using UnityEngine;

using UnityEngine.Rendering;

namespace UnityStandardAssets.CinematicEffects

{

[ExecuteInEditMode]

[RequireComponent(typeof(Camera))]

[AddComponentMenu("Image Effects/Cinematic/Ambient Occlusion")]

#if UNITY\_5\_4\_OR\_NEWER

[ImageEffectAllowedInSceneView]

#endif

public partial class AmbientOcclusion : MonoBehaviour

{

#region Public Properties

/// Effect settings.

[SerializeField]

public Settings settings = Settings.defaultSettings;

/// Checks if the ambient-only mode is supported under the current settings.

public bool isAmbientOnlySupported

{

get { return targetCamera.hdr && occlusionSource == OcclusionSource.GBuffer; }

}

/// Checks if the G-buffer is available

public bool isGBufferAvailable

{

get { return targetCamera.actualRenderingPath == RenderingPath.DeferredShading; }

}

#endregion

#region Private Properties

// Properties referring to the current settings

float intensity

{

get { return settings.intensity; }

}

float radius

{

get { return Mathf.Max(settings.radius, 1e-4f); }

}

SampleCount sampleCount

{

get { return settings.sampleCount; }

}

int sampleCountValue

{

get

{

switch (settings.sampleCount)

{

case SampleCount.Lowest: return 3;

case SampleCount.Low: return 6;

case SampleCount.Medium: return 12;

case SampleCount.High: return 20;

}

return Mathf.Clamp(settings.sampleCountValue, 1, 256);

}

}

OcclusionSource occlusionSource

{

get

{

if (settings.occlusionSource == OcclusionSource.GBuffer && !isGBufferAvailable)

// An unavailable source was chosen: fallback to DepthNormalsTexture.

return OcclusionSource.DepthNormalsTexture;

else

return settings.occlusionSource;

}

}

bool downsampling

{

get { return settings.downsampling; }

}

bool ambientOnly

{

get { return settings.ambientOnly && isAmbientOnlySupported; }

}

// AO shader

Shader aoShader

{

get

{

if (\_aoShader == null)

\_aoShader = Shader.Find("Hidden/Image Effects/Cinematic/AmbientOcclusion");

return \_aoShader;

}

}

[SerializeField] Shader \_aoShader;

// Temporary aterial for the AO shader

Material aoMaterial

{

get

{

if (\_aoMaterial == null)

\_aoMaterial = ImageEffectHelper.CheckShaderAndCreateMaterial(aoShader);

return \_aoMaterial;

}

}

Material \_aoMaterial;

// Command buffer for the AO pass

CommandBuffer aoCommands

{

get

{

if (\_aoCommands == null)

{

\_aoCommands = new CommandBuffer();

\_aoCommands.name = "AmbientOcclusion";

}

return \_aoCommands;

}

}

CommandBuffer \_aoCommands;

// Target camera

Camera targetCamera

{

get { return GetComponent<Camera>(); }

}

// Property observer

PropertyObserver propertyObserver { get; set; }

// Reference to the quad mesh in the built-in assets

// (used in MRT blitting)

Mesh quadMesh

{

get { return \_quadMesh; }

}

[SerializeField] Mesh \_quadMesh;

#endregion

#region Effect Passes

// Build commands for the AO pass (used in the ambient-only mode).

void BuildAOCommands()

{

var cb = aoCommands;

var tw = targetCamera.pixelWidth;

var th = targetCamera.pixelHeight;

var ts = downsampling ? 2 : 1;

var format = RenderTextureFormat.R8;

var rwMode = RenderTextureReadWrite.Linear;

var filter = FilterMode.Bilinear;

// AO buffer

var m = aoMaterial;

var rtMask = Shader.PropertyToID("\_OcclusionTexture");

cb.GetTemporaryRT(rtMask, tw / ts, th / ts, 0, filter, format, rwMode);

// AO estimation

cb.Blit((Texture)null, rtMask, m, 0);

// Blur buffer

var rtBlur = Shader.PropertyToID("\_OcclusionBlurTexture");

// Primary blur filter (large kernel)

cb.GetTemporaryRT(rtBlur, tw, th, 0, filter, format, rwMode);

cb.SetGlobalVector("\_BlurVector", Vector2.right \* 2);

cb.Blit(rtMask, rtBlur, m, 1);

cb.ReleaseTemporaryRT(rtMask);

cb.GetTemporaryRT(rtMask, tw, th, 0, filter, format, rwMode);

cb.SetGlobalVector("\_BlurVector", Vector2.up \* 2 \* ts);

cb.Blit(rtBlur, rtMask, m, 1);

cb.ReleaseTemporaryRT(rtBlur);

// Secondary blur filter (small kernel)

cb.GetTemporaryRT(rtBlur, tw, th, 0, filter, format, rwMode);

cb.SetGlobalVector("\_BlurVector", Vector2.right \* ts);

cb.Blit(rtMask, rtBlur, m, 2);

cb.ReleaseTemporaryRT(rtMask);

cb.GetTemporaryRT(rtMask, tw, th, 0, filter, format, rwMode);

cb.SetGlobalVector("\_BlurVector", Vector2.up \* ts);

cb.Blit(rtBlur, rtMask, m, 2);

cb.ReleaseTemporaryRT(rtBlur);

// Combine AO to the G-buffer.

var mrt = new RenderTargetIdentifier[] {

BuiltinRenderTextureType.GBuffer0, // Albedo, Occ

BuiltinRenderTextureType.CameraTarget // Ambient

};

cb.SetRenderTarget(mrt, BuiltinRenderTextureType.CameraTarget);

cb.SetGlobalTexture("\_OcclusionTexture", rtMask);

cb.DrawMesh(quadMesh, Matrix4x4.identity, m, 0, 4);

cb.ReleaseTemporaryRT(rtMask);

}

// Execute the AO pass immediately (used in the forward mode).

void ExecuteAOPass(RenderTexture source, RenderTexture destination)

{

var tw = source.width;

var th = source.height;

var ts = downsampling ? 2 : 1;

var format = RenderTextureFormat.R8;

var rwMode = RenderTextureReadWrite.Linear;

// AO buffer

var m = aoMaterial;

var rtMask = RenderTexture.GetTemporary(tw / ts, th / ts, 0, format, rwMode);

// AO estimation

Graphics.Blit((Texture)null, rtMask, m, 0);

// Primary blur filter (large kernel)

var rtBlur = RenderTexture.GetTemporary(tw, th, 0, format, rwMode);

m.SetVector("\_BlurVector", Vector2.right \* 2);

Graphics.Blit(rtMask, rtBlur, m, 1);

RenderTexture.ReleaseTemporary(rtMask);

rtMask = RenderTexture.GetTemporary(tw, th, 0, format, rwMode);

m.SetVector("\_BlurVector", Vector2.up \* 2 \* ts);

Graphics.Blit(rtBlur, rtMask, m, 1);

RenderTexture.ReleaseTemporary(rtBlur);

// Secondary blur filter (small kernel)

rtBlur = RenderTexture.GetTemporary(tw, th, 0, format, rwMode);

m.SetVector("\_BlurVector", Vector2.right \* ts);

Graphics.Blit(rtMask, rtBlur, m, 2);

RenderTexture.ReleaseTemporary(rtMask);

rtMask = RenderTexture.GetTemporary(tw, th, 0, format, rwMode);

m.SetVector("\_BlurVector", Vector2.up \* ts);

Graphics.Blit(rtBlur, rtMask, m, 2);

RenderTexture.ReleaseTemporary(rtBlur);

// Combine AO with the source.

m.SetTexture("\_OcclusionTexture", rtMask);

if (!settings.debug)

Graphics.Blit(source, destination, m, 3);

else

Graphics.Blit(source, destination, m, 5);

RenderTexture.ReleaseTemporary(rtMask);

}

// Update the common material properties.

void UpdateMaterialProperties()

{

var m = aoMaterial;

m.shaderKeywords = null;

m.SetFloat("\_Intensity", intensity);

m.SetFloat("\_Radius", radius);

m.SetFloat("\_TargetScale", downsampling ? 0.5f : 1);

// Occlusion source

if (occlusionSource == OcclusionSource.GBuffer)

m.EnableKeyword("\_SOURCE\_GBUFFER");

else if (occlusionSource == OcclusionSource.DepthTexture)

m.EnableKeyword("\_SOURCE\_DEPTH");

else

m.EnableKeyword("\_SOURCE\_DEPTHNORMALS");

// Sample count

if (sampleCount == SampleCount.Lowest)

m.EnableKeyword("\_SAMPLECOUNT\_LOWEST");

else

m.SetInt("\_SampleCount", sampleCountValue);

}

#endregion

#region MonoBehaviour Functions

void OnEnable()

{

// Check if the shader is supported in the current platform.

if (!ImageEffectHelper.IsSupported(aoShader, true, false, this))

{

enabled = false;

return;

}

// Register the command buffer if in the ambient-only mode.

if (ambientOnly)

targetCamera.AddCommandBuffer(CameraEvent.BeforeReflections, aoCommands);

// Enable depth textures which the occlusion source requires.

if (occlusionSource == OcclusionSource.DepthTexture)

targetCamera.depthTextureMode |= DepthTextureMode.Depth;

if (occlusionSource != OcclusionSource.GBuffer)

targetCamera.depthTextureMode |= DepthTextureMode.DepthNormals;

}

void OnDisable()

{

// Destroy all the temporary resources.

if (\_aoMaterial != null) DestroyImmediate(\_aoMaterial);

\_aoMaterial = null;

if (\_aoCommands != null)

targetCamera.RemoveCommandBuffer(CameraEvent.BeforeReflections, \_aoCommands);

\_aoCommands = null;

}

void Update()

{

if (propertyObserver.CheckNeedsReset(settings, targetCamera))

{

// Reinitialize all the resources by disabling/enabling itself.

// This is not very efficient way but just works...

OnDisable();

OnEnable();

// Build the command buffer if in the ambient-only mode.

if (ambientOnly)

{

aoCommands.Clear();

BuildAOCommands();

}

propertyObserver.Update(settings, targetCamera);

}

// Update the material properties (later used in the AO commands).

if (ambientOnly) UpdateMaterialProperties();

}

[ImageEffectOpaque]

void OnRenderImage(RenderTexture source, RenderTexture destination)

{

if (ambientOnly)

{

// Do nothing in the ambient-only mode.

Graphics.Blit(source, destination);

}

else

{

// Execute the AO pass.

UpdateMaterialProperties();

ExecuteAOPass(source, destination);

}

}

#endregion

}

}

using System;

using UnityEngine;

using Object = UnityEngine.Object;

namespace UnityStandardAssets.Utility

{

public class ActivateTrigger : MonoBehaviour

{

// A multi-purpose script which causes an action to occur when

// a trigger collider is entered.

public enum Mode

{

Trigger = 0, // Just broadcast the action on to the target

Replace = 1, // replace target with source

Activate = 2, // Activate the target GameObject

Enable = 3, // Enable a component

Animate = 4, // Start animation on target

Deactivate = 5 // Decativate target GameObject

}

public Mode action = Mode.Activate; // The action to accomplish

public Object target; // The game object to affect. If none, the trigger work on this game object

public GameObject source;

public int triggerCount = 1;

public bool repeatTrigger = false;

private void DoActivateTrigger()

{

triggerCount--;

if (triggerCount == 0 || repeatTrigger)

{

Object currentTarget = target ?? gameObject;

Behaviour targetBehaviour = currentTarget as Behaviour;

GameObject targetGameObject = currentTarget as GameObject;

if (targetBehaviour != null)

{

targetGameObject = targetBehaviour.gameObject;

}

switch (action)

{

case Mode.Trigger:

if (targetGameObject != null)

{

targetGameObject.BroadcastMessage("DoActivateTrigger");

}

break;

case Mode.Replace:

if (source != null)

{

if (targetGameObject != null)

{

Instantiate(source, targetGameObject.transform.position,

targetGameObject.transform.rotation);

DestroyObject(targetGameObject);

}

}

break;

case Mode.Activate:

if (targetGameObject != null)

{

targetGameObject.SetActive(true);

}

break;

case Mode.Enable:

if (targetBehaviour != null)

{

targetBehaviour.enabled = true;

}

break;

case Mode.Animate:

if (targetGameObject != null)

{

targetGameObject.GetComponent<Animation>().Play();

}

break;

case Mode.Deactivate:

if (targetGameObject != null)

{

targetGameObject.SetActive(false);

}

break;

}

}

}

private void OnTriggerEnter(Collider other)

{

DoActivateTrigger();

}

}

}

// put together:

if (v.z >= 0.0)

sunShaftsMaterial.SetVector ("\_SunColor", Vector4 (sunColor.r, sunColor.g, sunColor.b, sunColor.a) \* sunShaftIntensity);

else

sunShaftsMaterial.SetVector ("\_SunColor", Vector4.zero); // no backprojection !

sunShaftsMaterial.SetTexture ("\_ColorBuffer", lrDepthBuffer);

Graphics.Blit (source, destination, sunShaftsMaterial, (screenBlendMode == ShaftsScreenBlendMode.Screen) ? 0 : 4);

RenderTexture.ReleaseTemporary (lrDepthBuffer);

RenderTexture.ReleaseTemporary (secondQuarterRezColor);

}

// helper functions

private function ClampBlurIterationsToSomethingThatMakesSense (its : int) : int {

if (its < 1)

return 1;

else if (its > 4)

return 4;

else

return its;

}

}

#pragma strict

@CustomEditor (SunShafts)

class SunShaftsEditor extends Editor

{

var serObj : SerializedObject;

var sunTransform : SerializedProperty;

var radialBlurIterations : SerializedProperty;

var sunColor : SerializedProperty;

var sunShaftBlurRadius : SerializedProperty;

var sunShaftIntensity : SerializedProperty;

var useSkyBoxAlpha : SerializedProperty;

var useDepthTexture : SerializedProperty;

var resolution : SerializedProperty;

var screenBlendMode : SerializedProperty;

var maxRadius : SerializedProperty;

function OnEnable () {

serObj = new SerializedObject (target);

screenBlendMode = serObj.FindProperty("screenBlendMode");

sunTransform = serObj.FindProperty("sunTransform");

sunColor = serObj.FindProperty("sunColor");

sunShaftBlurRadius = serObj.FindProperty("sunShaftBlurRadius");

radialBlurIterations = serObj.FindProperty("radialBlurIterations");

sunShaftIntensity = serObj.FindProperty("sunShaftIntensity");

useSkyBoxAlpha = serObj.FindProperty("useSkyBoxAlpha");

resolution = serObj.FindProperty("resolution");

maxRadius = serObj.FindProperty("maxRadius");

useDepthTexture = serObj.FindProperty("useDepthTexture");

}

function OnInspectorGUI () {

serObj.Update ();

EditorGUILayout.BeginHorizontal();

var oldVal : boolean = useDepthTexture.boolValue;

EditorGUILayout.PropertyField (useDepthTexture, new GUIContent ("Rely on Z Buffer?"));

if((target as SunShafts).camera)

GUILayout.Label("Current camera mode: "+ (target as SunShafts).camera.depthTextureMode, EditorStyles.miniBoldLabel);

EditorGUILayout.EndHorizontal();

// depth buffer need

/\*

var newVal : boolean = useDepthTexture.boolValue;

if (newVal != oldVal) {

if(newVal)

(target as SunShafts).camera.depthTextureMode |= DepthTextureMode.Depth;

else

(target as SunShafts).camera.depthTextureMode &= ~DepthTextureMode.Depth;

}

\*/

EditorGUILayout.PropertyField (resolution, new GUIContent("Resolution"));

EditorGUILayout.PropertyField (screenBlendMode, new GUIContent("Blend mode"));

EditorGUILayout.Separator ();

EditorGUILayout.BeginHorizontal();

EditorGUILayout.PropertyField (sunTransform, new GUIContent("Shafts caster", "Chose a transform that acts as a root point for the produced sun shafts"));

if((target as SunShafts).sunTransform && (target as SunShafts).camera) {

if (GUILayout.Button("Center on " + (target as SunShafts).camera.name)) {

if (EditorUtility.DisplayDialog ("Move sun shafts source?", "The SunShafts caster named "+ (target as SunShafts).sunTransform.name +"\n will be centered along "+(target as SunShafts).camera.name+". Are you sure? ", "Please do", "Don't")) {

var ray : Ray = (target as SunShafts).camera.ViewportPointToRay(Vector3(0.5,0.5,0));

(target as SunShafts).sunTransform.position = ray.origin + ray.direction \* 500.0;

(target as SunShafts).sunTransform.LookAt ((target as SunShafts).transform);

}

}

}

EditorGUILayout.EndHorizontal();

EditorGUILayout.Separator ();

EditorGUILayout.PropertyField (sunColor, new GUIContent ("Shafts color"));

maxRadius.floatValue = 1.0f - EditorGUILayout.Slider ("Distance falloff", 1.0f - maxRadius.floatValue, 0.1, 1.0);

EditorGUILayout.Separator ();

sunShaftBlurRadius.floatValue = EditorGUILayout.Slider ("Blur size", sunShaftBlurRadius.floatValue, 1.0, 10.0);

radialBlurIterations.intValue = EditorGUILayout.IntSlider ("Blur iterations", radialBlurIterations.intValue, 1, 3);

EditorGUILayout.Separator ();

EditorGUILayout.PropertyField (sunShaftIntensity, new GUIContent("Intensity"));

useSkyBoxAlpha.floatValue = EditorGUILayout.Slider ("Use alpha mask", useSkyBoxAlpha.floatValue, 0.0, 1.0);

serObj.ApplyModifiedProperties();

}

}

#pragma strict

@script ExecuteInEditMode

@script RequireComponent (Camera)

@script AddComponentMenu ("Image Effects/Tilt shift")

class TiltShift extends PostEffectsBase {

public var tiltShiftShader : Shader;

private var tiltShiftMaterial : Material = null;

public var renderTextureDivider : int = 2;

public var blurIterations : int = 2;

public var enableForegroundBlur : boolean = true;

public var foregroundBlurIterations : int = 2;

public var maxBlurSpread : float = 1.5f;

public var focalPoint : float = 30.0f;

public var smoothness : float = 1.65f;

public var visualizeCoc : boolean = false;

// these values will be automatically determined

private var start01 : float = 0.0f;

private var distance01 : float = 0.2f;

private var end01 : float = 1.0f;

private var curve : float = 1.0f;

function CheckResources () : boolean {

CheckSupport (true);

tiltShiftMaterial = CheckShaderAndCreateMaterial (tiltShiftShader, tiltShiftMaterial);

if(!isSupported)

ReportAutoDisable ();

return isSupported;

}

function OnRenderImage (source : RenderTexture, destination : RenderTexture) {

if(CheckResources()==false) {

Graphics.Blit (source, destination);

return;

}

var widthOverHeight : float = (1.0f \* source.width) / (1.0f \* source.height);

var oneOverBaseSize : float = 1.0f / 512.0f;

// clamp some values

renderTextureDivider = renderTextureDivider < 1 ? 1 : renderTextureDivider;

targetGameObject = targetBehaviour.gameObject;

}

switch (action)

{

case Mode.Trigger:

if (targetGameObject != null)

{

targetGameObject.BroadcastMessage("DoActivateTrigger");

}

break;

case Mode.Replace:

if (source != null)

{

if (targetGameObject != null)

{

Instantiate(source, targetGameObject.transform.position,

targetGameObject.transform.rotation);

DestroyObject(targetGameObject);

}

}

break;

case Mode.Activate:

if (targetGameObject != null)

{

targetGameObject.SetActive(true);

}

break;

case Mode.Enable:

if (targetBehaviour != null)

{

targetBehaviour.enabled = true;

}

break;

case Mode.Animate:

if (targetGameObject != null)

{

targetGameObject.GetComponent<Animation>().Play();

}

break;

case Mode.Deactivate:

if (targetGameObject != null)

{

targetGameObject.SetActive(false);

}

break;

}

}

}

private void OnTriggerEnter(Collider other)

{

DoActivateTrigger();

}

}

}using System;

using UnityEngine;

using Random = UnityEngine.Random;

namespace UnityStandardAssets.Vehicles.Aeroplane

{

[RequireComponent(typeof (AeroplaneController))]

public class AeroplaneAiControl : MonoBehaviour

{

// This script represents an AI 'pilot' capable of flying the plane towards a designated target.

// It sends the equivalent of the inputs that a user would send to the Aeroplane controller.

[SerializeField] private float m\_RollSensitivity = .2f; // How sensitively the AI applies the roll controls

[SerializeField] private float m\_PitchSensitivity = .5f; // How sensitively the AI applies the pitch controls

[SerializeField] private float m\_LateralWanderDistance = 5; // The amount that the plane can wander by when heading for a target

[SerializeField] private float m\_LateralWanderSpeed = 0.11f; // The speed at which the plane will wander laterally

[SerializeField] private float m\_MaxClimbAngle = 45; // The maximum angle that the AI will attempt to make plane can climb at

[SerializeField] private float m\_MaxRollAngle = 45; // The maximum angle that the AI will attempt to u

[SerializeField] private float m\_SpeedEffect = 0.01f; // This increases the effect of the controls based on the plane's speed.

[SerializeField] private float m\_TakeoffHeight = 20; // the AI will fly straight and only pitch upwards until reaching this height

[SerializeField] private Transform m\_Target; // the target to fly towards

private AeroplaneController m\_AeroplaneController; // The aeroplane controller that is used to move the plane

private float m\_RandomPerlin; // Used for generating random point on perlin noise so that the plane will wander off path slightly

private bool m\_TakenOff; // Has the plane taken off yet

// setup script properties

private void Awake()

{

// get the reference to the aeroplane controller, so we can send move input to it and read its current state.

m\_AeroplaneController = GetComponent<AeroplaneController>();

// pick a random perlin starting point for lateral wandering

m\_RandomPerlin = Random.Range(0f, 100f);

}

// reset the object to sensible values

public void Reset()

{

m\_TakenOff = false;

}

// fixed update is called in time with the physics system update

private void FixedUpdate()

{

if (m\_Target != null)

{

// make the plane wander from the path, useful for making the AI seem more human, less robotic.

Vector3 targetPos = m\_Target.position +

transform.right\*

(Mathf.PerlinNoise(Time.time\*m\_LateralWanderSpeed, m\_RandomPerlin)\*2 - 1)\*

m\_LateralWanderDistance;

// adjust the yaw and pitch towards the target

Vector3 localTarget = transform.InverseTransformPoint(targetPos);

float targetAngleYaw = Mathf.Atan2(localTarget.x, localTarget.z);

float targetAnglePitch = -Mathf.Atan2(localTarget.y, localTarget.z);

// Set the target for the planes pitch, we check later that this has not passed the maximum threshold

targetAnglePitch = Mathf.Clamp(targetAnglePitch, -m\_MaxClimbAngle\*Mathf.Deg2Rad,

m\_MaxClimbAngle\*Mathf.Deg2Rad);

// calculate the difference between current pitch and desired pitch

float changePitch = targetAnglePitch - m\_AeroplaneController.PitchAngle;

// AI always applies gentle forward throttle

const float throttleInput = 0.5f;

// AI applies elevator control (pitch, rotation around x) to reach the target angle

float pitchInput = changePitch\*m\_PitchSensitivity;

// clamp the planes roll

float desiredRoll = Mathf.Clamp(targetAngleYaw, -m\_MaxRollAngle\*Mathf.Deg2Rad, m\_MaxRollAngle\*Mathf.Deg2Rad);

float yawInput = 0;

float rollInput = 0;

if (!m\_TakenOff)

{

// If the planes altitude is above m\_TakeoffHeight we class this as taken off

if (m\_AeroplaneController.Altitude > m\_TakeoffHeight)

{

m\_TakenOff = true;

}

}

else

{

// now we have taken off to a safe height, we can use the rudder and ailerons to yaw and roll

yawInput = targetAngleYaw;

rollInput = -(m\_AeroplaneController.RollAngle - desiredRoll)\*m\_RollSensitivity;

}

// adjust how fast the AI is changing the controls based on the speed. Faster speed = faster on the controls.

float currentSpeedEffect = 1 + (m\_AeroplaneController.ForwardSpeed\*m\_SpeedEffect);

rollInput \*= currentSpeedEffect;

pitchInput \*= currentSpeedEffect;

yawInput \*= currentSpeedEffect;

// pass the current input to the plane (false = because AI never uses air brakes!)

m\_AeroplaneController.Move(rollInput, pitchInput, yawInput, throttleInput, false);

}

else

{

// no target set, send zeroed input to the planeW

m\_AeroplaneController.Move(0, 0, 0, 0, false);

}

}

// allows other scripts to set the plane's target

public void SetTarget(Transform target)

{

m\_Target = target;

}

}

}using System;

using UnityEngine;

namespace UnityStandardAssets.Vehicles.Aeroplane

{

public class AeroplaneAudio : MonoBehaviour

{

[Serializable]

public class AdvancedSetttings // A class for storing the advanced options.

{

public float engineMinDistance = 50f; // The min distance of the engine audio source.

public float engineMaxDistance = 1000f; // The max distance of the engine audio source.

public float engineDopplerLevel = 1f; // The doppler level of the engine audio source.

[Range(0f, 1f)] public float engineMasterVolume = 0.5f; // An overall control of the engine sound volume.

public float windMinDistance = 10f; // The min distance of the wind audio source.

public float windMaxDistance = 100f; // The max distance of the wind audio source.

public float windDopplerLevel = 1f; // The doppler level of the wind audio source.

[Range(0f, 1f)] public float windMasterVolume = 0.5f; // An overall control of the wind sound volume.

}

[SerializeField] private AudioClip m\_EngineSound; // Looped engine sound, whose pitch and volume are affected by the plane's throttle setting.

[SerializeField] private float m\_EngineMinThrottlePitch = 0.4f; // Pitch of the engine sound when at minimum throttle.

[SerializeField] private float m\_EngineMaxThrottlePitch = 2f; // Pitch of the engine sound when at maximum throttle.

[SerializeField] private float m\_EngineFwdSpeedMultiplier = 0.002f; // Additional multiplier for an increase in pitch of the engine from the plane's speed.

[SerializeField] private AudioClip m\_WindSound; // Looped wind sound, whose pitch and volume are affected by the plane's velocity.

[SerializeField] private float m\_WindBasePitch = 0.2f; // starting pitch for wind (when plane is at zero speed)

[SerializeField] private float m\_WindSpeedPitchFactor = 0.004f; // Relative increase in pitch of the wind from the plane's speed.

[SerializeField] private float m\_WindMaxSpeedVolume = 100; // the speed the aircraft much reach before the wind sound reaches maximum volume.

[SerializeField] private AdvancedSetttings m\_AdvancedSetttings = new AdvancedSetttings();// container to make advanced settings appear as rollout in inspector

private AudioSource m\_EngineSoundSource; // Reference to the AudioSource for the engine.

private AudioSource m\_WindSoundSource; // Reference to the AudioSource for the wind.

private AeroplaneController m\_Plane; // Reference to the aeroplane controller.

private Rigidbody m\_Rigidbody;

private void Awake()

{

// Set up the reference to the aeroplane controller.

m\_Plane = GetComponent<AeroplaneController>();

m\_Rigidbody = GetComponent<Rigidbody>();

// Add the audiosources and get the references.

m\_EngineSoundSource = gameObject.AddComponent<AudioSource>();

m\_EngineSoundSource.playOnAwake = false;

m\_WindSoundSource = gameObject.AddComponent<AudioSource>();

m\_WindSoundSource.playOnAwake = false;

// Assign clips to the audiosources.

m\_EngineSoundSource.clip = m\_EngineSound;

m\_WindSoundSource.clip = m\_WindSound;

// Set the parameters of the audiosources.

m\_EngineSoundSource.minDistance = m\_AdvancedSetttings.engineMinDistance;

m\_EngineSoundSource.maxDistance = m\_AdvancedSetttings.engineMaxDistance;

m\_EngineSoundSource.loop = true;

m\_EngineSoundSource.dopplerLevel = m\_AdvancedSetttings.engineDopplerLevel;

m\_WindSoundSource.minDistance = m\_AdvancedSetttings.windMinDistance;

m\_WindSoundSource.maxDistance = m\_AdvancedSetttings.windMaxDistance;

m\_WindSoundSource.loop = true;

m\_WindSoundSource.dopplerLevel = m\_AdvancedSetttings.windDopplerLevel;

// call update here to set the sounds pitch and volumes before they actually play

Update();

// Start the sounds playing.

m\_EngineSoundSource.Play();

m\_WindSoundSource.Play();

}

private void Update()

{

// Find what proportion of the engine's power is being used.

var enginePowerProportion = Mathf.InverseLerp(0, m\_Plane.MaxEnginePower, m\_Plane.EnginePower);

// Set the engine's pitch to be proportional to the engine's current power.

m\_EngineSoundSource.pitch = Mathf.Lerp(m\_EngineMinThrottlePitch, m\_EngineMaxThrottlePitch, enginePowerProportion);

// Increase the engine's pitch by an amount proportional to the aeroplane's forward speed.

// (this makes the pitch increase when going into a dive!)

m\_EngineSoundSource.pitch += m\_Plane.ForwardSpeed\*m\_EngineFwdSpeedMultiplier;

// Set the engine's volume to be proportional to the engine's current power.

m\_EngineSoundSource.volume = Mathf.InverseLerp(0, m\_Plane.MaxEnginePower\*m\_AdvancedSetttings.engineMasterVolume,

m\_Plane.EnginePower);

// Set the wind's pitch and volume to be proportional to the aeroplane's forward speed.

float planeSpeed = m\_Rigidbody.velocity.magnitude;

m\_WindSoundSource.pitch = m\_WindBasePitch + planeSpeed\*m\_WindSpeedPitchFactor;

m\_WindSoundSource.volume = Mathf.InverseLerp(0, m\_WindMaxSpeedVolume, planeSpeed)\*m\_AdvancedSetttings.windMasterVolume;

}

}

}

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;

}

}

}using System;

using UnityEngine;

namespace UnityStandardAssets.Vehicles.Aeroplane

{

public class AeroplaneControlSurfaceAnimator : MonoBehaviour

{

[SerializeField] private float m\_Smoothing = 5f; // The smoothing applied to the movement of control surfaces.

[SerializeField] private ControlSurface[] m\_ControlSurfaces; // Collection of control surfaces.

private AeroplaneController m\_Plane; // Reference to the aeroplane controller.

private void Start()

{

// Get the reference to the aeroplane controller.

m\_Plane = GetComponent<AeroplaneController>();

// Store the original local rotation of each surface, so we can rotate relative to this

foreach (var surface in m\_ControlSurfaces)

{

surface.originalLocalRotation = surface.transform.localRotation;

}

}

private void Update()

{

foreach (var surface in m\_ControlSurfaces)

{

switch (surface.type)

{

case ControlSurface.Type.Aileron:

{

// Ailerons rotate around the x axis, according to the plane's roll input

Quaternion rotation = Quaternion.Euler(surface.amount\*m\_Plane.RollInput, 0f, 0f);

RotateSurface(surface, rotation);

break;

}

case ControlSurface.Type.Elevator:

{

// Elevators rotate negatively around the x axis, according to the plane's pitch input

Quaternion rotation = Quaternion.Euler(surface.amount\*-m\_Plane.PitchInput, 0f, 0f);

RotateSurface(surface, rotation);

break;

}

case ControlSurface.Type.Rudder:

{

// Rudders rotate around their y axis, according to the plane's yaw input

Quaternion rotation = Quaternion.Euler(0f, surface.amount\*m\_Plane.YawInput, 0f);

RotateSurface(surface, rotation);

break;

}

case ControlSurface.Type.RuddervatorPositive:

{

// Ruddervators are a combination of rudder and elevator, and rotate

// around their z axis by a combination of the yaw and pitch input

float r = m\_Plane.YawInput + m\_Plane.PitchInput;

Quaternion rotation = Quaternion.Euler(0f, 0f, surface.amount\*r);

RotateSurface(surface, rotation);

break;

}

case ControlSurface.Type.RuddervatorNegative:

{

// ... and because ruddervators are "special", we need a negative version too. >\_<

float r = m\_Plane.YawInput - m\_Plane.PitchInput;

Quaternion rotation = Quaternion.Euler(0f, 0f, surface.amount\*r);

RotateSurface(surface, rotation);

break;

}

}

}

}

private void RotateSurface(ControlSurface surface, Quaternion rotation)

{

// Create a target which is the surface's original rotation, rotated by the input.

Quaternion target = surface.originalLocalRotation\*rotation;

// Slerp the surface's rotation towards the target rotation.

surface.transform.localRotation = Quaternion.Slerp(surface.transform.localRotation, target,

m\_Smoothing\*Time.deltaTime);

}

// This class presents a nice custom structure in which to define each of the plane's contol surfaces to animate.

// They show up in the inspector as an array.

[Serializable]

public class ControlSurface // Control surfaces represent the different flaps of the aeroplane.

{

public enum Type // Flaps differ in position and rotation and are represented by different types.

{

Aileron, // Horizontal flaps on the wings, rotate on the x axis.

Elevator, // Horizontal flaps used to adjusting the pitch of a plane, rotate on the x axis.

Rudder, // Vertical flaps on the tail, rotate on the y axis.

RuddervatorNegative, // Combination of rudder and elevator.

RuddervatorPositive, // Combination of rudder and elevator.

}

public Transform transform; // The transform of the control surface.

public float amount; // The amount by which they can rotate.

public Type type; // The type of control surface.

[HideInInspector] public Quaternion originalLocalRotation; // The rotation of the surface at the start.

}

using System;

using UnityEngine;

namespace UnityStandardAssets.Vehicles.Aeroplane

{

public class AeroplanePropellerAnimator : MonoBehaviour

{

[SerializeField] private Transform m\_PropellorModel; // The model of the the aeroplane's propellor.

[SerializeField] private Transform m\_PropellorBlur; // The plane used for the blurred propellor textures.

[SerializeField] private Texture2D[] m\_PropellorBlurTextures; // An array of increasingly blurred propellor textures.

[SerializeField] [Range(0f, 1f)] private float m\_ThrottleBlurStart = 0.25f; // The point at which the blurred textures start.

[SerializeField] [Range(0f, 1f)] private float m\_ThrottleBlurEnd = 0.5f; // The point at which the blurred textures stop changing.

[SerializeField] private float m\_MaxRpm = 2000; // The maximum speed the propellor can turn at.

private AeroplaneController m\_Plane; // Reference to the aeroplane controller.

private int m\_PropellorBlurState = -1; // To store the state of the blurred textures.

private const float k\_RpmToDps = 60f; // For converting from revs per minute to degrees per second.

private Renderer m\_PropellorModelRenderer;

private Renderer m\_PropellorBlurRenderer;

private void Awake()

{

// Set up the reference to the aeroplane controller.

m\_Plane = GetComponent<AeroplaneController>();

m\_PropellorModelRenderer = m\_PropellorModel.GetComponent<Renderer>();

m\_PropellorBlurRenderer = m\_PropellorBlur.GetComponent<Renderer>();

// Set the propellor blur gameobject's parent to be the propellor.

m\_PropellorBlur.parent = m\_PropellorModel;

}

private void Update()

{

// Rotate the propellor model at a rate proportional to the throttle.

m\_PropellorModel.Rotate(0, m\_MaxRpm\*m\_Plane.Throttle\*Time.deltaTime\*k\_RpmToDps, 0);

// Create an integer for the new state of the blur textures.

var newBlurState = 0;

// choose between the blurred textures, if the throttle is high enough

if (m\_Plane.Throttle > m\_ThrottleBlurStart)

{

var throttleBlurProportion = Mathf.InverseLerp(m\_ThrottleBlurStart, m\_ThrottleBlurEnd, m\_Plane.Throttle);

newBlurState = Mathf.FloorToInt(throttleBlurProportion\*(m\_PropellorBlurTextures.Length - 1));

}

// If the blur state has changed

if (newBlurState != m\_PropellorBlurState)

{

m\_PropellorBlurState = newBlurState;

if (m\_PropellorBlurState == 0)

{

// switch to using the 'real' propellor model

m\_PropellorModelRenderer.enabled = true;

m\_PropellorBlurRenderer.enabled = false;

}

else

{

// Otherwise turn off the propellor model and turn on the blur.

m\_PropellorModelRenderer.enabled = false;

m\_PropellorBlurRenderer.enabled = true;

// set the appropriate texture from the blur array

m\_PropellorBlurRenderer.material.mainTexture = m\_PropellorBlurTextures[m\_PropellorBlurState];

}

}

}

}

}

using System;

using UnityEngine;

using UnityStandardAssets.CrossPlatformInput;

namespace UnityStandardAssets.Vehicles.Aeroplane

{

[RequireComponent(typeof (AeroplaneController))]

public class AeroplaneUserControl2Axis : MonoBehaviour

{

// these max angles are only used on mobile, due to the way pitch and roll input are handled

public float maxRollAngle = 80;

public float maxPitchAngle = 80;

// reference to the aeroplane that we're controlling

private AeroplaneController m\_Aeroplane;

private void Awake()

{

// Set up the reference to the aeroplane controller.

m\_Aeroplane = GetComponent<AeroplaneController>();

}

private void FixedUpdate()

{

// Read input for the pitch, yaw, roll and throttle of the aeroplane.

float roll = CrossPlatformInputManager.GetAxis("Horizontal");

float pitch = CrossPlatformInputManager.GetAxis("Vertical");

bool airBrakes = CrossPlatformInputManager.GetButton("Fire1");

// auto throttle up, or down if braking.

float throttle = airBrakes ? -1 : 1;

#if MOBILE\_INPUT

AdjustInputForMobileControls(ref roll, ref pitch, ref throttle);

#endif

// Pass the input to the aeroplane

m\_Aeroplane.Move(roll, pitch, 0, throttle, airBrakes);

}

private void AdjustInputForMobileControls(ref float roll, ref float pitch, ref float throttle)

{

// because mobile tilt is used for roll and pitch, we help out by

// assuming that a centered level device means the user

// wants to fly straight and level!

// this means on mobile, the input represents the \*desired\* roll angle of the aeroplane,

// and the roll input is calculated to achieve that.

// whereas on non-mobile, the input directly controls the roll of the aeroplane.

float intendedRollAngle = roll\*maxRollAngle\*Mathf.Deg2Rad;

float intendedPitchAngle = pitch\*maxPitchAngle\*Mathf.Deg2Rad;

roll = Mathf.Clamp((intendedRollAngle - m\_Aeroplane.RollAngle), -1, 1);

pitch = Mathf.Clamp((intendedPitchAngle - m\_Aeroplane.PitchAngle), -1, 1);

// similarly, the throttle axis input is considered to be the desired absolute value, not a relative change to current throttle.

float intendedThrottle = throttle\*0.5f + 0.5f;

throttle = Mathf.Clamp(intendedThrottle - m\_Aeroplane.Throttle, -1, 1);

}

}

}

using System;

using UnityEngine;

using UnityStandardAssets.CrossPlatformInput;

namespace UnityStandardAssets.Vehicles.Aeroplane

{

[RequireComponent(typeof (AeroplaneController))]

public class AeroplaneUserControl4Axis : MonoBehaviour

{

// these max angles are only used on mobile, due to the way pitch and roll input are handled

public float maxRollAngle = 80;

public float maxPitchAngle = 80;

// reference to the aeroplane that we're controlling

private AeroplaneController m\_Aeroplane;

private float m\_Throttle;

private bool m\_AirBrakes;

private float m\_Yaw;

private void Awake()

{

// Set up the reference to the aeroplane controller.

m\_Aeroplane = GetComponent<AeroplaneController>();

}

private void FixedUpdate()

{

// Read input for the pitch, yaw, roll and throttle of the aeroplane.

float roll = CrossPlatformInputManager.GetAxis("Mouse X");

float pitch = CrossPlatformInputManager.GetAxis("Mouse Y");

m\_AirBrakes = CrossPlatformInputManager.GetButton("Fire1");

m\_Yaw = CrossPlatformInputManager.GetAxis("Horizontal");

m\_Throttle = CrossPlatformInputManager.GetAxis("Vertical");

#if MOBILE\_INPUT

AdjustInputForMobileControls(ref roll, ref pitch, ref m\_Throttle);

#endif

// Pass the input to the aeroplane

m\_Aeroplane.Move(roll, pitch, m\_Yaw, m\_Throttle, m\_AirBrakes);

}

private void AdjustInputForMobileControls(ref float roll, ref float pitch, ref float throttle)

{

// because mobile tilt is used for roll and pitch, we help out by

// assuming that a centered level device means the user

// wants to fly straight and level!

// this means on mobile, the input represents the \*desired\* roll angle of the aeroplane,

// and the roll input is calculated to achieve that.

// whereas on non-mobile, the input directly controls the roll of the aeroplane.

float intendedRollAngle = roll\*maxRollAngle\*Mathf.Deg2Rad;

float intendedPitchAngle = pitch\*maxPitchAngle\*Mathf.Deg2Rad;

roll = Mathf.Clamp((intendedRollAngle - m\_Aeroplane.RollAngle), -1, 1);

pitch = Mathf.Clamp((intendedPitchAngle - m\_Aeroplane.PitchAngle), -1, 1);

}

}

}using System;

using UnityEngine;

namespace UnityStandardAssets.Effects

{

[RequireComponent(typeof (SphereCollider))]

public class AfterburnerPhysicsForce : MonoBehaviour

{

public float effectAngle = 15;

public float effectWidth = 1;

public float effectDistance = 10;

public float force = 10;

private Collider[] m\_Cols;

private SphereCollider m\_Sphere;

private void OnEnable()

{

m\_Sphere = (GetComponent<Collider>() as SphereCollider);

}

private void FixedUpdate()

{

m\_Cols = Physics.OverlapSphere(transform.position + m\_Sphere.center, m\_Sphere.radius);

for (int n = 0; n < m\_Cols.Length; ++n)

{

if (m\_Cols[n].attachedRigidbody != null)

{

Vector3 localPos = transform.InverseTransformPoint(m\_Cols[n].transform.position);

localPos = Vector3.MoveTowards(localPos, new Vector3(0, 0, localPos.z), effectWidth\*0.5f);

float angle = Mathf.Abs(Mathf.Atan2(localPos.x, localPos.z)\*Mathf.Rad2Deg);

float falloff = Mathf.InverseLerp(effectDistance, 0, localPos.magnitude);

falloff \*= Mathf.InverseLerp(effectAngle, 0, angle);

Vector3 delta = m\_Cols[n].transform.position - transform.position;

m\_Cols[n].attachedRigidbody.AddForceAtPosition(delta.normalized\*force\*falloff,

Vector3.Lerp(m\_Cols[n].transform.position,

transform.TransformPoint(0, 0, localPos.z),

0.1f));

}

}

}

private void OnDrawGizmosSelected()

{

//check for editor time simulation to avoid null ref

if(m\_Sphere == null)

m\_Sphere = (GetComponent<Collider>() as SphereCollider);

m\_Sphere.radius = effectDistance\*.5f;

m\_Sphere.center = new Vector3(0, 0, effectDistance\*.5f);

var directions = new Vector3[] {Vector3.up, -Vector3.up, Vector3.right, -Vector3.right};

var perpDirections = new Vector3[] {-Vector3.right, Vector3.right, Vector3.up, -Vector3.up};

Gizmos.color = new Color(0, 1, 0, 0.5f);

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

{

Vector3 origin = transform.position + transform.rotation\*directions[n]\*effectWidth\*0.5f;

Vector3 direction =

transform.TransformDirection(Quaternion.AngleAxis(effectAngle, perpDirections[n])\*Vector3.forward);

Gizmos.DrawLine(origin, origin + direction\*m\_Sphere.radius\*2);

}

}

}

}

using System;

using UnityEngine;

namespace UnityStandardAssets.Characters.ThirdPerson

{

[RequireComponent(typeof (UnityEngine.AI.NavMeshAgent))]

[RequireComponent(typeof (ThirdPersonCharacter))]

public class AICharacterControl : MonoBehaviour

{

public UnityEngine.AI.NavMeshAgent agent { get; private set; } // the navmesh agent required for the path finding

public ThirdPersonCharacter character { get; private set; } // the character we are controlling

public Transform target; // target to aim for

private void Start()

{

// get the components on the object we need ( should not be null due to require component so no need to check )

agent = GetComponentInChildren<UnityEngine.AI.NavMeshAgent>();

character = GetComponent<ThirdPersonCharacter>();

agent.updateRotation = false;

agent.updatePosition = true;

}

private void Update()

{

if (target != null)

agent.SetDestination(target.position);

if (agent.remainingDistance > agent.stoppingDistance)

character.Move(agent.desiredVelocity, false, false);

else

character.Move(Vector3.zero, false, false);

}

public void SetTarget(Transform target)

{

this.target = target;

}

}

}using UnityEngine.PostProcessing;

namespace UnityEditor.PostProcessing

{

using Settings = AmbientOcclusionModel.Settings;

[PostProcessingModelEditor(typeof(AmbientOcclusionModel))]

public class AmbientOcclusionModelEditor : PostProcessingModelEditor

{

SerializedProperty m\_Intensity;

SerializedProperty m\_Radius;

SerializedProperty m\_SampleCount;

SerializedProperty m\_Downsampling;

SerializedProperty m\_ForceForwardCompatibility;

SerializedProperty m\_AmbientOnly;

SerializedProperty m\_HighPrecision;

public override void OnEnable()

{

m\_Intensity = FindSetting((Settings x) => x.intensity);

m\_Radius = FindSetting((Settings x) => x.radius);

m\_SampleCount = FindSetting((Settings x) => x.sampleCount);

m\_Downsampling = FindSetting((Settings x) => x.downsampling);

m\_ForceForwardCompatibility = FindSetting((Settings x) => x.forceForwardCompatibility);

m\_AmbientOnly = FindSetting((Settings x) => x.ambientOnly);

m\_HighPrecision = FindSetting((Settings x) => x.highPrecision);

}

public override void OnInspectorGUI()

{

EditorGUILayout.PropertyField(m\_Intensity);

EditorGUILayout.PropertyField(m\_Radius);

EditorGUILayout.PropertyField(m\_SampleCount);

EditorGUILayout.PropertyField(m\_Downsampling);

EditorGUILayout.PropertyField(m\_ForceForwardCompatibility);

EditorGUILayout.PropertyField(m\_HighPrecision, EditorGUIHelper.GetContent("High Precision (Forward)"));

using (new EditorGUI.DisabledGroupScope(m\_ForceForwardCompatibility.boolValue))

EditorGUILayout.PropertyField(m\_AmbientOnly, EditorGUIHelper.GetContent("Ambient Only (Deferred + HDR)"));

}

}

}

using System;

using UnityEngine;

namespace UnityStandardAssets.ImageEffects

{

public enum AAMode

{

FXAA2 = 0,

FXAA3Console = 1,

FXAA1PresetA = 2,

FXAA1PresetB = 3,

NFAA = 4,

SSAA = 5,

DLAA = 6,

}

[ExecuteInEditMode]

[RequireComponent(typeof (Camera))]

[AddComponentMenu("Image Effects/Other/Antialiasing")]

public class Antialiasing : PostEffectsBase

{

public AAMode mode = AAMode.FXAA3Console;

public bool showGeneratedNormals = false;

public float offsetScale = 0.2f;

public float blurRadius = 18.0f;

public float edgeThresholdMin = 0.05f;

public float edgeThreshold = 0.2f;

public float edgeSharpness = 4.0f;

public bool dlaaSharp = false;

public Shader ssaaShader;

private Material ssaa;

public Shader dlaaShader;

private Material dlaa;

public Shader nfaaShader;

private Material nfaa;

public Shader shaderFXAAPreset2;

private Material materialFXAAPreset2;

public Shader shaderFXAAPreset3;

private Material materialFXAAPreset3;

public Shader shaderFXAAII;

private Material materialFXAAII;

public Shader shaderFXAAIII;

private Material materialFXAAIII;

public Material CurrentAAMaterial()

{

Material returnValue = null;

switch (mode)

{

case AAMode.FXAA3Console:

returnValue = materialFXAAIII;

break;

case AAMode.FXAA2:

returnValue = materialFXAAII;

break;

case AAMode.FXAA1PresetA:

returnValue = materialFXAAPreset2;

break;

case AAMode.FXAA1PresetB:

returnValue = materialFXAAPreset3;

break;

case AAMode.NFAA:

returnValue = nfaa;

break;

case AAMode.SSAA:

returnValue = ssaa;

break;

case AAMode.DLAA:

returnValue = dlaa;

break;

default:

returnValue = null;

break;

}

return returnValue;

}

public override bool CheckResources()

{

CheckSupport(false);

materialFXAAPreset2 = CreateMaterial(shaderFXAAPreset2, materialFXAAPreset2);

materialFXAAPreset3 = CreateMaterial(shaderFXAAPreset3, materialFXAAPreset3);

materialFXAAII = CreateMaterial(shaderFXAAII, materialFXAAII);

materialFXAAIII = CreateMaterial(shaderFXAAIII, materialFXAAIII);

nfaa = CreateMaterial(nfaaShader, nfaa);

ssaa = CreateMaterial(ssaaShader, ssaa);

dlaa = CreateMaterial(dlaaShader, dlaa);

if (!ssaaShader.isSupported)

{

NotSupported();

ReportAutoDisable();

}

return isSupported;

}

public void OnRenderImage(RenderTexture source, RenderTexture destination)

{

if (CheckResources() == false)

{

Graphics.Blit(source, destination);

return;

}

// ----------------------------------------------------------------

// FXAA antialiasing modes

if (mode == AAMode.FXAA3Console && (materialFXAAIII != null))

{

materialFXAAIII.SetFloat("\_EdgeThresholdMin", edgeThresholdMin);

materialFXAAIII.SetFloat("\_EdgeThreshold", edgeThreshold);

materialFXAAIII.SetFloat("\_EdgeSharpness", edgeSharpness);

Graphics.Blit(source, destination, materialFXAAIII);

}

else if (mode == AAMode.FXAA1PresetB && (materialFXAAPreset3 != null))

{

Graphics.Blit(source, destination, materialFXAAPreset3);

}

else if (mode == AAMode.FXAA1PresetA && materialFXAAPreset2 != null)

{

source.anisoLevel = 4;

Graphics.Blit(source, destination, materialFXAAPreset2);

source.anisoLevel = 0;

}

else if (mode == AAMode.FXAA2 && materialFXAAII != null)

{

Graphics.Blit(source, destination, materialFXAAII);

}

else if (mode == AAMode.SSAA && ssaa != null)

{

// ----------------------------------------------------------------

// SSAA antialiasing

Graphics.Blit(source, destination, ssaa);

}

else if (mode == AAMode.DLAA && dlaa != null)

{

// ----------------------------------------------------------------

// DLAA antialiasing

source.anisoLevel = 0;

RenderTexture interim = RenderTexture.GetTemporary(source.width, source.height);

Graphics.Blit(source, interim, dlaa, 0);

Graphics.Blit(interim, destination, dlaa, dlaaSharp ? 2 : 1);

RenderTexture.ReleaseTemporary(interim);

}

else if (mode == AAMode.NFAA && nfaa != null)

{

// ----------------------------------------------------------------

// nfaa antialiasing

source.anisoLevel = 0;

nfaa.SetFloat("\_OffsetScale", offsetScale);

nfaa.SetFloat("\_BlurRadius", blurRadius);

Graphics.Blit(source, destination, nfaa, showGeneratedNormals ? 1 : 0);

}

else

{

// none of the AA is supported, fallback to a simple blit

Graphics.Blit(source, destination);

}

}

}

}using System;

namespace UnityEngine.PostProcessing

{

[Serializable]

public class AntialiasingModel : PostProcessingModel

{

public enum Method

{

Fxaa,

Taa

}

// Most settings aren't exposed to the user anymore, presets are enough. Still, I'm leaving

// the tooltip attributes in case an user wants to customize each preset.

#region FXAA Settings

public enum FxaaPreset

{

ExtremePerformance,

Performance,

Default,

Quality,

ExtremeQuality

}

[Serializable]

public struct FxaaQualitySettings

{

[Tooltip("The amount of desired sub-pixel aliasing removal. Effects the sharpeness of the output.")]

[Range(0f, 1f)]

public float subpixelAliasingRemovalAmount;

[Tooltip("The minimum amount of local contrast required to qualify a region as containing an edge.")]

[Range(0.063f, 0.333f)]

public float edgeDetectionThreshold;

[Tooltip("Local contrast adaptation value to disallow the algorithm from executing on the darker regions.")]

[Range(0f, 0.0833f)]

public float minimumRequiredLuminance;

public static FxaaQualitySettings[] presets =

{

// ExtremePerformance

new FxaaQualitySettings

{

subpixelAliasingRemovalAmount = 0f,

edgeDetectionThreshold = 0.333f,

minimumRequiredLuminance = 0.0833f

},

// Performance

new FxaaQualitySettings

{

subpixelAliasingRemovalAmount = 0.25f,

edgeDetectionThreshold = 0.25f,

minimumRequiredLuminance = 0.0833f

},

// Default

new FxaaQualitySettings

{

subpixelAliasingRemovalAmount = 0.75f,

edgeDetectionThreshold = 0.166f,

minimumRequiredLuminance = 0.0833f

},

// Quality

new FxaaQualitySettings

{

subpixelAliasingRemovalAmount = 1f,

edgeDetectionThreshold = 0.125f,

minimumRequiredLuminance = 0.0625f

},

// ExtremeQuality

new FxaaQualitySettings

{

subpixelAliasingRemovalAmount = 1f,

edgeDetectionThreshold = 0.063f,

minimumRequiredLuminance = 0.0312f

}

};

}

[Serializable]

public struct FxaaConsoleSettings

{

[Tooltip("The amount of spread applied to the sampling coordinates while sampling for subpixel information.")]

[Range(0.33f, 0.5f)]

public float subpixelSpreadAmount;

[Tooltip("This value dictates how sharp the edges in the image are kept; a higher value implies sharper edges.")]

[Range(2f, 8f)]

public float edgeSharpnessAmount;

[Tooltip("The minimum amount of local contrast required to qualify a region as containing an edge.")]

[Range(0.125f, 0.25f)]

public float edgeDetectionThreshold;

[Tooltip("Local contrast adaptation value to disallow the algorithm from executing on the darker regions.")]

[Range(0.04f, 0.06f)]

public float minimumRequiredLuminance;

public static FxaaConsoleSettings[] presets =

{

// ExtremePerformance

new FxaaConsoleSettings

{

subpixelSpreadAmount = 0.33f,

edgeSharpnessAmount = 8f,

edgeDetectionThreshold = 0.25f,

minimumRequiredLuminance = 0.06f

},

// Performance

new FxaaConsoleSettings

{

subpixelSpreadAmount = 0.33f,

edgeSharpnessAmount = 8f,

edgeDetectionThreshold = 0.125f,

minimumRequiredLuminance = 0.06f

},

// Default

new FxaaConsoleSettings

{

subpixelSpreadAmount = 0.5f,

edgeSharpnessAmount = 8f,

edgeDetectionThreshold = 0.125f,

minimumRequiredLuminance = 0.05f

},

// Quality

new FxaaConsoleSettings

{

subpixelSpreadAmount = 0.5f,

edgeSharpnessAmount = 4f,

edgeDetectionThreshold = 0.125f,

minimumRequiredLuminance = 0.04f

},

// ExtremeQuality

new FxaaConsoleSettings

{

subpixelSpreadAmount = 0.5f,

edgeSharpnessAmount = 2f,

edgeDetectionThreshold = 0.125f,

minimumRequiredLuminance = 0.04f

}

};

}

[Serializable]

public struct FxaaSettings

{

public FxaaPreset preset;

public static FxaaSettings defaultSettings

{

get

{

return new FxaaSettings

{

preset = FxaaPreset.Default

};

}

}

}

#endregion

#region TAA Settings

[Serializable]

public struct TaaSettings

{

[Tooltip("The diameter (in texels) inside which jitter samples are spread. Smaller values result in crisper but more aliased output, while larger values result in more stable but blurrier output.")]

[Range(0.1f, 1f)]

public float jitterSpread;

[Tooltip("Controls the amount of sharpening applied to the color buffer.")]

[Range(0f, 3f)]

public float sharpen;

[Tooltip("The blend coefficient for a stationary fragment. Controls the percentage of history sample blended into the final color.")]

[Range(0f, 0.99f)]

public float stationaryBlending;

[Tooltip("The blend coefficient for a fragment with significant motion. Controls the percentage of history sample blended into the final color.")]

[Range(0f, 0.99f)]

public float motionBlending;

public static TaaSettings defaultSettings

{

get

{

return new TaaSettings

{

jitterSpread = 0.75f,

sharpen = 0.3f,

stationaryBlending = 0.95f,

motionBlending = 0.85f

};

}

}

}

#endregion

[Serializable]

public struct Settings

{

public Method method;

public FxaaSettings fxaaSettings;

public TaaSettings taaSettings;

public static Settings defaultSettings

{

get

{

return new Settings

{

method = Method.Fxaa,

fxaaSettings = FxaaSettings.defaultSettings,

taaSettings = TaaSettings.defaultSettings

};

}

}

}

[SerializeField]

Settings m\_Settings = Settings.defaultSettings;

public Settings settings

{

get { return m\_Settings; }

set { m\_Settings = value; }

}

public override void Reset()

{

m\_Settings = Settings.defaultSettings;

}

}

}

using UnityEngine;

using UnityEngine.PostProcessing;

namespace UnityEditor.PostProcessing

{

using Method = AntialiasingModel.Method;

using Settings = AntialiasingModel.Settings;

[PostProcessingModelEditor(typeof(AntialiasingModel))]

public class AntialiasingModelEditor : PostProcessingModelEditor

{

SerializedProperty m\_Method;

SerializedProperty m\_FxaaPreset;

SerializedProperty m\_TaaJitterSpread;

SerializedProperty m\_TaaSharpen;

SerializedProperty m\_TaaStationaryBlending;

SerializedProperty m\_TaaMotionBlending;

static string[] s\_MethodNames =

{

"Fast Approximate Anti-aliasing",

"Temporal Anti-aliasing"

};

public override void OnEnable()

{

m\_Method = FindSetting((Settings x) => x.method);

m\_FxaaPreset = FindSetting((Settings x) => x.fxaaSettings.preset);

m\_TaaJitterSpread = FindSetting((Settings x) => x.taaSettings.jitterSpread);

m\_TaaSharpen = FindSetting((Settings x) => x.taaSettings.sharpen);

m\_TaaStationaryBlending = FindSetting((Settings x) => x.taaSettings.stationaryBlending);

m\_TaaMotionBlending = FindSetting((Settings x) => x.taaSettings.motionBlending);

}

public override void OnInspectorGUI()

{

m\_Method.intValue = EditorGUILayout.Popup("Method", m\_Method.intValue, s\_MethodNames);

if (m\_Method.intValue == (int)Method.Fxaa)

{

EditorGUILayout.PropertyField(m\_FxaaPreset);

}

else if (m\_Method.intValue == (int)Method.Taa)

{

if (QualitySettings.antiAliasing > 1)

EditorGUILayout.HelpBox("Temporal Anti-Aliasing doesn't work correctly when MSAA is enabled.", MessageType.Warning);

EditorGUILayout.LabelField("Jitter", EditorStyles.boldLabel);

EditorGUI.indentLevel++;

EditorGUILayout.PropertyField(m\_TaaJitterSpread, EditorGUIHelper.GetContent("Spread"));

EditorGUI.indentLevel--;

EditorGUILayout.Space();

EditorGUILayout.LabelField("Blending", EditorStyles.boldLabel);

EditorGUI.indentLevel++;

EditorGUILayout.PropertyField(m\_TaaStationaryBlending, EditorGUIHelper.GetContent("Stationary"));

EditorGUILayout.PropertyField(m\_TaaMotionBlending, EditorGUIHelper.GetContent("Motion"));

EditorGUI.indentLevel--;

EditorGUILayout.Space();

EditorGUILayout.PropertyField(m\_TaaSharpen);

}

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

using UnityEngine.SceneManagement;

public class AsyncLoad : MonoBehaviour {

// public Image FG;

public Text tishi\_ui;

public Text progressText;

public Transform jiazai;

string[] tishi=new string[5];

private static string NextScene;

public Sprite[] BG\_ImageList;

public Image BG;

public static void LoadingScene(string sceneName)

{

NextScene = sceneName;

SceneManager.LoadScene("AsyncLoad");

}

bool n=true;

// Use this for initialization

void Start()

{

BG.sprite = BG\_ImageList[Random.Range(0, 5)];

tishi[0] = "温馨小提示：浮动螺母用于配合螺丝钉的安装，以便于固定螺钉。";

tishi[1] = "温馨小提示：U与U之间的分界线作为计算设备安装空间的参考点。";

tishi[2] = "温馨小提示：在使用功率超过特定瓦数的用电设备前，必须得到上级主管批准，并在保证线路安全的基础上使用。";

tishi[3] = "温馨小提示：工作人员离开工作区域前，应保证工作区域内保存的重要文件、资料、设备、数据处于安全保护状态。";

tishi[4] = "温馨小提示：在使用功率超过特定瓦数的用电设备前，必须得到上级主管批准，并在保证线路安全的基础上使用。";

tishi\_ui.text = tishi[Random.Range(0,5)];

}

void Update()

{

progressText.text = (int)(currentProgress \* 100) + "%";

jiazai.Rotate(new Vector3(0, 0, 1), -Time.deltaTime \* 300);

if(n)

{

n = false;

StartCoroutine(Load());

}

}

AsyncOperation async;

float currentProgress = 0;

IEnumerator Load()

{

async = SceneManager.LoadSceneAsync(NextScene);

async.allowSceneActivation = false;//不允许场景激活

while (!async.isDone)//加载是否完成

{

if (async.progress >= 0.9F)

{

break;

}

if (currentProgress < async.progress)//加载的进度

{

currentProgress += 0.01F;

}

yield return new WaitForEndOfFrame();

//FG.fillAmount = currentProgress;

}

while (currentProgress < 1F)

{

currentProgress += 0.01F;

yield return new WaitForEndOfFrame();

//FG.fillAmount = currentProgress;

}

async.allowSceneActivation = true;//允许场景激活

async = null;

NextScene = string.Empty;

yield return async;

}

}

using System;

using UnityEngine;

#if UNITY\_EDITOR

#endif

namespace UnityStandardAssets.Cameras

{

[ExecuteInEditMode]

public class AutoCam : PivotBasedCameraRig

{

[SerializeField] private float m\_MoveSpeed = 3; // How fast the rig will move to keep up with target's position

[SerializeField] private float m\_TurnSpeed = 1; // How fast the rig will turn to keep up with target's rotation

[SerializeField] private float m\_RollSpeed = 0.2f;// How fast the rig will roll (around Z axis) to match target's roll.

[SerializeField] private bool m\_FollowVelocity = false;// Whether the rig will rotate in the direction of the target's velocity.

[SerializeField] private bool m\_FollowTilt = true; // Whether the rig will tilt (around X axis) with the target.

[SerializeField] private float m\_SpinTurnLimit = 90;// The threshold beyond which the camera stops following the target's rotation. (used in situations where a car spins out, for example)

[SerializeField] private float m\_TargetVelocityLowerLimit = 4f;// the minimum velocity above which the camera turns towards the object's velocity. Below this we use the object's forward direction.

[SerializeField] private float m\_SmoothTurnTime = 0.2f; // the smoothing for the camera's rotation

private float m\_LastFlatAngle; // The relative angle of the target and the rig from the previous frame.

private float m\_CurrentTurnAmount; // How much to turn the camera

private float m\_TurnSpeedVelocityChange; // The change in the turn speed velocity

private Vector3 m\_RollUp = Vector3.up;// The roll of the camera around the z axis ( generally this will always just be up )

protected override void FollowTarget(float deltaTime)

{

// if no target, or no time passed then we quit early, as there is nothing to do

if (!(deltaTime > 0) || m\_Target == null)

{

return;

}

// initialise some vars, we'll be modifying these in a moment

var targetForward = m\_Target.forward;

var targetUp = m\_Target.up;

if (m\_FollowVelocity && Application.isPlaying)

{

// in follow velocity mode, the camera's rotation is aligned towards the object's velocity direction

// but only if the object is traveling faster than a given threshold.

if (targetRigidbody.velocity.magnitude > m\_TargetVelocityLowerLimit)

{

// velocity is high enough, so we'll use the target's velocty

targetForward = targetRigidbody.velocity.normalized;

targetUp = Vector3.up;

}

else

{

targetUp = Vector3.up;

}

m\_CurrentTurnAmount = Mathf.SmoothDamp(m\_CurrentTurnAmount, 1, ref m\_TurnSpeedVelocityChange, m\_SmoothTurnTime);

}

else

{

// we're in 'follow rotation' mode, where the camera rig's rotation follows the object's rotation.

// This section allows the camera to stop following the target's rotation when the target is spinning too fast.

// eg when a car has been knocked into a spin. The camera will resume following the rotation

// of the target when the target's angular velocity slows below the threshold.

var currentFlatAngle = Mathf.Atan2(targetForward.x, targetForward.z)\*Mathf.Rad2Deg;

if (m\_SpinTurnLimit > 0)

{

var targetSpinSpeed = Mathf.Abs(Mathf.DeltaAngle(m\_LastFlatAngle, currentFlatAngle))/deltaTime;

var desiredTurnAmount = Mathf.InverseLerp(m\_SpinTurnLimit, m\_SpinTurnLimit\*0.75f, targetSpinSpeed);

var turnReactSpeed = (m\_CurrentTurnAmount > desiredTurnAmount ? .1f : 1f);

if (Application.isPlaying)

{

m\_CurrentTurnAmount = Mathf.SmoothDamp(m\_CurrentTurnAmount, desiredTurnAmount,

ref m\_TurnSpeedVelocityChange, turnReactSpeed);

}

else

{

// for editor mode, smoothdamp won't work because it uses deltaTime internally

m\_CurrentTurnAmount = desiredTurnAmount;

}

}

else

{

m\_CurrentTurnAmount = 1;

}

m\_LastFlatAngle = currentFlatAngle;

}

// camera position moves towards target position:

transform.position = Vector3.Lerp(transform.position, m\_Target.position, deltaTime\*m\_MoveSpeed);

// camera's rotation is split into two parts, which can have independend speed settings:

// rotating towards the target's forward direction (which encompasses its 'yaw' and 'pitch')

if (!m\_FollowTilt)

{

targetForward.y = 0;

if (targetForward.sqrMagnitude < float.Epsilon)

{

targetForward = transform.forward;

}

}

var rollRotation = Quaternion.LookRotation(targetForward, m\_RollUp);

// and aligning with the target object's up direction (i.e. its 'roll')

m\_RollUp = m\_RollSpeed > 0 ? Vector3.Slerp(m\_RollUp, targetUp, m\_RollSpeed\*deltaTime) : Vector3.up;

transform.rotation = Quaternion.Lerp(transform.rotation, rollRotation, m\_TurnSpeed\*m\_CurrentTurnAmount\*deltaTime);

}

}

}

using System;

using System.Collections.Generic;

using UnityEngine;

#if UNITY\_EDITOR

using UnityEditor;

#endif

namespace UnityStandardAssets.Utility

{

public class AutoMobileShaderSwitch : MonoBehaviour

{

[SerializeField] private ReplacementList m\_ReplacementList;

// Use this for initialization

private void OnEnable()

{

#if UNITY\_IPHONE || UNITY\_ANDROID || UNITY\_WP8 || UNITY\_TIZEN

var renderers = FindObjectsOfType<Renderer>();

Debug.Log (renderers.Length+" renderers");

var oldMaterials = new List<Material>();

var newMaterials = new List<Material>();

int materialsReplaced = 0;

int materialInstancesReplaced = 0;

foreach(ReplacementDefinition replacementDef in m\_ReplacementList.items)

{

foreach(var r in renderers)

{

Material[] modifiedMaterials = null;

for(int n=0; n<r.sharedMaterials.Length; ++n)

{

var material = r.sharedMaterials[n];

if (material.shader == replacementDef.original)

{

if (modifiedMaterials == null)

{

modifiedMaterials = r.materials;

}

if (!oldMaterials.Contains(material))

{

oldMaterials.Add(material);

Material newMaterial = (Material)Instantiate(material);

newMaterial.shader = replacementDef.replacement;

newMaterials.Add(newMaterial);

++materialsReplaced;

}

Debug.Log ("replacing "+r.gameObject.name+" renderer "+n+" with "+newMaterials[oldMaterials.IndexOf(material)].name);

modifiedMaterials[n] = newMaterials[oldMaterials.IndexOf(material)];

++materialInstancesReplaced;

}

}

if (modifiedMaterials != null)

{

r.materials = modifiedMaterials;

}

}

}

Debug.Log (materialInstancesReplaced+" material instances replaced");

Debug.Log (materialsReplaced+" materials replaced");

for(int n=0; n<oldMaterials.Count; ++n)

{

Debug.Log (oldMaterials[n].name+" ("+oldMaterials[n].shader.name+")"+" replaced with "+newMaterials[n].name+" ("+newMaterials[n].shader.name+")");

}

#endif

}

[Serializable]

public class ReplacementDefinition

{

public Shader original = null;

public Shader replacement = null;

}

[Serializable]

public class ReplacementList

{

public ReplacementDefinition[] items = new ReplacementDefinition[0];

}

}

}

namespace UnityStandardAssets.Utility.Inspector

{

#if UNITY\_EDITOR

[CustomPropertyDrawer(typeof (AutoMobileShaderSwitch.ReplacementList))]

public class ReplacementListDrawer : PropertyDrawer

{

const float k\_LineHeight = 18;

const float k\_Spacing = 4;

public override void OnGUI(Rect position, SerializedProperty property, GUIContent label)

{

EditorGUI.BeginProperty(position, label, property);

float x = position.x;

float y = position.y;

float inspectorWidth = position.width;

// Don't make child fields be indented

var indent = EditorGUI.indentLevel;

EditorGUI.indentLevel = 0;

var items = property.FindPropertyRelative("items");

var titles = new string[] {"Original", "Replacement", ""};

var props = new string[] {"original", "replacement", "-"};

var widths = new float[] {.45f, .45f, .1f};

const float lineHeight = 18;

bool changedLength = false;

if (items.arraySize > 0)

{

for (int i = -1; i < items.arraySize; ++i)

{

var item = items.GetArrayElementAtIndex(i);

float rowX = x;

for (int n = 0; n < props.Length; ++n)

{

float w = widths[n]\*inspectorWidth;

// Calculate rects

Rect rect = new Rect(rowX, y, w, lineHeight);

rowX += w;

if (i == -1)

{

// draw title labels

EditorGUI.LabelField(rect, titles[n]);

}

else

{

if (props[n] == "-" || props[n] == "^" || props[n] == "v")

{

if (GUI.Button(rect, props[n]))

{

switch (props[n])

{

case "-":

items.DeleteArrayElementAtIndex(i);

items.DeleteArrayElementAtIndex(i);

changedLength = true;

break;

case "v":

if (i > 0)

{

items.MoveArrayElement(i, i + 1);

}

break;

case "^":

if (i < items.arraySize - 1)

{

items.MoveArrayElement(i, i - 1);

}

break;

}

}

}

else

{

SerializedProperty prop = item.FindPropertyRelative(props[n]);

EditorGUI.PropertyField(rect, prop, GUIContent.none);

}

}

}

y += lineHeight + k\_Spacing;

if (changedLength)

{

break;

}

}

}

// add button

var addButtonRect = new Rect((x + position.width) - widths[widths.Length - 1]\*inspectorWidth, y,

widths[widths.Length - 1]\*inspectorWidth, lineHeight);

if (GUI.Button(addButtonRect, "+"))

{

items.InsertArrayElementAtIndex(items.arraySize);

}

y += lineHeight + k\_Spacing;

// Set indent back to what it was

EditorGUI.indentLevel = indent;

EditorGUI.EndProperty();

}

public override float GetPropertyHeight(SerializedProperty property, GUIContent label)

{

SerializedProperty items = property.FindPropertyRelative("items");

float lineAndSpace = k\_LineHeight + k\_Spacing;

return 40 + (items.arraySize\*lineAndSpace) + lineAndSpace;

}

}

#endif

}