```
#pragma strict

static var meshes : Mesh[];
static var currentTris : int = 0;

static function HasMeshes () : boolean {
    if (!meshes)
        return false;
    for (var m : Mesh in meshes)
        if (null == m)
            return false;

    return true;
}

static function Cleanup () {
    if (!meshes)
        return;

    for (var m : Mesh in meshes) {
        if (null != m) {
            DestroyImmediate (m);
            m = null;
        }
    }
    meshes = null;
}

static function GetMeshes (totalWidth : int, totalHeight : int) : Mesh[]
{
    if (HasMeshes () && (currentTris == (totalWidth * totalHeight))) {
        return meshes;
    }

    var maxTris : int = 65000 / 3;
    var totalTris : int = totalWidth * totalHeight;
    currentTris = totalTris;

    var meshCount : int = Mathf.CeilToInt ((1.0f * totalTris) / (1.0f * maxTris));
```

```
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;
            }
        }

        bool downsampling
        {
            get { return settings.downsampling; }
        }

        bool ambientOnly
        {
            get { return settings.ambientOnly && isAmbientOnlySupported; }
        }

        // AO shader
        Shader aoShader
        {
            get
            {
                if (_aoShader == null)
                    _aoShader = Shader.Find("Hidden/Image Effects/Cinematic/AmbientOcclusion");
                return _aoShader;
            }
        }

        [SerializeField] Shader _aoShader;

        // Temporary aterial for the AO shader
        Material aoMaterial
        {
            get
            {
                if (_aoMaterial == null)
                    _aoMaterial = ImageEffectHelper.CheckShaderAndCreateMaterial(aoShader);
                return _aoMaterial;
            }
        }

        Material _aoMaterial;

        // Command buffer for the AO pass
        CommandBuffer aoCommands
        {
            get
            {
                if (_aoCommands == null)
                {
                    _aoCommands = new CommandBuffer();
                    _aoCommands.name = "AmbientOcclusion";
                }
                return _aoCommands;
            }
        }

        CommandBuffer _aoCommands;

        // Target camera
        Camera targetCamera
        {
```

```
        get { return GetComponent<Camera>(); }
    }

    // Property observer
    PropertyObserver propertyObserver { get; set; }

    // Reference to the quad mesh in the built-in assets
    // (used in MRT blitting)
    Mesh quadMesh
    {
        get { return _quadMesh; }
    }

    [SerializeField] Mesh _quadMesh;

    #endregion

    #region Effect Passes

    // Build commands for the AO pass (used in the ambient-only mode).
    void BuildAOCommands()
    {
        var cb = aoCommands;

        var tw = targetCamera.pixelWidth;
        var th = targetCamera.pixelHeight;
        var ts = downsampling ? 2 : 1;
        var format = RenderTextureFormat.R8;
        var rwMode = RenderTextureReadWrite.Linear;
        var filter = FilterMode.Bilinear;

        // AO buffer
        var m = aoMaterial;
        var rtMask = Shader.PropertyToID("_OcclusionTexture");
        cb.GetTemporaryRT(rtMask, tw / ts, th / ts, 0, filter, format, rwMode);

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

        // Blur buffer
        var rtBlur = Shader.PropertyToID("_OcclusionBlurTexture");

        // Primary blur filter (large kernel)
        cb.GetTemporaryRT(rtBlur, tw, th, 0, filter, format, rwMode);
        cb.SetGlobalVector("_BlurVector", Vector2.right * 2);
        cb.Blit(rtMask, rtBlur, m, 1);
        cb.ReleaseTemporaryRT(rtMask);

        cb.GetTemporaryRT(rtMask, tw, th, 0, filter, format, rwMode);
        cb.SetGlobalVector("_BlurVector", Vector2.up * 2 * ts);
        cb.Blit(rtBlur, rtMask, m, 1);
        cb.ReleaseTemporaryRT(rtBlur);

        // Secondary blur filter (small kernel)
        cb.GetTemporaryRT(rtBlur, tw, th, 0, filter, format, rwMode);
        cb.SetGlobalVector("_BlurVector", Vector2.right * ts);
        cb.Blit(rtMask, rtBlur, m, 2);
        cb.ReleaseTemporaryRT(rtMask);

        cb.GetTemporaryRT(rtMask, tw, th, 0, filter, format, rwMode);
        cb.SetGlobalVector("_BlurVector", Vector2.up * ts);
        cb.Blit(rtBlur, rtMask, m, 2);
        cb.ReleaseTemporaryRT(rtBlur);

        // Combine AO to the G-buffer.
        var mrt = new RenderTargetIdentifier[] {
            BuiltinRenderTextureType.GBuffer0,       // Albedo, Occ
            BuiltinRenderTextureType.CameraTarget   // Ambient
        };
        cb.SetRenderTarget(mrt, BuiltinRenderTextureType.CameraTarget);
        cb.SetGlobalTexture("_OcclusionTexture", rtMask);
        cb.DrawMesh(quadMesh, Matrix4x4.identity, m, 0, 4);

        cb.ReleaseTemporaryRT(rtMask);
    }

    // Execute the AO pass immediately (used in the forward mode).
    void ExecuteAOPass(RenderTexture source, RenderTexture destination)
    {
        var tw = source.width;
        var th = source.height;
        var ts = downsampling ? 2 : 1;
        var format = RenderTextureFormat.R8;
        var rwMode = RenderTextureReadWrite.Linear;

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

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

            // Primary blur filter (large kernel)
            var rtBlur = RenderTexture.GetTemporary(tw, th, 0, format, rwMode);
            m.SetVector("_BlurVector", Vector2.right * 2);
            Graphics.Blit(rtMask, rtBlur, m, 1);
            RenderTexture.ReleaseTemporary(rtMask);

            rtMask = RenderTexture.GetTemporary(tw, th, 0, format, rwMode);
            m.SetVector("_BlurVector", Vector2.up * 2 * ts);
            Graphics.Blit(rtBlur, rtMask, m, 1);
            RenderTexture.ReleaseTemporary(rtBlur);

            // Secondary blur filter (small kernel)
            rtBlur = RenderTexture.GetTemporary(tw, th, 0, format, rwMode);
            m.SetVector("_BlurVector", Vector2.right * ts);
            Graphics.Blit(rtMask, rtBlur, m, 2);
            RenderTexture.ReleaseTemporary(rtMask);

            rtMask = RenderTexture.GetTemporary(tw, th, 0, format, rwMode);
            m.SetVector("_BlurVector", Vector2.up * ts);
            Graphics.Blit(rtBlur, rtMask, m, 2);
            RenderTexture.ReleaseTemporary(rtBlur);

            // Combine AO with the source.
            m.SetTexture("_OcclusionTexture", rtMask);

            if (!settings.debug)
                Graphics.Blit(source, destination, m, 3);
            else
                Graphics.Blit(source, destination, m, 5);

            RenderTexture.ReleaseTemporary(rtMask);
        }

        // Update the common material properties.
        void UpdateMaterialProperties()
        {
            var m = aoMaterial;
            m.shaderKeywords = null;
            m.SetFloat("_Intensity", intensity);
            m.SetFloat("_Radius", radius);
            m.SetFloat("_TargetScale", downsampling ? 0.5f : 1);

            // Occlusion source
            if (occlusionSource == OcclusionSource.GBuffer)
                m.EnableKeyword("_SOURCE_GBUFFER");
            else if (occlusionSource == OcclusionSource.DepthTexture)
                m.EnableKeyword("_SOURCE_DEPTH");
            else
                m.EnableKeyword("_SOURCE_DEPTHNORMALS");

            // Sample count
            if (sampleCount == SampleCount.Lowest)
                m.EnableKeyword("_SAMPLECOUNT_LOWEST");
            else
                m.SetInt("_SampleCount", sampleCountValue);
        }

        #endregion

        #region MonoBehaviour Functions

        void OnEnable()
        {
            // Check if the shader is supported in the current platform.
            if (!ImageEffectHelper.IsSupported(aoShader, true, false, this))
            {
                enabled = false;
                return;
            }

            // Register the command buffer if in the ambient-only mode.
            if (ambientOnly)
                targetCamera.AddCommandBuffer(CameraEvent.BeforeReflections,
aoCommands);

            // Enable depth textures which the occlusion source requires.
            if (occlusionSource == OcclusionSource.DepthTexture)
                targetCamera.depthTextureMode |= DepthTextureMode.Depth;

            if (occlusionSource != OcclusionSource.GBuffer)
                targetCamera.depthTextureMode |= DepthTextureMode.DepthNormals;
        }
        void OnDisable()
        {
            // Destroy all the temporary resources.
            if (_aoMaterial != null) DestroyImmediate(_aoMaterial);
            _aoMaterial = null;

            if (_aoCommands != null)
                targetCamera.RemoveCommandBuffer(CameraEvent.BeforeReflections,
_aoCommands);
            _aoCommands = null;
        }

        void Update()
        {
            if (propertyObserver.CheckNeedsReset(settings, targetCamera))
            {
                // Reinitialize all the resources by disabling/enabling itself.
                // This is not very efficient way but just works...
                OnDisable();
                OnEnable();

                // Build the command buffer if in the ambient-only mode.
                if (ambientOnly)
                {
                    aoCommands.Clear();
                    BuildAOCommands();
                }

                propertyObserver.Update(settings, targetCamera);
            }

            // Update the material properties (later used in the AO commands).
            if (ambientOnly) UpdateMaterialProperties();
        }

        [ImageEffectOpaque]
        void OnRenderImage(RenderTexture source, RenderTexture destination)
        {
            if (ambientOnly)
            {
                // Do nothing in the ambient-only mode.
                Graphics.Blit(source, destination);
            }
            else
            {
                // Execute the AO pass.
                UpdateMaterialProperties();
                ExecuteAOPass(source, destination);
            }
        }

        #endregion
    }
}
using System;
using UnityEngine;
using Object = UnityEngine.Object;

namespace UnityStandardAssets.Utility
{
    public class ActivateTrigger : MonoBehaviour
    {
        // A multi-purpose script which causes an action to occur when
        // a trigger collider is entered.
        public enum Mode
        {
            Trigger = 0,    // Just broadcast the action on to the target
            Replace = 1,    // replace target with source
            Activate = 2,   // Activate the target GameObject
            Enable = 3,     // Enable a component
            Animate = 4,    // Start animation on target
            Deactivate = 5  // Decativate target GameObject
        }

        public Mode action = Mode.Activate;         // The action to accomplish
        public Object target;                       // The game object to affect.
If none, the trigger work on this game object
        public GameObject source;
        public int triggerCount = 1;
        public bool repeatTrigger = false;


        private void DoActivateTrigger()
        {
            triggerCount--;

            if (triggerCount == 0 || repeatTrigger)
            {
                Object currentTarget = target ?? gameObject;
                Behaviour targetBehaviour = currentTarget as Behaviour;
                GameObject targetGameObject = currentTarget as GameObject;
                if (targetBehaviour != null)
                {
                    targetGameObject = targetBehaviour.gameObject;
                }

                switch (action)
                {
                    case Mode.Trigger:
                        if (targetGameObject != null)
                        {

targetGameObject.BroadcastMessage("DoActivateTrigger");
                        }
                        break;
                    case Mode.Replace:
                        if (source != null)
                        {
                            if (targetGameObject != null)
                            {
                                Instantiate(source,
targetGameObject.transform.position,

targetGameObject.transform.rotation);
                                DestroyObject(targetGameObject);
                            }
                        }
                        break;
                    case Mode.Activate:
                        if (targetGameObject != null)
                        {
                            targetGameObject.SetActive(true);
                        }
                        break;
                    case Mode.Enable:
                        if (targetBehaviour != null)
                        {
                            targetBehaviour.enabled = true;
                        }
                        break;
                    case Mode.Animate:
                        if (targetGameObject != null)
                        {
                            targetGameObject.GetComponent<Animation>().Play();
                        }
                        break;
                    case Mode.Deactivate:
                        if (targetGameObject != null)
                        {
                            targetGameObject.SetActive(false);
                        }
                        break;
                }
            }
        }


        private void OnTriggerEnter(Collider other)
        {
            DoActivateTrigger();
        }
    }
}

        // put together:

        if (v.z >= 0.0)
            sunShaftsMaterial.SetVector  (″_SunColor″,  Vector4  (sunColor.r,
sunColor.g, sunColor.b, sunColor.a) * sunShaftIntensity);
        else
            sunShaftsMaterial.SetVector  (″_SunColor″,  Vector4.zero);  //  no
backprojection !
        sunShaftsMaterial.SetTexture (″_ColorBuffer″, lrDepthBuffer);
        Graphics.Blit (source, destination, sunShaftsMaterial, (screenBlendMode
== ShaftsScreenBlendMode.Screen) ? 0 : 4);

        RenderTexture.ReleaseTemporary (lrDepthBuffer);
        RenderTexture.ReleaseTemporary (secondQuarterRezColor);
    }

    // helper functions

    private function ClampBlurIterationsToSomethingThatMakesSense (its : int) :
int {
        if (its < 1)
```

```
            return 1;
        else if (its > 4)
            return 4;
        else
            return its;
    }

}
#pragma strict

@CustomEditor (SunShafts)

class SunShaftsEditor extends Editor
{
    var serObj : SerializedObject;

    var sunTransform : SerializedProperty;
    var radialBlurIterations : SerializedProperty;
    var sunColor : SerializedProperty;
    var sunShaftBlurRadius : SerializedProperty;
    var sunShaftIntensity : SerializedProperty;
    var useSkyBoxAlpha : SerializedProperty;
    var useDepthTexture : SerializedProperty;
    var resolution : SerializedProperty;
    var screenBlendMode : SerializedProperty;
    var maxRadius : SerializedProperty;

    function OnEnable () {
        serObj = new SerializedObject (target);

        screenBlendMode = serObj.FindProperty("screenBlendMode");

        sunTransform = serObj.FindProperty("sunTransform");
        sunColor = serObj.FindProperty("sunColor");

        sunShaftBlurRadius = serObj.FindProperty("sunShaftBlurRadius");
        radialBlurIterations = serObj.FindProperty("radialBlurIterations");

        sunShaftIntensity = serObj.FindProperty("sunShaftIntensity");
        useSkyBoxAlpha = serObj.FindProperty("useSkyBoxAlpha");

        resolution =  serObj.FindProperty("resolution");

        maxRadius = serObj.FindProperty("maxRadius");
        useDepthTexture = serObj.FindProperty("useDepthTexture");
    }

    function OnInspectorGUI () {
        serObj.Update ();

        EditorGUILayout.BeginHorizontal();

        var oldVal : boolean = useDepthTexture.boolValue;
        EditorGUILayout.PropertyField (useDepthTexture, new GUIContent ("Rely on
Z Buffer?"));
        if((target as SunShafts).camera)
            GUILayout.Label("Current camera mode: "+ (target as
SunShafts).camera.depthTextureMode, EditorStyles.miniBoldLabel);

        EditorGUILayout.EndHorizontal();

        // depth buffer need
        /*
        var newVal : boolean = useDepthTexture.boolValue;
        if (newVal != oldVal) {
            if(newVal)
                (target as SunShafts).camera.depthTextureMode |=
DepthTextureMode.Depth;
            else
                (target as SunShafts).camera.depthTextureMode &=
~DepthTextureMode.Depth;
        }
        */

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

        EditorGUILayout.Separator ();

        EditorGUILayout.BeginHorizontal();

        EditorGUILayout.PropertyField (sunTransform, new GUIContent("Shafts
caster", "Chose a transform that acts as a root point for the produced sun shafts"));
        if((target as SunShafts).sunTransform && (target as SunShafts).camera) {
            if (GUILayout.Button("Center on " + (target as SunShafts).camera.name))
{
                if (EditorUtility.DisplayDialog ("Move sun shafts source?", "The
SunShafts caster named "+ (target as SunShafts).sunTransform.name +"\n will be
centered along "+(target as SunShafts).camera.name+". Are you sure? ", "Please do",
"Don’t")) {
                    var ray : Ray = (target as
SunShafts).camera.ViewportPointToRay(Vector3(0.5,0.5,0));
                    (target as SunShafts).sunTransform.position = ray.origin +
ray.direction * 500.0;
                    (target as SunShafts).sunTransform.LookAt ((target as
SunShafts).transform);
                }
            }
        }

        EditorGUILayout.EndHorizontal();

        EditorGUILayout.Separator ();

        EditorGUILayout.PropertyField (sunColor, new GUIContent ("Shafts
color"));
        maxRadius.floatValue = 1.0f - EditorGUILayout.Slider ("Distance falloff",
1.0f - maxRadius.floatValue, 0.1, 1.0);

        EditorGUILayout.Separator ();

        sunShaftBlurRadius.floatValue = EditorGUILayout.Slider ("Blur size",
sunShaftBlurRadius.floatValue, 1.0, 10.0);
        radialBlurIterations.intValue = EditorGUILayout.IntSlider ("Blur
iterations", radialBlurIterations.intValue, 1, 3);

        EditorGUILayout.Separator ();

        EditorGUILayout.PropertyField (sunShaftIntensity, new
GUIContent("Intensity"));
        useSkyBoxAlpha.floatValue = EditorGUILayout.Slider ("Use alpha mask",
useSkyBoxAlpha.floatValue, 0.0, 1.0);

        serObj.ApplyModifiedProperties();
    }
}
#pragma strict

@script ExecuteInEditMode
```

```
@script RequireComponent (Camera)
@script AddComponentMenu ("Image Effects/Tilt shift")

class TiltShift extends PostEffectsBase {
    public var tiltShiftShader : Shader;
    private var tiltShiftMaterial : Material = null;

    public var renderTextureDivider : int = 2;
    public var blurIterations : int = 2;
    public var enableForegroundBlur : boolean = true;
    public var foregroundBlurIterations : int = 2;
    public var maxBlurSpread : float = 1.5f;

    public var focalPoint : float = 30.0f;
    public var smoothness : float = 1.65f;

    public var visualizeCoc : boolean = false;

    // these values will be automatically determined

    private var start01 : float = 0.0f;
    private var distance01 : float = 0.2f;
    private var end01 : float = 1.0f;
    private var curve : float = 1.0f;

    function CheckResources () : boolean {
        CheckSupport (true);

        tiltShiftMaterial = CheckShaderAndCreateMaterial (tiltShiftShader,
tiltShiftMaterial);

        if(!isSupported)
            ReportAutoDisable ();
        return isSupported;
    }

    function OnRenderImage (source : RenderTexture, destination : RenderTexture)
{
        if(CheckResources()==false) {
            Graphics.Blit (source, destination);
            return;
        }

        var widthOverHeight : float = (1.0f * source.width) / (1.0f *
source.height);
        var oneOverBaseSize : float = 1.0f / 512.0f;

        // clamp some values

        renderTextureDivider = renderTextureDivider < 1 ? 1 : renderTextureDivider;
        renderTextureDivider = renderTextureDivider > 4 ? 4 : renderTextureDivider;
        blurIterations = blurIterations < 1 ? 0 : blurIterations;
        blurIterations = blurIterations > 4 ? 4 : blurIterations;

        // automagically calculate parameters based on focalPoint

        var focalPoint01 : float = GetComponent.<Camera>().WorldToViewportPoint
(focalPoint        *        GetComponent.<Camera>().transform.forward        +
GetComponent.<Camera>().transform.position).z                              /
(GetComponent.<Camera>().farClipPlane);

        distance01 = focalPoint01;
        start01 = 0.0;
        end01 = 1.0;
        start01 = Mathf.Min (focalPoint01 - Mathf.Epsilon, start01);
        end01 = Mathf.Max (focalPoint01 + Mathf.Epsilon, end01);
        curve = smoothness * distance01;

        // resources

        var cocTex : RenderTexture = RenderTexture.GetTemporary (source.width,
source.height, 0);
        var cocTex2 : RenderTexture = RenderTexture.GetTemporary (source.width,
source.height, 0);
        var lrTex1 : RenderTexture = RenderTexture.GetTemporary (source.width /
renderTextureDivider, source.height / renderTextureDivider, 0);
        var lrTex2 : RenderTexture = RenderTexture.GetTemporary (source.width /
renderTextureDivider, source.height / renderTextureDivider, 0);

        // coc

        tiltShiftMaterial.SetVector  (″_SimpleDofParams″,  Vector4  (start01,
distance01, end01, curve));
        tiltShiftMaterial.SetTexture (″_Coc″, cocTex);

        if (enableForegroundBlur) {
            Graphics.Blit (source, cocTex, tiltShiftMaterial, 0);
            Graphics.Blit (cocTex, lrTex1); // downwards (only really needed if
lrTex resolution is different)

            for (var fgBlurIter : int = 0; fgBlurIter < foregroundBlurIterations;
fgBlurIter++ ) {
                tiltShiftMaterial.SetVector    ("offsets",    Vector4    (0.0,
(maxBlurSpread * 0.75f) * oneOverBaseSize, 0.0, 0.0));
                Graphics.Blit (lrTex1, lrTex2, tiltShiftMaterial, 3);
                tiltShiftMaterial.SetVector ("offsets", Vector4 ((maxBlurSpread *
0.75f / widthOverHeight) * oneOverBaseSize, 0.0, 0.0, 0.0));
                Graphics.Blit (lrTex2, lrTex1, tiltShiftMaterial, 3);
            }

            Graphics.Blit (lrTex1, cocTex2, tiltShiftMaterial, 7);   // upwards
(only really needed if lrTex resolution is different)
            tiltShiftMaterial.SetTexture ("_Coc", cocTex2);
        } else {
            RenderTexture.active = cocTex;
            GL.Clear (false, true, Color.black);
        }

        // combine coc's
        Graphics.Blit (source, cocTex, tiltShiftMaterial, 5);
        tiltShiftMaterial.SetTexture ("_Coc", cocTex);

        // downsample & blur

        Graphics.Blit (source, lrTex2);

        for (var iter : int = 0; iter < blurIterations; iter++ ) {
            tiltShiftMaterial.SetVector ("offsets", Vector4 (0.0, (maxBlurSpread
* 1.0f) * oneOverBaseSize, 0.0, 0.0));
            Graphics.Blit (lrTex2, lrTex1, tiltShiftMaterial, 6);
            tiltShiftMaterial.SetVector  ("offsets",  Vector4  ((maxBlurSpread  *
1.0f / widthOverHeight) * oneOverBaseSize, 0.0, 0.0, 0.0));
            Graphics.Blit (lrTex1, lrTex2, tiltShiftMaterial, 6);
        }

        tiltShiftMaterial.SetTexture ("_Blurred", lrTex2);

        Graphics.Blit (source, destination, tiltShiftMaterial, visualizeCoc ? 4 :
1);

        RenderTexture.ReleaseTemporary (cocTex);
        RenderTexture.ReleaseTemporary (cocTex2);
        RenderTexture.ReleaseTemporary (lrTex1);
        RenderTexture.ReleaseTemporary (lrTex2);
    }
}

#pragma strict

@CustomEditor (TiltShift)
class TiltShiftEditor extends Editor
{
    var serObj : SerializedObject;

    var focalPoint : SerializedProperty;
    var smoothness : SerializedProperty;
    var visualizeCoc : SerializedProperty;

    var renderTextureDivider : SerializedProperty;
    var blurIterations : SerializedProperty;
    var foregroundBlurIterations : SerializedProperty;
    var maxBlurSpread : SerializedProperty;
    var enableForegroundBlur : SerializedProperty;

    function OnEnable () {
        serObj = new SerializedObject (target);

        focalPoint = serObj.FindProperty ("focalPoint");
        smoothness = serObj.FindProperty ("smoothness");
        visualizeCoc = serObj.FindProperty ("visualizeCoc");

        renderTextureDivider = serObj.FindProperty ("renderTextureDivider");
        blurIterations = serObj.FindProperty ("blurIterations");
        foregroundBlurIterations              =             serObj.FindProperty
("foregroundBlurIterations");
        maxBlurSpread = serObj.FindProperty ("maxBlurSpread");
        enableForegroundBlur  =  serObj.FindProperty  ("enableForegroundBlur");

    }

    function OnInspectorGUI () {
        serObj.Update ();

        var go : GameObject = (target as TiltShift).gameObject;

        if (!go)
```

```
            return;

        if (!go.camera)
            return;

        GUILayout.Label ("Current: "+go.camera.name+", near
"+go.camera.nearClipPlane+", far: "+go.camera.farClipPlane+", focal:
"+focalPoint.floatValue, EditorStyles.miniBoldLabel);

        GUILayout.Label ("Focal Settings", EditorStyles.boldLabel);
        EditorGUILayout.PropertyField (visualizeCoc, new
GUIContent("Visualize"));
        focalPoint.floatValue = EditorGUILayout.Slider ("Distance",
focalPoint.floatValue, go.camera.nearClipPlane, go.camera.farClipPlane);
        EditorGUILayout.PropertyField (smoothness, new
GUIContent("Smoothness"));

        EditorGUILayout.Separator ();

        GUILayout.Label ("Background Blur", EditorStyles.boldLabel);
        renderTextureDivider.intValue = EditorGUILayout.Slider ("Downsample",
renderTextureDivider.intValue, 1, 3);
        blurIterations.intValue = EditorGUILayout.Slider ("Iterations",
blurIterations.intValue, 1, 4);
        EditorGUILayout.PropertyField (maxBlurSpread, new GUIContent("Max blur
spread"));

        EditorGUILayout.Separator ();

        GUILayout.Label ("Foreground Blur", EditorStyles.boldLabel);
        EditorGUILayout.PropertyField (enableForegroundBlur, new
GUIContent("Enable"));

        if (enableForegroundBlur.boolValue)
            foregroundBlurIterations.intValue = EditorGUILayout.Slider
("Iterations", foregroundBlurIterations.intValue, 1, 4);

        //GUILayout.Label ("Background options");
        //edgesOnly.floatValue = EditorGUILayout.Slider ("Edges only",
edgesOnly.floatValue, 0.0, 1.0);
        //EditorGUILayout.PropertyField (edgesOnlyBgColor, new GUIContent
("Background"));

        serObj.ApplyModifiedProperties();
```

```
    }
}
#pragma strict

@script ExecuteInEditMode
@script RequireComponent (Camera)
@script AddComponentMenu ("Image Effects/Tonemapping")

class Tonemapping extends PostEffectsBase {

    public enum TonemapperType {
        SimpleReinhard,
        UserCurve,
        Hable,
        Photographic,
        OptimizedHejiDawson,
        AdaptiveReinhard,
        AdaptiveReinhardAutoWhite,
    };

    public enum AdaptiveTexSize {
        Square16 = 16,
        Square32 = 32,
        Square64 = 64,
        Square128 = 128,
        Square256 = 256,
        Square512 = 512,
        Square1024 = 1024,
    };

    public var type : TonemapperType = TonemapperType.Photographic;
    public var adaptiveTextureSize = AdaptiveTexSize.Square256;

    // CURVE parameter
    public var remapCurve : AnimationCurve;
    private var curveTex : Texture2D = null;

    // UNCHARTED parameter
    public var exposureAdjustment : float = 1.5f;

    // REINHARD parameter
    public var middleGrey : float = 0.4f;
    public var white : float = 2.0f;
    public var adaptionSpeed : float = 1.5f;
    // usual & internal stuff
    public var tonemapper : Shader = null;
    public var validRenderTextureFormat : boolean = true;
    private var tonemapMaterial : Material = null;
    private var rt : RenderTexture = null;
    private var rtFormat : RenderTextureFormat =  RenderTextureFormat.ARGBHalf;

    function CheckResources () : boolean {
        CheckSupport (false, true);

        tonemapMaterial        =        CheckShaderAndCreateMaterial(tonemapper,
tonemapMaterial);
        if (!curveTex && type == TonemapperType.UserCurve) {
            curveTex = new Texture2D (256, 1, TextureFormat.ARGB32, false, true);

            curveTex.filterMode = FilterMode.Bilinear;
            curveTex.wrapMode = TextureWrapMode.Clamp;
            curveTex.hideFlags = HideFlags.DontSave;
        }

        if(!isSupported)
            ReportAutoDisable ();
        return isSupported;
    }

    public function UpdateCurve () : float {
        var range : float = 1.0f;
        if(remapCurve.keys.length  < 1)
            remapCurve =  new AnimationCurve(Keyframe(0, 0), Keyframe(2, 1));

        if (remapCurve) {
            if(remapCurve.length)
                range = remapCurve[remapCurve.length-1].time;
            for (var i : float = 0.0f; i <= 1.0f; i += 1.0f / 255.0f) {
                var c : float = remapCurve.Evaluate(i * 1.0f * range);
                curveTex.SetPixel (Mathf.Floor(i*255.0f), 0, Color(c,c,c));
            }
            curveTex.Apply ();
        }
        return 1.0f / range;
    }

    function OnDisable () {
        if (rt) {
            DestroyImmediate (rt);
            rt = null;
        }
        if (tonemapMaterial) {
            DestroyImmediate (tonemapMaterial);
            tonemapMaterial = null;
        }
        if (curveTex) {
            DestroyImmediate (curveTex);
            curveTex = null;
        }
    }

    function CreateInternalRenderTexture () : boolean {
        if (rt) {
            return false;
        }
        rtFormat = SystemInfo.SupportsRenderTextureFormat
(RenderTextureFormat.RGHalf) ? RenderTextureFormat.RGHalf :
RenderTextureFormat.ARGBHalf;
        rt = new RenderTexture(1,1, 0, rtFormat);
        rt.hideFlags = HideFlags.DontSave;
        return true;
    }

    // a new attribute we introduced in 3.5 indicating that the image filter chain
will continue in LDR
    @ImageEffectTransformsToLDR
    function OnRenderImage (source : RenderTexture, destination : RenderTexture)
{
        if (CheckResources() == false) {
            Graphics.Blit (source, destination);
            return;
        }

        #if UNITY_EDITOR
        validRenderTextureFormat = true;
        if (source.format != RenderTextureFormat.ARGBHalf) {
            validRenderTextureFormat = false;
        }
        #endif

        // clamp some values to not go out of a valid range
        exposureAdjustment = exposureAdjustment < 0.001f ? 0.001f :
exposureAdjustment;

        // SimpleReinhard tonemappers (local, non adaptive)

        if (type == TonemapperType.UserCurve) {
            var rangeScale : float = UpdateCurve ();
            tonemapMaterial.SetFloat(″_RangeScale″, rangeScale);
            tonemapMaterial.SetTexture(″_Curve″, curveTex);
            Graphics.Blit(source, destination, tonemapMaterial, 4);
            return;
        }

        if (type == TonemapperType.SimpleReinhard) {
            tonemapMaterial.SetFloat(″_ExposureAdjustment″, exposureAdjustment);

            Graphics.Blit(source, destination, tonemapMaterial, 6);
            return;
        }

        if (type == TonemapperType.Hable) {
            tonemapMaterial.SetFloat(″_ExposureAdjustment″,
exposureAdjustment);
            Graphics.Blit(source, destination, tonemapMaterial, 5);
            return;
        }

        if (type == TonemapperType.Photographic) {
            tonemapMaterial.SetFloat(″_ExposureAdjustment″,
exposureAdjustment);
            Graphics.Blit(source, destination, tonemapMaterial, 8);
            return;
        }

        if (type == TonemapperType.OptimizedHejiDawson) {
            tonemapMaterial.SetFloat(″_ExposureAdjustment″, 0.5f *
exposureAdjustment);
            Graphics.Blit(source, destination, tonemapMaterial, 7);
            return;
        }

        // still here?
        // =>  adaptive tone mapping:
```

```
        // builds an average log luminance, tonemaps according to
        // middle grey and white values (user controlled)

        // AdaptiveReinhardAutoWhite will calculate white value automagically

        var freshlyBrewedInternalRt : boolean = CreateInternalRenderTexture (); //
this retrieves rtFormat, so should happen before rt allocations

        var rtSquared : RenderTexture =
RenderTexture.GetTemporary(adaptiveTextureSize, adaptiveTextureSize, 0,
rtFormat);
        Graphics.Blit(source, rtSquared);

        var downsample : int = Mathf.Log(rtSquared.width * 1.0f, 2);

        var div : int = 2;
        var rts : RenderTexture[] = new RenderTexture[downsample];
        for (var i : int = 0; i < downsample; i++) {
            rts[i] = RenderTexture.GetTemporary(rtSquared.width / div,
rtSquared.width / div, 0, rtFormat);
            div *= 2;
        }

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

        // downsample pyramid

        var lumRt = rts[downsample-1];
        Graphics.Blit(rtSquared, rts[0], tonemapMaterial, 1);
        if (type == TonemapperType.AdaptiveReinhardAutoWhite) {
            for(i = 0; i < downsample-1; i++) {
                Graphics.Blit(rts[i], rts[i+1], tonemapMaterial, 9);
                lumRt = rts[i+1];
            }
        }
        else if (type == TonemapperType.AdaptiveReinhard) {
            for(i = 0; i < downsample-1; i++) {
                Graphics.Blit(rts[i], rts[i+1]);
                lumRt = rts[i+1];
            }
        }

        // we have the needed values, let's apply adaptive tonemapping
        adaptionSpeed = adaptionSpeed < 0.001f ? 0.001f : adaptionSpeed;

                        {
                            // 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] == ″ˆ″ || 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 ″ˆ″:
                                            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
}
```