```
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 automaticall
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_BLACKBERRY
            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
}

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;
            }
        }
        GUILayout.Label("Mapping HDR to LDR ranges since 1982", Edito
    if(mode.intValue>0)
    {
      EditorGUILayout.PropertyField (chromaticAberration, new GUIContent("
Tangential Aberration"));
      EditorGUILayout.PropertyField (axialAberration, new GUIContent(" Axial
Aberration"));
      luminanceDependency.floatValue = EditorGUILayout.Slider(" Contrast
Dependency", luminanceDependency.floatValue, 0.001f, 1.0f);
    }
    else
      EditorGUILayout.PropertyField (chromaticAberration, new GUIContent("
Chromatic Aberration"));

    serObj.ApplyModifiedProperties();
    }using UnityEngine;

[ExecuteInEditMode]
[AddComponentMenu("Image Effects/Vortex")]
public class VortexEffect : ImageEffectBase {
    public Vector2  radius = new Vector2(0.4F,0.4F);
    public float    angle = 50;
    public Vector2  center = new Vector2(0.5F, 0.5F);

    // Called by camera to apply image effect
    void OnRenderImage (RenderTexture source, RenderTexture destination) {
        ImageEffects.RenderDistortion (material, source, destination, angle,
center, radius);
    }
}

}


}
}
using System;
using UnityEngine;

namespace UnityStandardAssets.Cameras
{
    public abstract class AbstractTargetFollower : MonoBehaviour
    {
        public enum UpdateType // The available methods of updating are:
        {
            FixedUpdate, // Update in FixedUpdate (for tracking rigidbodies).
            LateUpdate, // Update in LateUpdate. (for tracking objects that are
moved in Update)
            ManualUpdate, // user must call to update camera
        }

        [SerializeField] protected Transform m_Target;            // The target
object to follow
        [SerializeField] private bool m_AutoTargetPlayer = true;  // Whether the
rig should automatically target the player.
        [SerializeField] private UpdateType m_UpdateType;         // stores the
selected update type

        protected Rigidbody targetRigidbody;


        protected virtual void Start()
        {
            // if auto targeting is used, find the object tagged "Player"
            // any class inheriting from this should call base.Start() to perform
this action!
            if (m_AutoTargetPlayer)
            {
                FindAndTargetPlayer();
            }
            if (m_Target == null) return;
            targetRigidbody = m_Target.GetComponent<Rigidbody>();
        }


        private void FixedUpdate()
        {
            // we update from here if updatetype is set to Fixed, or in auto mode,
            // if the target has a rigidbody, and isn't kinematic.
            if      (m_AutoTargetPlayer      &&      (m_Target      ==      null
|| !m_Target.gameObject.activeSelf))
            {
                FindAndTargetPlayer();
            }
            if (m_UpdateType == UpdateType.FixedUpdate)
            {
                FollowTarget(Time.deltaTime);
            }
        }
        private void LateUpdate()
        {
            // we update from here if updatetype is set to Late, or in auto mode,
            // if the target does not have a rigidbody, or - does have a rigidbody
but is set to kinematic.
            if (m_AutoTargetPlayer && (m_Target == null
|| !m_Target.gameObject.activeSelf))
            {
                FindAndTargetPlayer();
            }
            if (m_UpdateType == UpdateType.LateUpdate)
            {
                FollowTarget(Time.deltaTime);
            }
        }


        public void ManualUpdate()
        {
            // we update from here if updatetype is set to Late, or in auto mode,
            // if the target does not have a rigidbody, or - does have a rigidbody
but is set to kinematic.
            if (m_AutoTargetPlayer && (m_Target == null
|| !m_Target.gameObject.activeSelf))
            {
                FindAndTargetPlayer();
            }
            if (m_UpdateType == UpdateType.ManualUpdate)
            {
                FollowTarget(Time.deltaTime);
            }
        }

        protected abstract void FollowTarget(float deltaTime);


        public void FindAndTargetPlayer()
        {
            // auto target an object tagged player, if no target has been assigned
            var targetObj = GameObject.FindGameObjectWithTag("Player");
            if (targetObj)
            {
                SetTarget(targetObj.transform);
            }
        }


        public virtual void SetTarget(Transform newTransform)
        {
            m_Target = newTransform;
        }


        public Transform Target
        {
            get { return m_Target; }
        }
    }
}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();
        }
    }
}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
}
```