Altitude calculations - we raycast downwards from the aeroplane

// starting a safe distance below the plane to avoid colliding with any of the plane's own colliders

var ray = new Ray(transform.position - Vector3.up\*10, -Vector3.up);

RaycastHit hit;

Altitude = Physics.Raycast(ray, out hit) ? hit.distance + 10 : transform.position.y;

}

// Immobilize can be called from other objects, for example if this plane is hit by a weapon and should become uncontrollable

public void Immobilize()

{

m\_Immobilized = true;

}

// Reset is called via the ObjectResetter script, if present.

public void Reset()

{

m\_Immobilized = false;

}

}

}