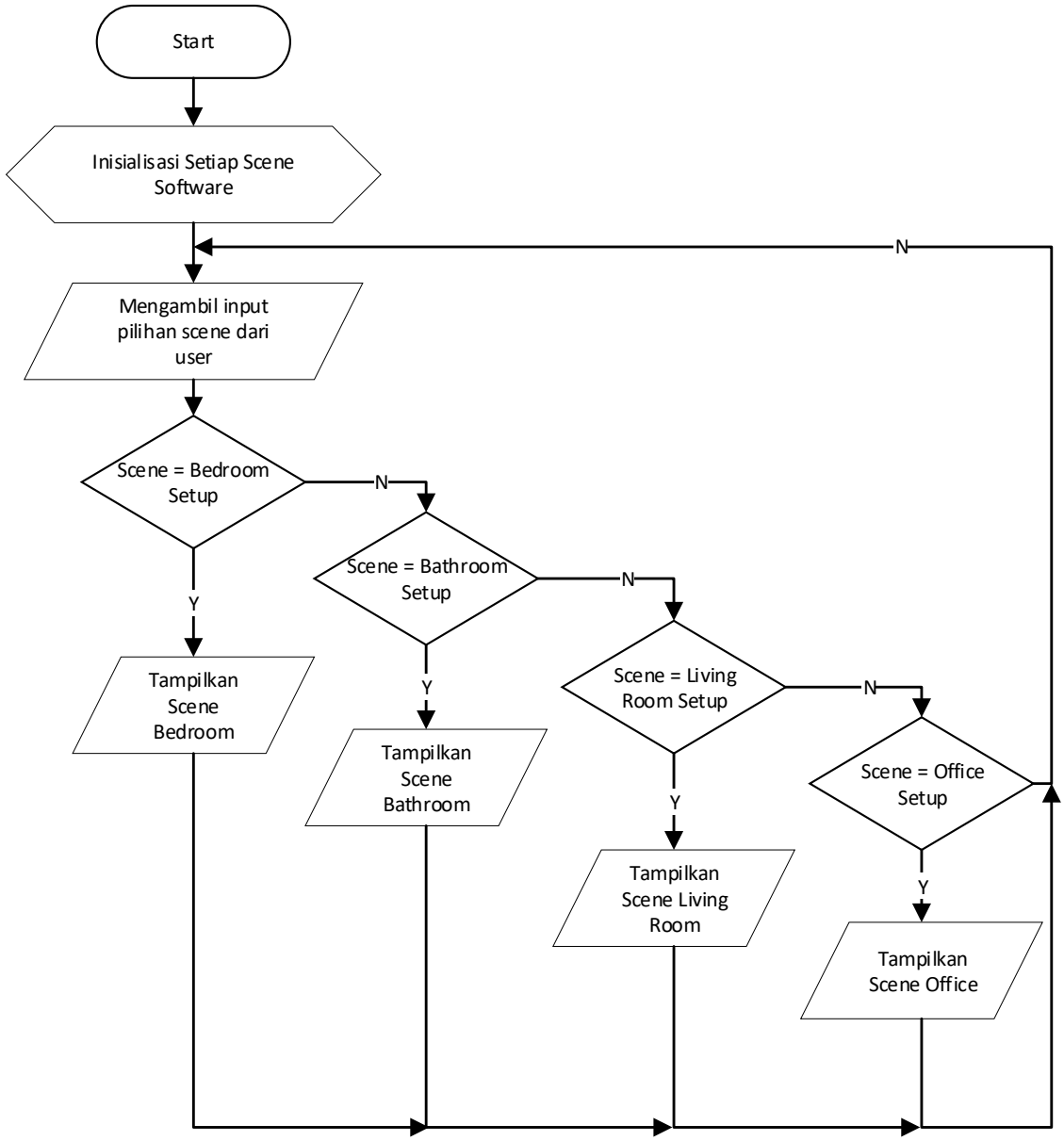
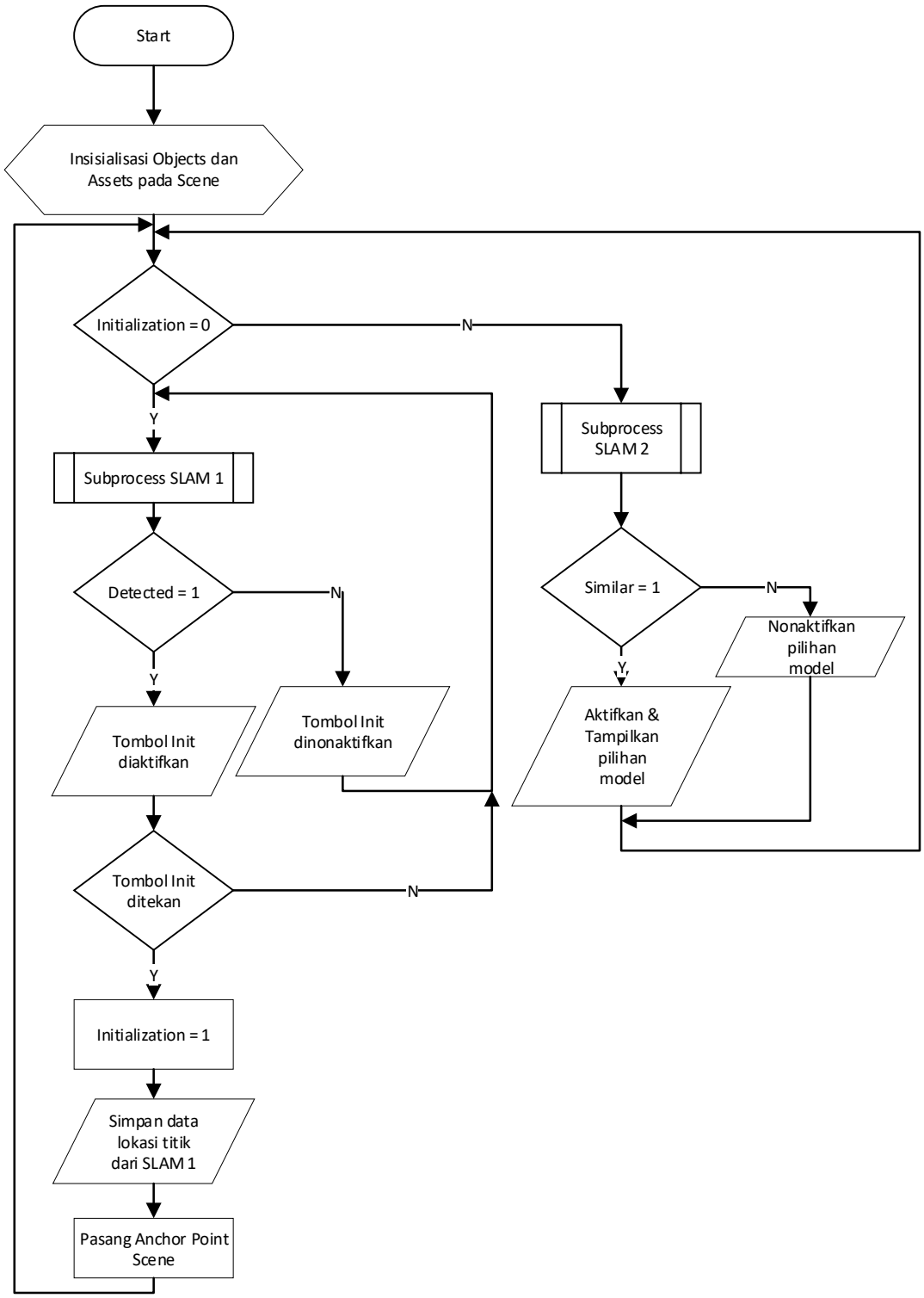
LAMPIRAN A

FLOWCHART

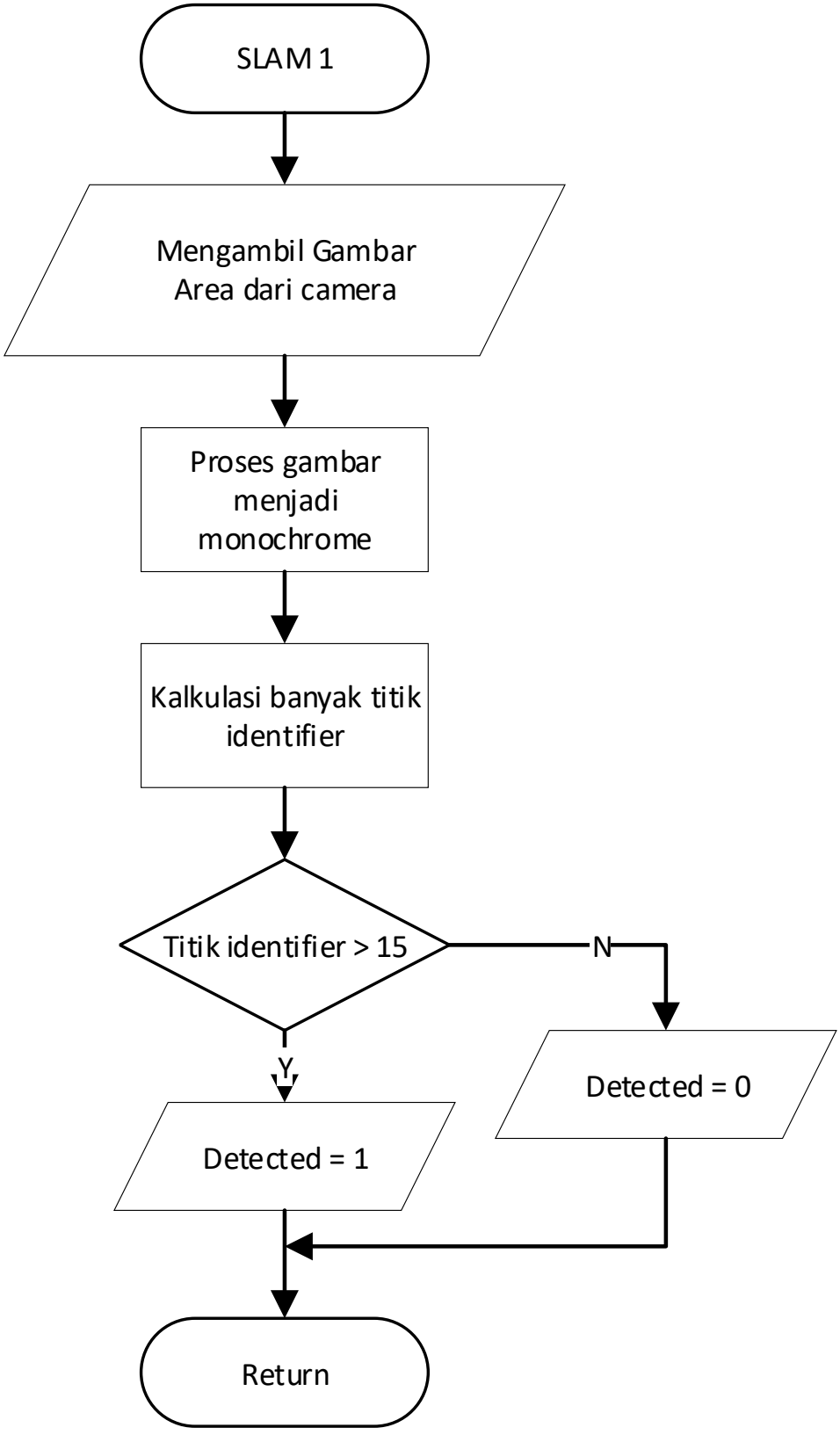
Flowchart Main Menu Scene



Flowchart Scene Opening



Flowchart Subprocess SLAM 1



Flowchart Subprocess SLAM 2

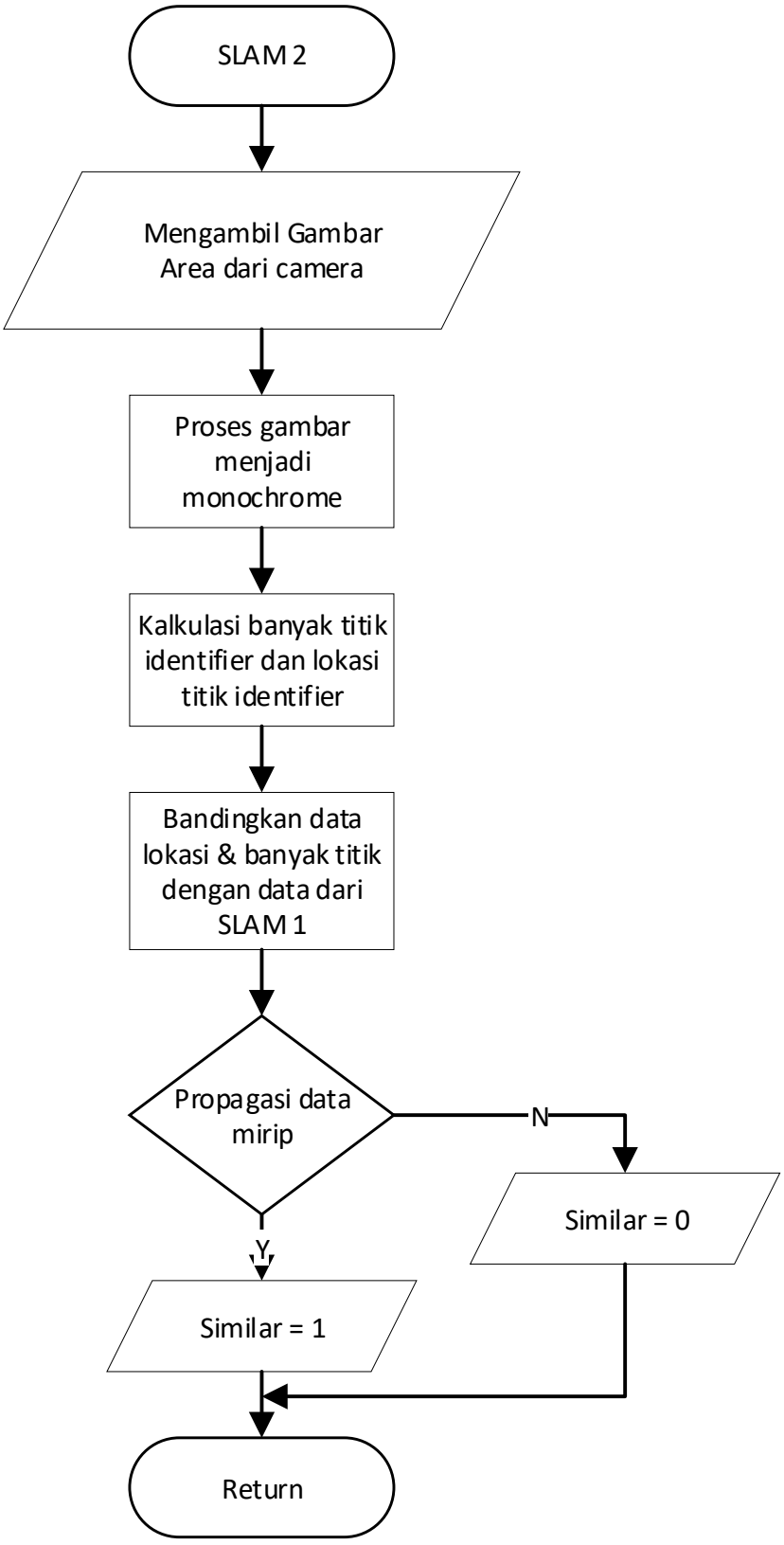
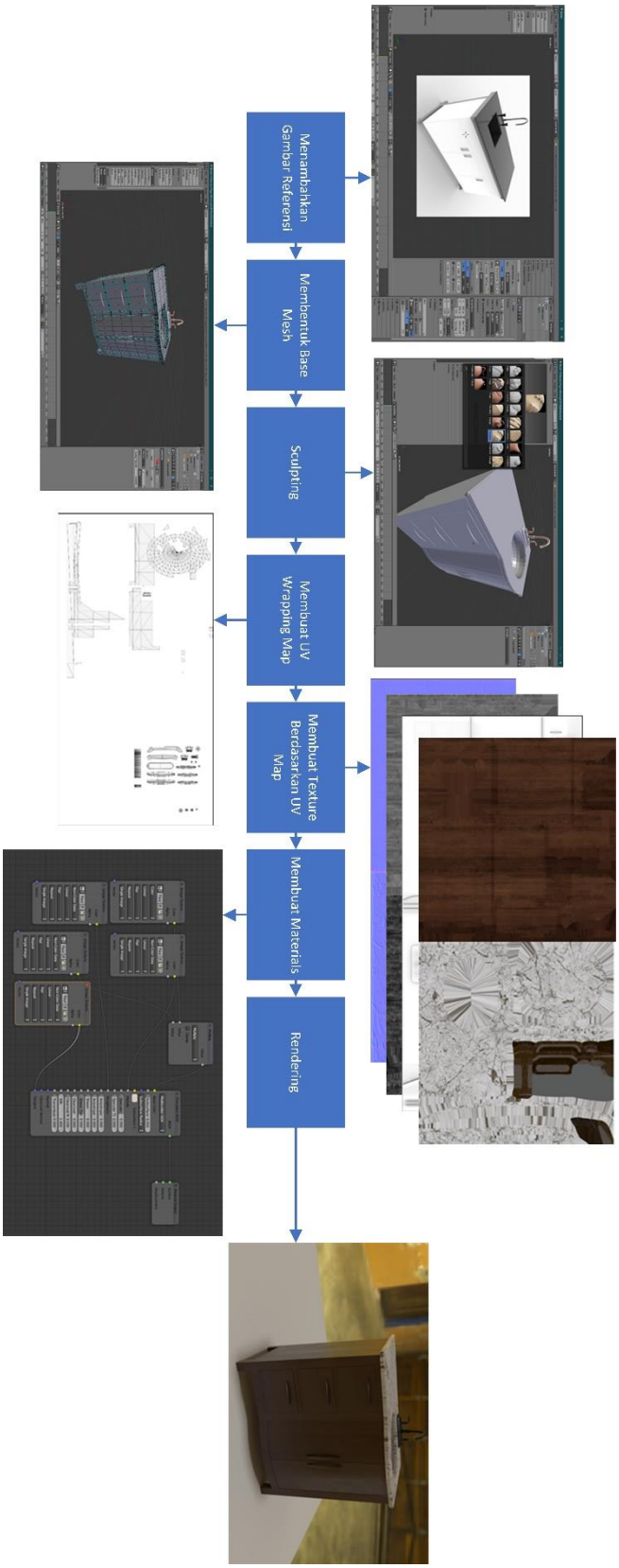


Diagram Langkah Pembuatan 3D Model



LAMPIRAN B

SOURCE CODE

AR Camera

```
using System;
using System.Collections.Generic;
using System.Runtime.InteropServices;
using UnityEngine;
using UnityEngine.Events;

namespace AR
{
  public class ARCamera : MonoBehaviour
  {
    [HideInInspector]
    public ARCamera.OnSDKInitializedEvent OnSDKInitialized;
    [SerializeField]
    [HideInInspector]
    private string _ARLicenseKey;
    [SerializeField]
    [HideInInspector]
    private bool _enableCameraRendering;
    [SerializeField]
    [HideInInspector]
    private bool _staticCamera;
    private Vector3 _calibrationPositionOffset;
    private Quaternion _calibrationRotationOffset;
    [SerializeField]
    [HideInInspector]
    private bool _ignoreTrackableScale;
    [SerializeField]
    [HideInInspector]
    public TransformOverride ActiveOverride;
    private ARBridge _bridge;
    private HashSet<TrackerBehaviour> _registeredTrackers;
    private GameObject _backgroundCamera;
    private bool _invertCulling;
    private bool _wasCullingInverted;
    private bool _initialized;
    [SerializeField]
    [HideInInspector]
    public ARCamera.OnCameraErrorEvent OnCameraError;
    [SerializeField]
    [HideInInspector]
    public                    ARCamera.OnCameraOpenedEvent
OnCameraOpened;
    private CapturePosition _cachedDevicePosition;
    private CaptureFocusMode _cachedFocusMode;
    [SerializeField]
    [HideInInspector]
    private CaptureDeviceResolution _desiredDeviceResolution;
    [SerializeField]
    [HideInInspector]
    private CaptureDeviceFramerate _desiredDeviceFramerate;
    [SerializeField]
    [HideInInspector]
    private bool _enableCamera2;
    [SerializeField]
    [HideInInspector]
    private bool _enableLivePreview;
    [SerializeField]
    [HideInInspector]
    private LivePreviewMode _livePreviewMode;
    private Texture2D _previousStaticImage;
    [SerializeField]
    [HideInInspector]
    private Texture2D _staticImage;
    [SerializeField]
    [HideInInspector]
    private string _webCamName;
    [SerializeField]
    [HideInInspector]
    private CapturePosition _remoteCameraPosition;
    private LivePreviewMode? _previousMode;
    private WebCamTexture _feed;
    private Color32[] _pixels;
    private byte[] _bytes;
    private ulong _frameIndex;
    private long _renderFrameIndex;
    private RingBuffer _ringBuffer;
    private int _previousWidth;
    private int _previousHeight;
    private HashSet<Plugin> _plugins;
    private bool _hasInputPlugins;
    private bool _requestsCameraFrameRendering;
    private Texture2D _inputPluginTexture;

    public string ARLicenseKey
    {
      get
      {
        return this._ARLicenseKey;
      }
      set
      {
        if (!this.Initialized)
          this._ARLicenseKey = value;
        else
          this.PrintSetAfterInitializationError(nameof
(ARLicenseKey));
      }
    }

    public bool EnableCameraRendering
    {
      get
      {
        return this._enableCameraRendering;
      }
      set
      {
        if (value == this._enableCameraRendering)
          return;
        this._enableCameraRendering = value;
        this.InitializeCamera();
      }
    }

    public bool StaticCamera
    {
      get
      {
        return this._staticCamera;
      }
      set
      {
        this._staticCamera = value;
      }
    }

    public Vector3 CalibrationPositionOffset
    {
      get
      {
        return this._calibrationPositionOffset;
      }
      set
      {
        this._calibrationPositionOffset = value;
      }
```

```
  }

  public Quaternion CalibrationRotationOffset
  {
    get
    {
      return this._calibrationRotationOffset;
    }
    set
    {
      this._calibrationRotationOffset = value;
    }
  }

  public bool IgnoreTrackableScale
  {
    get
    {
      return this._ignoreTrackableScale;
    }
    set
    {
      this._ignoreTrackableScale = value;
    }
  }

  public Texture CameraTexture
  {
    get
    {
      return ARSDK.PlatformBridge.GetRenderTexture();
    }
  }

  internal bool Initialized { get; private set; }

  internal void RegisterTracker(TrackerBehaviour tracker)
  {
    if (this._registeredTrackers.Contains(tracker))
      return;
    this._registeredTrackers.Add(tracker);
  }

  private void Awake()
  {
    if (this._enableLivePreview || Application.get_platform() !=
null && Application.get_platform() != 7)
      return;
    ARSDK.ForceUnityBridge();
  }

  private void Start()
  {
    if (Application.get_platform() == null ||
Application.get_platform() == 7)
      Application.set_runInBackground(true);
    if (PlatformBase.Instance == null)
      Debug.LogError((object) "AR SDK ERROR: Platform
was not properly initialized. Please make sure that the
Platform.cs script is part of your build!");
    this._bridge.Setup();
    ((Object) this._bridge.getTrackerManager()).get_name();
    ARSDK.PlatformBridge.SetCaptureDeviceResolution((int)
this._desiredDeviceResolution);
    ARSDK.PlatformBridge.SetCaptureDeviceFramerate((int)
this._desiredDeviceFramerate);

ARSDK.PlatformBridge.EnableCamera2(this._enableCamera2);

ARSDK.ARBridge.InstantiateARNativeSDK(this._ARLicense
Key,
ARSDK.PlatformBridge.InstantiatePlatform(Application.get_te
mporaryCachePath()),
ARSDK.PlatformBridge.GetSDKMutex());

ARSDK.PlatformBridge.SetLicenseKey(this._ARLicenseKey);
    ARSDK.PlatformBridge.Start();
    this.InitializeCamera();
    this.InitializeNativeCamera();
    this.UpdateRenderTexture();
    ARSDK.ARBridge.StartARNativeSDK();
    this.Initialized = true;
    foreach (TrackerBehaviour registeredTracker in
this._registeredTrackers)
      registeredTracker.OnARCameraInitialized();
    if (Object.op_Inequality((Object)
PluginManager.WeakInstance, (Object) null))
      PluginManager.WeakInstance.Initialize(this);
    this.UpdateCulling();
    this.OnSDKInitialized.Invoke();
    this._initialized = true;
  }

  private void OnEnable()
  {
    if (!this._initialized)
      return;
    ARSDK.ARBridge.StartARNativeSDK();
  }

  private void OnDisable()
  {
    if (!this._initialized)
      return;
    ARSDK.ARBridge.StopARNativeSDK();
  }

  private void OnDestroy()
  {
    this._initialized = false;
    this._bridge.OnDestroy();
    ARSDK.ARBridge.StopARNativeSDK();
    ARSDK.ARBridge.DestroyARNativeSDK();
    ARSDK.PlatformBridge.DestroyPlatform();
    ARSDK.Destroy();
    this.DestroyLivePreview();
    this.UpdateCulling();
  }

  private void InitializeCamera()
  {
    if (this._enableCameraRendering)
    {
      Camera component = (Camera) ((Component)
this).GetComponent<Camera>();
      if (Object.op_Equality((Object) component, (Object) null))
      {
        Debug.LogError((object) "ARCamera script is not
attached to a camera!");
        ((Behaviour) this).set_enabled(false);
      }
      if ((!this._hasInputPlugins ||
this._requestsCameraFrameRendering) &&
component.get_clearFlags() != 3)
      {
        Debug.LogWarning((object) "ARCamera Clear Flags
need to be set to DepthOnly in order to work correctly.
Automatically switching to DepthOnly.");
        component.set_clearFlags((CameraClearFlags) 3);
      }
      if (component.get_orthographic())
      {
        Debug.LogWarning((object) "ARCamera needs to be set
to perspective in order to work correctly. Automatically
switching to perspective.");
        component.set_orthographic(false);
      }
      if (Object.op_Inequality((Object) this._backgroundCamera,
(Object) null))
      {
        Debug.LogError((object) "Recreating background camera
when it already exists");
      }
      else
      {
        this._backgroundCamera = new GameObject();
        ((Object)
this._backgroundCamera).set_name("BackgroundCamera");
```

```
        ((Object)
this._backgroundCamera).set_hideFlags((HideFlags) 61);

this._backgroundCamera.get_transform().SetParent(((Compone
nt) this).get_transform());
        M0                    m0                    =
this._backgroundCamera.AddComponent<Camera>();
        ((Camera) m0).set_clearFlags((CameraClearFlags) 2);
        ((Camera) m0).set_backgroundColor(Color.get_black());
        ((BackgroundCamera)
this._backgroundCamera.AddComponent<BackgroundCamera
>()).ARCamera = this;
      }
    }
    else                if                (Object.op_Equality((Object)
this._backgroundCamera, (Object) null))
    {
      Debug.LogError((object) "Destroying background camera
when it doesn't exits yet");
    }
    else
    {
      Object.Destroy((Object) this._backgroundCamera);
      this._backgroundCamera = (GameObject) null;
    }
  }

  private void InitializeNativeCamera()
  {
    ARSDK.PlatformBridge.SetCaptureDeviceResolution((int)
this._desiredDeviceResolution);
    ARSDK.PlatformBridge.SetCaptureDeviceFramerate((int)
this._desiredDeviceFramerate);
    ARSDK.PlatformBridge.SetCaptureDevicePosition((int)
this._cachedDevicePosition);
    this.UpdateCulling();
  }

  private void UpdateRenderTexture()
  {
    ARSDK.PlatformBridge.SetCaptureDeviceResolution((int)
this._desiredDeviceResolution);
    ARSDK.PlatformBridge.SetCaptureDeviceFramerate((int)
this._desiredDeviceFramerate);
    ARSDK.PlatformBridge.SurfaceChanged((uint)
Screen.get_width(), (uint) Screen.get_height());
  }

  private void UpdateCulling()
  {
    if (this._enableCameraRendering)
    {
      if                (this.CachedDevicePosition                ==
CapturedevicePosition.Front)
        this._invertCulling = true;
      else
        this._invertCulling = false;
    }
    else
      this._invertCulling = false;
  }

  private void OnPreRender()
  {
    this._wasCullingInverted = GL.get_invertCulling();
    GL.set_invertCulling(this._invertCulling);
  }

  private void OnPostRender()
  {
    GL.set_invertCulling(this._wasCullingInverted);
  }

  private void OnApplicationPause(bool paused)
  {
    if (paused)
      ARSDK.ARBridge.StopARNativeSDK();
    else
      ARSDK.ARBridge.StartARNativeSDK();
  }

  private void Update()
  {
    if                (this._enableCameraRendering                &&
(this.CameraTexture.get_width()    !=    Screen.get_width()    ||
this.CameraTexture.get_height() != Screen.get_height()))
      this.UpdateRenderTexture();
    ARSDK.PlatformBridge.Update();

ARSDK.PlatformBridge.UpdateOrientation(Screen.get_orientat
ion());
    this._bridge.InternalUpdate();
    if (!this.IsLivePreviewRunning)
      return;
    this.UpdateLivePreview();
  }

  public CapturedevicePosition DevicePosition
  {
    get
    {
      return this._cachedDevicePosition;
    }
    set
    {
      if (this._cachedDevicePosition == value)
        return;
      this._cachedDevicePosition = value;
      if (!this.Initialized)
        return;
      this.InitializeNativeCamera();
    }
  }

  internal CapturedevicePosition CachedDevicePosition
  {
    get
    {
      return this._cachedDevicePosition;
    }
  }

  public CaptureFocusMode FocusMode
  {
    get
    {
      this.CheckSDKInitialized(nameof                (FocusMode),
ARCamera.CallType.Getter);
      this._cachedFocusMode        =        (CaptureFocusMode)
ARSDK.PlatformBridge.GetCaptureFocusMode();
      return this._cachedFocusMode;
    }
    set
    {
      this.CheckSDKInitialized(nameof                (FocusMode),
ARCamera.CallType.Setter);
      ARSDK.PlatformBridge.SetCaptureFocusMode((int)
value);
      this._cachedFocusMode = value;
    }
  }

  public    bool    IsFocusModeSupported(CaptureFocusMode
focusMode)
  {
    this.CheckSDKInitialized(nameof (IsFocusModeSupported),
ARCamera.CallType.Method);
    return
ARSDK.PlatformBridge.IsCaptureFocusModeSupported((int)
focusMode);
  }

  public CaptureExposureMode ExposureMode
  {
    get
    {
      this.CheckSDKInitialized(nameof            (ExposureMode),
ARCamera.CallType.Getter);
```

```
      return (CaptureExposureMode)
ARSDK.PlatformBridge.GetCaptureExposureMode();
    }
    set
    {
     this.CheckSDKInitialized(nameof (ExposureMode),
ARCamera.CallType.Setter);
      ARSDK.PlatformBridge.SetCaptureExposureMode((int)
value);
    }
  }

  public bool
IsExposureModeSupported(CaptureExposureMode
exposureMode)
  {
   this.CheckSDKInitialized(nameof
(IsExposureModeSupported), ARCamera.CallType.Method);
    return
ARSDK.PlatformBridge.IsCaptureExposureModeSupported((in
t) exposureMode);
  }

  public float ManualFocusDistance
  {
   set
   {
    if (this._cachedFocusMode != CaptureFocusMode.Locked)
     this.OnCameraError.Invoke(new Error(11001,
"com.AR.unity.camera", "Cannot set Manual Focus Distance
when Focus Mode is not set to Locked!"));
    else if (Application.get_platform() == 11
&& !this._enableCamera2)
     this.OnCameraError.Invoke(new Error(11001,
"com.AR.unity.camera", "Cannot set Manual Focus Distance on
Android if Camera2 is not enabled!"));
    else if ((double) value < 0.0 || (double) value > 1.0)
     this.OnCameraError.Invoke(new Error(11003,
"com.AR.unity.camera", "Manual Focus should have a value
between 0 and 1!"));
    else
     ARSDK.PlatformBridge.SetManualFocusDistance(value);
   }
  }

  public bool IsManualFocusAvailable
  {
   get
   {
    this.CheckSDKInitialized(nameof
(IsManualFocusAvailable), ARCamera.CallType.Getter);
    if (Application.get_platform() == 11
&& !this._enableCamera2)
     return false;
    return ARSDK.PlatformBridge.IsManualFocusAvailable();
   }
  }

  public CaptureAutoFocusRestriction AutoFocusRestriction
  {
   get
   {
    if (Application.get_platform() != 8)
     this.OnCameraError.Invoke(new Error(11004,
"com.AR.unity.camera", "AutoFocusRestriction is only
available on iOS!"));
    this.CheckSDKInitialized(nameof (AutoFocusRestriction),
ARCamera.CallType.Getter);
    return (CaptureAutoFocusRestriction)
ARSDK.PlatformBridge.GetCaptureAutoFocusRestriction();
   }
   set
   {
    if (Application.get_platform() != 8)
     this.OnCameraError.Invoke(new Error(11004,
"com.AR.unity.camera", "AutoFocusRestriction is not available
on iOS!"));
    this.CheckSDKInitialized(nameof (AutoFocusRestriction),
ARCamera.CallType.Setter);
ARSDK.PlatformBridge.SetCaptureAutoFocusRestriction((int)
value);
   }
  }

  public bool IsAutoFocusRestrictionSupported
  {
   get
   {
    this.CheckSDKInitialized(nameof
(IsAutoFocusRestrictionSupported),
ARCamera.CallType.Getter);
    return
ARSDK.PlatformBridge.IsCaptureAutoFocusRestrictionSuppor
ted();
   }
  }

  public float ZoomLevel
  {
   get
   {
    this.CheckSDKInitialized(nameof (ZoomLevel),
ARCamera.CallType.Getter);
    return ARSDK.PlatformBridge.GetCaptureZoomLevel();
   }
   set
   {
    this.CheckSDKInitialized(nameof (ZoomLevel),
ARCamera.CallType.Setter);
    ARSDK.PlatformBridge.SetCaptureZoomLevel(value);
   }
  }

  public bool IsZoomLevelSupported(float zoomLevel)
  {
   this.CheckSDKInitialized(nameof (IsZoomLevelSupported),
ARCamera.CallType.Getter);
   return
ARSDK.PlatformBridge.IsCaptureZoomLevelSupported(zoom
Level);
  }

  public float MaxZoomLevel
  {
   get
   {
    this.CheckSDKInitialized(nameof (MaxZoomLevel),
ARCamera.CallType.Getter);
    return
ARSDK.PlatformBridge.GetCaptureMaxZoomLevel();
   }
  }

  public CaptureFlashMode FlashMode
  {
   get
   {
    this.CheckSDKInitialized(nameof (FlashMode),
ARCamera.CallType.Getter);
    return (CaptureFlashMode)
ARSDK.PlatformBridge.GetCaptureFlashMode();
   }
   set
   {
    this.CheckSDKInitialized(nameof (FlashMode),
ARCamera.CallType.Setter);
    ARSDK.PlatformBridge.SetCaptureFlashMode((int)
value);
   }
  }

  public bool IsFlashModeSupported(CaptureFlashMode
flashMode)
  {
   this.CheckSDKInitialized(nameof (IsFlashModeSupported),
ARCamera.CallType.Getter);
```

```
    return
ARSDK.PlatformBridge.IsCaptureFlashModeSupported((int)
flashMode);
  }

  public CaptureDeviceResolution DesiredCameraResolution
  {
    get
    {
      return this._desiredDeviceResolution;
    }
    set
    {
      if (value == this._desiredDeviceResolution)
        return;
      if (!this.Initialized)
      {

ARSDK.PlatformBridge.SetCaptureDeviceResolution((int)
value);
        this._desiredDeviceResolution = value;
      }
      else
        this.PrintSetAfterInitializationError(nameof
(DesiredCameraResolution));
    }
  }

  public CaptureDeviceFramerate DesiredCameraFramerate
  {
    get
    {
      return this._desiredDeviceFramerate;
    }
    set
    {
      if (value == this._desiredDeviceFramerate)
        return;
      if (!this.Initialized)
      {
        ARSDK.PlatformBridge.SetCaptureDeviceFramerate((int)
value);
        this._desiredDeviceFramerate = value;
      }
      else
        this.PrintSetAfterInitializationError(nameof
(DesiredCameraFramerate));
    }
  }

  public bool EnableCamera2
  {
    get
    {
      return this._enableCamera2;
    }
    set
    {
      if (value == this._enableCamera2)
        return;
      if (!this.Initialized)
        this._enableCamera2 = value;
      else
        this.PrintSetAfterInitializationError(nameof
(EnableCamera2));
    }
  }

  public void FocusAtPointOfInterest(Vector2 pointOfInterest)
  {
    this.CheckSDKInitialized("FocusAtPointOfInterst",
ARCamera.CallType.Getter);
    ARSDK.PlatformBridge.FocusAtPointOfInterest((float)
pointOfInterest.x, (float) pointOfInterest.y);
  }

  public bool IsFocusAtPointOfInterestSupported
  {
    get
    {
      this.CheckSDKInitialized(nameof
(IsFocusAtPointOfInterestSupported),
ARCamera.CallType.Getter);
      return
ARSDK.PlatformBridge.IsFocusAtPointOfInterestSupported();
    }
  }

  public void ExposeAtPointOfInterest(Vector2
pointOfInterest, CaptureExposureMode exposureMode)
  {
    this.CheckSDKInitialized(nameof
(ExposeAtPointOfInterest), ARCamera.CallType.Getter);
    ARSDK.PlatformBridge.ExposeAtPointOfInterest((float)
pointOfInterest.x, (float) pointOfInterest.y, (int) exposureMode);
  }

  public bool IsExposeAtPointOfInterstSupported
  {
    get
    {

this.CheckSDKInitialized("IsExposeAtPointOfInterestSupporte
d", ARCamera.CallType.Getter);
      return
ARSDK.PlatformBridge.IsExposeAtPointOfInterestSupported();
    }
  }

  private void CheckSDKInitialized(string callName,
ARCamera.CallType callType)
  {
    if (this.Initialized)
      return;
    string str1;
    switch (callType)
    {
      case ARCamera.CallType.Getter:
        str1 = "Cannot get the value of " + callName;
        break;
      case ARCamera.CallType.Setter:
        str1 = "Cannot set the value of " + callName;
        break;
      default:
        str1 = "Cannot call " + callName;
        break;
    }
    string str2 = " because the SDK was not initialized yet.
Please use the OnSDKInitialized callback to know when it's
safe to access these properties.";
    this.OnCameraError.Invoke(new Error(11005,
"com.AR.unity.camera", str1 + str2));
  }

  private void PrintSetAfterInitializationError(string
propertyName)
  {
    this.OnCameraError.Invoke(new Error(11006,
"com.AR.unity.camera", "Cannot change the value of the " +
propertyName + " because the ARCamera was already
initialized. This property can only be changed before Start is
called."));
  }

  public bool EnableLivePreview
  {
    get
    {
      return this._enableLivePreview;
    }
    set
    {
      this._enableLivePreview = value;
    }
  }

  public LivePreviewMode LivePreviewMode
  {
```

```
  get
  {
   return this._livePreviewMode;
  }
  set
  {
   this._livePreviewMode = value;
  }
 }

 public Texture2D StaticImage
 {
  get
  {
   return this._staticImage;
  }
  set
  {
   this._staticImage = value;
  }
 }

 public string WebCamName
 {
  get
  {
   return this._webCamName;
  }
  set
  {
   this._webCamName = value;
  }
 }

 private bool IsWebCamNameValid
 {
  get
  {
   if (this._webCamName != null)
    return this._webCamName.Length > 0;
   return false;
  }
 }

 public CapturedevicePosition RemoteCameraPosition
 {
  get
  {
   return this._remoteCameraPosition;
  }
  set
  {
   this._remoteCameraPosition = value;
  }
 }

 internal bool ForceResendData { get; set; }

 internal bool IsLivePreviewRunning
 {
  get
  {
   if (!this._enableLivePreview)
    return false;
   if (Application.get_platform() != null)
    return Application.get_platform() == 7;
   return true;
  }
 }

 private Texture CurrentTexture
 {
  get
  {
   if (this._hasInputPlugins)
    return (Texture) this._inputPluginTexture;
   switch (this.LivePreviewMode)
   {
    case LivePreviewMode.StaticImage:
     return (Texture) this.StaticImage;
    case LivePreviewMode.WebCam:
    case LivePreviewMode.RemoteCamera:
     return (Texture) this._feed;
    default:
     return (Texture) null;
   }
  }
 }

 private RenderTexture CurrentRenderTexture
 {
  get
  {
   if (this._renderFrameIndex == -1L)
    return this._ringBuffer.GetLatestTexture();
   return this._ringBuffer.GetTexture((ulong)
this._renderFrameIndex);
  }
 }

 private int CurrentWidth
 {
  get
  {
   return this.CurrentTexture.get_width();
  }
 }

 private int CurrentHeight
 {
  get
  {
   return this.CurrentTexture.get_height();
  }
 }

 internal void SetGL()
 {
  if (Object.op_Equality((Object) this.CurrentTexture, (Object)
null))
  {
   GL.LoadPixelMatrix(0.0f, 1f, 0.0f, 1f);
  }
  else
  {
   Vector3 one = Vector3.get_one();
   float num1 = 0.0f;
   if (this.LivePreviewMode ==
LivePreviewMode.RemoteCamera)
    num1 = (float) this._feed.get_videoRotationAngle();
   float num2 = (float) Screen.get_width() / (float)
Screen.get_height();
   float num3 = (double) num1 == 0.0 || (double) num1 ==
180.0 ? (float) this.CurrentTexture.get_width() / (float)
this.CurrentTexture.get_height() : (float)
this.CurrentTexture.get_height() / (float)
this.CurrentTexture.get_width();
   if ((double) num1 == 0.0 || (double) num1 == 180.0)
   {
    if ((double) num2 > (double) num3)
     one.y = (__Null) ((double) num2 / (double) num3);
    else
     one.x = (__Null) ((double) num3 / (double) num2);
   }
   else if ((double) num2 > (double) num3)
    one.x = (__Null) ((double) num2 / (double) num3);
   else
    one.y = (__Null) ((double) num3 / (double) num2);
   if (this._remoteCameraPosition ==
CapturedevicePosition.Front)
    GL.LoadPixelMatrix(1f, 0.0f, 1f, 0.0f);
   else
    GL.LoadPixelMatrix(0.0f, 1f, 1f, 0.0f);

GL.MultMatrix(Matrix4x4.op_Multiply(Matrix4x4.op_Multipl
y(Matrix4x4.TRS(new Vector3(0.5f, 0.5f, 0.0f),
Quaternion.get_identity(), Vector3.get_one()),
Matrix4x4.TRS(Vector3.get_zero(), Quaternion.Euler(0.0f, 0.0f,
```

```
num1), one)), Matrix4x4.TRS(new Vector3(-0.5f, -0.5f, 0.0f), Quaternion.get_identity(), Vector3.get_one())));
    }
  }

  private void UpdateLivePreview()
  {
   if (!this._hasInputPlugins && (!this._previousMode.HasValue || this.LivePreviewMode != this._previousMode.Value))
    {
     switch (this.LivePreviewMode)
     {
      case LivePreviewMode.StaticImage:
       ARSDK.PlatformBridge.IsUsingRemote = false;
       break;
      case LivePreviewMode.WebCam:
       ARSDK.PlatformBridge.IsUsingRemote = false;
       bool flag1 = false;
       foreach (WebCamDevice device in WebCamTexture.get_devices())
       {
        if (!this.IsWebCamNameValid || this._webCamName == ((WebCamDevice) ref device).get_name())
        {
         this._feed = new WebCamTexture(((WebCamDevice) ref device).get_name(), 640, 480);
         this._feed.Play();
         flag1 = true;
         break;
        }
       }
       if (!flag1 && this.IsWebCamNameValid)
       {
        Debug.LogError((object) ("Cannot find any webcams named '" + this._webCamName + "'!"));
        break;
       }
       break;
      case LivePreviewMode.RemoteCamera:
       Input.get_gyro().set_enabled(true);
       ARSDK.PlatformBridge.IsUsingRemote = true;
       bool flag2 = false;
       foreach (WebCamDevice device in WebCamTexture.get_devices())
       {
        if (((WebCamDevice) ref device).get_name().Contains("Remote") && this._remoteCameraPosition == CapturedevicePosition.Front == ((WebCamDevice) ref device).get_isFrontFacing())
        {
         this._feed = new WebCamTexture(((WebCamDevice) ref device).get_name(), 640, 480);
         this._feed.Play();
         flag2 = true;
         break;
        }
       }
       if (!flag2)
        return;
       break;
     }
     this._previousMode = new LivePreviewMode?(this.LivePreviewMode);
    }
    if (Object.op_Equality((Object) this.CurrentTexture, (Object) null))
    {
     if (this._hasInputPlugins)
      return;
     Debug.LogError((object) "Invalid live preview texture.");
    }
    else
    {
     if (!this._hasInputPlugins)
     {
      switch (this.LivePreviewMode)
      {
       case LivePreviewMode.StaticImage:
        if (Object.op_Equality((Object) this.StaticImage, (Object) null))
         return;
        break;
       case LivePreviewMode.WebCam:
        if (!this._feed.get_didUpdateThisFrame())
         return;
        break;
       case LivePreviewMode.RemoteCamera:
        if (!this._feed.get_didUpdateThisFrame() || this.CurrentWidth < 64 || this.CurrentHeight < 64)
         return;
        break;
      }
     }
     if (this.CurrentWidth <= 0 || this.CurrentHeight <= 0)
      Debug.LogError((object) ("Camera feed has unexpected size. Width: " + (object) this.CurrentWidth + " Height: " + (object) this.CurrentHeight));
     bool flag = this.ForceResendData;
     if (this.CurrentWidth != this._previousWidth || this.CurrentHeight != this._previousHeight || Object.op_Inequality((Object) this._previousStaticImage, (Object) this.StaticImage))
     {
      ARSDK.PlatformBridge.SetMetadata(58f, (uint) this.CurrentWidth, (uint) this.CurrentHeight);
      this._ringBuffer.SetSize(this.CurrentWidth, this.CurrentHeight);
      this._previousWidth = this.CurrentWidth;
      this._previousHeight = this.CurrentHeight;
      this._previousStaticImage = this.StaticImage;
      if (!this._hasInputPlugins)
      {
       switch (this.LivePreviewMode)
       {
        case LivePreviewMode.StaticImage:
         this._pixels = this.StaticImage.GetPixels32();
         if (this._pixels == null)
          Debug.LogError((object) "The selected static image is not readable!");
         flag = true;
         break;
        case LivePreviewMode.WebCam:
        case LivePreviewMode.RemoteCamera:
         this._pixels = new Color32[this.CurrentWidth * this.CurrentHeight];
         break;
       }
      }
     }
     if (!this._hasInputPlugins)
     {
      switch (this.LivePreviewMode)
      {
       case LivePreviewMode.StaticImage:
        if (this.ForceResendData)
        {
         this._pixels = this.StaticImage.GetPixels32();
         break;
        }
        break;
       case LivePreviewMode.WebCam:
       case LivePreviewMode.RemoteCamera:
        this._feed.GetPixels32(this._pixels);
        flag = true;
        break;
      }
     }
     if (this._pixels != null & flag && !this._hasInputPlugins)
     {
      GCHandle gcHandle = new GCHandle();
      try
      {
       gcHandle = GCHandle.Alloc((object) this._pixels, GCHandleType.Pinned);

ARSDK.PlatformBridge.NotifyNewInputFrame(++this._frameIndex, gcHandle.AddrOfPinnedObject());
```

```
        }
        finally
        {
          if (gcHandle != new GCHandle())
            gcHandle.Free();
        }
      }
      if (!this._hasInputPlugins ||
this._requestsCameraFrameRendering)
      {
        this._renderFrameIndex =
ARSDK.ARBridge.GetRenderFrameId();
        this._ringBuffer.AddTexture(this._frameIndex,
this.CurrentTexture);
        ARSDK.PlatformBridge.UpdateRenderTexture((Texture)
this.CurrentRenderTexture);
      }
      float num = 0.0f;
      if (!this._hasInputPlugins)
      {
        LivePreviewMode livePreviewMode =
this.LivePreviewMode;
        if (livePreviewMode != LivePreviewMode.StaticImage
&& (uint) (livePreviewMode - 1) <= 1U)
          num = (float) this._feed.get_videoRotationAngle();

ARSDK.PlatformBridge.CameraToSurfaceAngleChanged(num
);
      }
      switch (this.LivePreviewMode)
      {
        case LivePreviewMode.StaticImage:
        case LivePreviewMode.WebCam:

ARSDK.PlatformBridge.NotifyNewDeviceRotationEvent(new
float[16]
          {
            -0.01622f,
            -0.69259f,
            0.72115f,
            0.0f,
            -0.99964f,
            -0.00409f,
            -0.02641f,
            0.0f,
            0.02124f,
            -0.72132f,
            -0.69227f,
            0.0f,
            0.0f,
            0.0f,
            0.0f,
            1f
          });
          break;
        case LivePreviewMode.RemoteCamera:
          Matrix4x4 matrix4x4_1 =
Matrix4x4.op_Multiply(Math.GetRotationMatrix(180f, 0.0f,
0.0f), Math.GetRotationMatrix(0.0f, 0.0f, num));
          Matrix4x4 rotationMatrix =
Math.GetRotationMatrix(Input.get_gyro().get_attitude());
          Matrix4x4 transpose = ((Matrix4x4) ref
rotationMatrix).get_transpose();
          Matrix4x4 matrix4x4_2 =
Matrix4x4.op_Multiply(matrix4x4_1, transpose);
          float[] rotation = new float[16];
          for (int index = 0; index < 16; ++index)
            rotation[index] = ((Matrix4x4) ref
matrix4x4_2).get_Item(index);

ARSDK.PlatformBridge.NotifyNewDeviceRotationEvent(rotati
on);
          break;
      }
      ARSDK.PlatformBridge.Update();
      this.ForceResendData = false;
    }
  }
```

```
  internal bool HasInputPlugins
  {
    get
    {
      return this._hasInputPlugins;
    }
  }

  internal bool RequestsCameraFrameRendering
  {
    get
    {
      return this._requestsCameraFrameRendering;
    }
  }

  internal void CheckForInputPlugins()
  {
    this._hasInputPlugins = false;
    this._requestsCameraFrameRendering = false;
    foreach (Plugin plugin in this._plugins)
    {
      if (plugin.HasInputModule)
      {
        this._hasInputPlugins = true;
        if (plugin.RequestsCameraFrameRendering)
          this._requestsCameraFrameRendering = true;
      }
    }
  }

  internal void RegisterPlugin(Plugin plugin)
  {
    if (!this.EnableLivePreview)
      return;
    this._plugins.Add(plugin);
    this.CheckForInputPlugins();
    if (!this._hasInputPlugins)
      return;
    if (Object.op_Inequality((Object) this._feed, (Object) null))
      this._feed = (WebCamTexture) null;
    plugin.CameraReleased();
  }

  internal void DeregisterPlugin(Plugin plugin)
  {
    if (!this.EnableLivePreview
|| !this._plugins.Contains(plugin))
      return;
    this._plugins.Remove(plugin);
    this.CheckForInputPlugins();
  }

  internal void NotifyNewCameraFrame(string identifier,
CameraFrame cameraFrame)
  {
    if (cameraFrame.ColorMetadata.ColorSpace !=
FrameColorSpace.RGBA)
    {
      Debug.LogWarning((object) "Live Preview can only
display RGBA frames when using Input Plugins!");
    }
    else
    {
      int width = cameraFrame.ColorMetadata.Width;
      int height = cameraFrame.ColorMetadata.Height;
      int length = width * height;
      if (Object.op_Equality((Object) this._inputPluginTexture,
(Object) null) || ((Texture)
this._inputPluginTexture).get_width() != width || ((Texture)
this._inputPluginTexture).get_height() != height)
      {
        if (Object.op_Inequality((Object)
this._inputPluginTexture, (Object) null))
          Object.Destroy((Object) this._inputPluginTexture);
        this._inputPluginTexture = new Texture2D(width, height,
(TextureFormat) 4, false);
        this._pixels = new Color32[length];
        this._bytes = new byte[length * 4];
```

```
      }
      Marshal.Copy(cameraFrame.ColorData[0].Data,
this._bytes, 0, length * 4);
      GCHandle gcHandle = new GCHandle();
      try
      {
        gcHandle = GCHandle.Alloc((object) this._pixels,
GCHandleType.Pinned);
        Marshal.Copy(this._bytes, 0,
gcHandle.AddrOfPinnedObject(), length * 4);
      }
      finally
      {
        if (gcHandle != new GCHandle())
          gcHandle.Free();
      }
      this._inputPluginTexture.SetPixels32(this._pixels);
      this._inputPluginTexture.Apply(false, false);
    }
  }

  private void DestroyLivePreview()
  {
    if (!Object.op_Inequality((Object) this._feed, (Object) null))
      return;
    this._feed.Stop();
  }

  public ARCamera()
  {
    base.\u002Ector();
  }

  [Serializable]
  public class OnSDKInitializedEvent : UnityEvent
  {
    public OnSDKInitializedEvent()
    {
      base.\u002Ector();
    }
  }

  [Serializable]
  public class OnCameraErrorEvent : UnityEvent<Error>
  {
    public OnCameraErrorEvent()
    {
      base.\u002Ector();
    }
  }

  [Serializable]
  public class OnCameraOpenedEvent : UnityEvent
  {
    public OnCameraOpenedEvent()
    {
      base.\u002Ector();
    }
  }

  private enum CallType
  {
    Method,
    Getter,
    Setter,
  }
 }
}
```

AR SDK

```
using System.Collections.Generic;
using UnityEngine;

namespace AR
{
 public static class ARSDK
 {
  public const int RecommendedNumberOfConcurrentTargets
= 5;
  private static IARBridge _ARBridge;
  private static IPlatformBridge _platformBridge;
  private static Dictionary<RuntimePlatform, string>
_platforms;
  private static SDKBuildInformation
_cachedBuildInformation;

  internal static IARBridge ARBridge
  {
    get
    {
      if (ARSDK._ARBridge == null)
        ARSDK._ARBridge = ARSDK.CreateARBridge();
      return ARSDK._ARBridge;
    }
  }

  internal static bool ARBridgeExists
  {
    get
    {
      return ARSDK._ARBridge != null;
    }
  }

  internal static IPlatformBridge PlatformBridge
  {
    get
    {
      if (ARSDK._platformBridge == null)
        ARSDK._platformBridge =
ARSDK.CreatePlatformBridge();
      return ARSDK._platformBridge;
    }
  }

  internal static void Destroy()
  {
    ARSDK._ARBridge = (IARBridge) null;
    ARSDK._platformBridge = (IPlatformBridge) null;
  }

  private static IARBridge CreateARBridge()
  {
    string bridgeName;
    if
(ARSDK._platforms.TryGetValue(Application.get_platform(),
out bridgeName))
      return BridgeFactory.ConstructARBridge(bridgeName);
    Debug.Log((object) ("The current platfrom (" + (object)
Application.get_platform() + ") is not supported by the AR
Unity Plugin"));
    return BridgeFactory.ConstructARBridge("Unity");
  }

  private static IPlatformBridge CreatePlatformBridge()
  {
    string bridgeName;
    if
(ARSDK._platforms.TryGetValue(Application.get_platform(),
out bridgeName))
      return
BridgeFactory.ConstructPlatformBridge(bridgeName);
    return BridgeFactory.ConstructPlatformBridge("Unity");
  }

  internal static void ForceUnityBridge()
  {
    ARSDK._ARBridge =
BridgeFactory.ConstructARBridge("Unity");
    ARSDK._platformBridge =
BridgeFactory.ConstructPlatformBridge("Unity");
  }

  public static SDKBuildInformation BuildInformation
  {
    get
    {
      if (ARSDK._cachedBuildInformation == null)
```

```
      {
        string buildConfiguration = ARSDK.ARBridge.GetSDKBuildConfiguration();
        string sdkBuildNumber = ARSDK.ARBridge.GetSDKBuildNumber();
        string sdkBuildDate = ARSDK.ARBridge.GetSDKBuildDate();
        string str = "8.0.0";
        string buildNumber = sdkBuildNumber;
        string buildDate = sdkBuildDate;
        string sdkVersion = str;
        ARSDK._cachedBuildInformation = new SDKBuildInformation(buildConfiguration, buildNumber, buildDate, sdkVersion);
      }
      return ARSDK._cachedBuildInformation;
    }
  }

  static ARSDK()
  {
    Dictionary<RuntimePlatform, string> dictionary = new Dictionary<RuntimePlatform, string>();
    dictionary.Add((RuntimePlatform) 8, "iOS");
    dictionary.Add((RuntimePlatform) 11, "Android");
    dictionary.Add((RuntimePlatform) 0, "MacOSEditor");
    dictionary.Add((RuntimePlatform) 7, "WindowsEditor");
    dictionary.Add((RuntimePlatform) 19, "UWP");
    ARSDK._platforms = dictionary;
  }
 }
}
```

Instant Tracker

```
using System;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using UnityEngine.Events;

namespace AR
{
 public class InstantTracker : TrackerBehaviour
 {
  [HideInInspector]
  public InstantTracker.OnStateChangedEvent OnStateChanged = new InstantTracker.OnStateChangedEvent();
  [HideInInspector]
  public InstantTracker.OnErrorEvent OnError = new InstantTracker.OnErrorEvent();
  [SerializeField]
  [HideInInspector]
  private bool _smartEnabled = true;
  private float _deviceHeightAboveGround = 1f;
  [HideInInspector]
  public InstantTracker.OnScreenConversionComputedEvent OnScreenConversionComputed = new InstantTracker.OnScreenConversionComputedEvent();
  [HideInInspector]
  public InstantTracker.OnPointCloudRequestFinishedEvent OnPointCloudRequestFinished = new InstantTracker.OnPointCloudRequestFinishedEvent();
  [SerializeField]
  [HideInInspector]
  private PlaneFilter _planeFilter = PlaneFilter.HorizontalUpward;
  private HashSet<Plane> _trackedPlanes = new HashSet<Plane>();
  private HashSet<Plane> _cachedPlanesToAdd = new HashSet<Plane>();
  private HashSet<Plane> _cachedPlanesToRemove = new HashSet<Plane>();
  [SerializeField]
  [HideInInspector]
  private InstantTrackingPlaneOrientation _trackingPlaneOrientation;
  [SerializeField]
  [HideInInspector]
  private float _trackingPlaneOrientationAngle;
  [SerializeField]
  [HideInInspector]
  private TrackerEfficiencyMode _trackerEfficiencyMode;
  [SerializeField]
  [HideInInspector]
  private bool _planeDetectionEnabled;
  [SerializeField]
  [HideInInspector]
  private bool _convexHullEnabled;
  private bool _pointCloudRequestPending;
  private Action<string> _saveTargetSuccessHandler;
  private Action<Error> _saveTargetErrorHandler;
  private Action<string> _loadTargetSuccessHandler;
  private Action<Error> _loadTargetErrorHandler;
  internal InstantTrackingState CurrentState;
  private Action<SmartAvailability> _isPlatformAssistedTrackingSupportedCallback;
  private bool _instantiated;

  public bool SMARTEnabled
  {
   get
   {
    return this._smartEnabled;
   }
   set
   {
    this._smartEnabled = value;
   }
  }

  public float DeviceHeightAboveGround
  {
   get
   {
    return this._deviceHeightAboveGround;
   }
   set
   {
    if ((double) this._deviceHeightAboveGround == (double) value)
     return;
    this._deviceHeightAboveGround = value;
    if (!this.Initialized)
     return;
    ARSDK.ARBridge.SetDeviceHeightAboveGround(value);
   }
  }

  public InstantTrackingPlaneOrientation TrackingPlaneOrientation
  {
   get
   {
    return this._trackingPlaneOrientation;
   }
   set
   {
    this._trackingPlaneOrientation = value;
    if (!((Behaviour) this).get_enabled() || !this.Initialized)
     return;
    if (this.CurrentState == InstantTrackingState.Initializing)
    {
     if (this.TrackingPlaneOrientation == InstantTrackingPlaneOrientation.Custom)
      return;

ARSDK.ARBridge.SetTrackingPlaneOrientation(this.ActualPlaneOrientationAngle);
    }
    else
     Debug.LogWarning((object) "Cannot set the TrackingPlaneOrientation while the Tracker is in Tracking state!");
   }
  }

  public float TrackingPlaneOrientationAngle
  {
```

```
    get
    {
     return this._trackingPlaneOrientationAngle;
    }
    set
    {
     if (this.TrackingPlaneOrientation != InstantTrackingPlaneOrientation.Custom)
      Debug.LogWarning((object) "TrackingPlaneOrientationAngle is set to a new value, but TrackingPlaneOrientation is not set to Custom!");
     this._trackingPlaneOrientationAngle = value;
     if (!((Behaviour) this).get_enabled())
      return;
     if (this.CurrentState == InstantTrackingState.Initializing)
      ARSDK.ARBridge.SetTrackingPlaneOrientation(this.ActualPlaneOrientationAngle);
     else
      Debug.LogWarning((object) "Cannot set the TrackingPlaneOrientationAngle while the Tracker is in Tracking state!");
    }
   }

   private float ActualPlaneOrientationAngle
   {
    get
    {
     float num;
     switch (this.TrackingPlaneOrientation)
     {
      case InstantTrackingPlaneOrientation.Horizontal:
       num = 0.0f;
       break;
      case InstantTrackingPlaneOrientation.Vertical:
       num = 90f;
       break;
      case InstantTrackingPlaneOrientation.Custom:
       num = this.TrackingPlaneOrientationAngle;
       break;
      default:
       num = 0.0f;
       break;
     }
     return num;
    }
   }

   public TrackerEfficiencyMode TrackerEfficiencyMode
   {
    get
    {
     return this._trackerEfficiencyMode;
    }
    set
    {
     this._trackerEfficiencyMode = value;
    }
   }

   public bool PlaneDetectionEnabled
   {
    get
    {
     return this._planeDetectionEnabled;
    }
    set
    {
     this._planeDetectionEnabled = value;
    }
   }

   public PlaneFilter PlaneFilter
   {
    get
    {
     return this._planeFilter;
    }
    set
    {
     this._planeFilter = value;
    }
   }

   public bool ConvexHullEnabled
   {
    get
    {
     return this._convexHullEnabled;
    }
    set
    {
     this._convexHullEnabled = value;
    }
   }

   public void SetState(InstantTrackingState state)
   {
    ARSDK.ARBridge.SetState((int) state);
   }

   public bool CanStartTracking()
   {
    return ARSDK.ARBridge.CanStartTracking();
   }

   public void ConvertScreenCoordinate(Vector2 screenCoordinate)
   {
    ARSDK.ARBridge.ConvertScreenCoordinate(screenCoordinate);
   }

   internal void OnScreenConversionSuccessful(Vector2 screenCoordinate, Vector3 pointCloudCoordinate)
   {
    Vector3 vector3 = pointCloudCoordinate;
    this.OnScreenConversionComputed.Invoke(true, screenCoordinate, vector3);
   }

   internal void OnScreenConversionFailed(Vector2 screenCoordinate)
   {
    this.OnScreenConversionComputed.Invoke(false, screenCoordinate, Vector3.get_zero());
   }

   public void RequestPointCloud()
   {
    if (this._pointCloudRequestPending)
     Debug.LogError((object) "A new point cloud request was issued, but the previous request didn't complete yet!");
    this._pointCloudRequestPending = true;
    ARSDK.ARBridge.RequestPointCloud();
   }

   public void IsPlatformAssistedTrackingSupported(Action<SmartAvailability> callback)
   {
    if (this._isPlatformAssistedTrackingSupportedCallback != null)
     Debug.LogError((object) "IsPlatformAssistedTrackingSupported called before previous call completed!");
    else if (!this._smartEnabled)
    {
     Debug.LogError((object) "IsPlatformAssistedTrackingSupported called when SMART is disabled!");
    }
    else
    {
     this._isPlatformAssistedTrackingSupportedCallback = callback;
```

```
ARSDK.PlatformBridge.IsPlatformAssistedTrackingSupported(
false);
    }
  }

  internal                                                void
OnIsPlatformAssistedTrackingSupported(SmartAvailability
smartAvailability, bool internalCall)
  {
   if (internalCall)
   {
    if (this._instantiated)
     return;
    switch (smartAvailability)
    {
     case SmartAvailability.IndeterminateQueryFailed:
      ARSDK.ARBridge.InstantiateInstantTracker(false,
this._deviceHeightAboveGround,
this.ActualPlaneOrientationAngle,                       (int)
this._trackerEfficiencyMode, this.GetPlaneConfiguration());
      this._instantiated = true;
      break;
     case SmartAvailability.Unsupported:
     case SmartAvailability.SupportedUpdateRequired:
     case SmartAvailability.Supported:

ARSDK.ARBridge.InstantiateInstantTracker(this._smartEnable
d,                                  this._deviceHeightAboveGround,
this.ActualPlaneOrientationAngle,                       (int)
this._trackerEfficiencyMode, this.GetPlaneConfiguration());
      this._instantiated = true;
      break;
    }
   }
   else
   {
    if (this._isPlatformAssistedTrackingSupportedCallback ==
null)
     return;

this._isPlatformAssistedTrackingSupportedCallback(smartAvai
lability);
    }
  }

  internal void OnPointCloudRequestSuccessful(Vector3[]
pointCloud)
  {
   this._pointCloudRequestPending = false;
   this.OnPointCloudRequestFinished.Invoke(pointCloud);
  }

  public void SaveCurrentInstantTarget(string path,
Action<string> successHandler = null, Action<Error>
errorHandler = null)
  {
   this._saveTargetSuccessHandler = successHandler;
   this._saveTargetErrorHandler = errorHandler;
   ARSDK.ARBridge.SaveCurrentInstantTarget(path);
  }

  internal void OnSaveInstantTargetSuccessful(string path)
  {
   this._saveTargetSuccessHandler(path);
  }

  internal void OnSaveInstantTargetError(Error error)
  {
   this._saveTargetErrorHandler(error);
  }

  public void LoadInstantTarget(TargetCollectionResource
instantTargetResource, InstantTargetRestorationConfiguration
configuration, Action<string> successHandler = null,
Action<Error> errorHandler = null)
  {
   this._loadTargetSuccessHandler = successHandler;
   this._loadTargetErrorHandler = errorHandler;
   this.LoadInstantTarget(instantTargetResource,
configuration);
  }

  public void LoadInstantTarget(TargetCollectionResource
instantTargetResource, InstantTargetRestorationConfiguration
configuration)
  {
   instantTargetResource.Register(this.Manager);
   instantTargetResource.OnEnable();

ARSDK.ARBridge.LoadInstantTarget(instantTargetResource.I
dentifier, (int) configuration.ExpansionPolicy);
  }

  internal void OnLoadInstantTargetSuccessful(string path)
  {
   this._loadTargetSuccessHandler(path);
  }

  internal void OnLoadInstantTargetError(Error error)
  {
   this._loadTargetErrorHandler(error);
  }

  internal void OnStateChangedInternal(InstantTrackingState
newState)
  {
   this.CurrentState = newState;
   this.OnStateChanged.Invoke(newState);
  }

  internal void OnErrorInternal(Error error)
  {
   this.OnError.Invoke(error);
  }

  protected override void OnInitialize()
  {
  }

  protected override void OnActivate()
  {
   if (!this.Initialized)
    return;
   if (!this._smartEnabled)
   {

ARSDK.ARBridge.InstantiateInstantTracker(this._smartEnable
d,                                  this._deviceHeightAboveGround,
this.ActualPlaneOrientationAngle,                       (int)
this._trackerEfficiencyMode, this.GetPlaneConfiguration());
    this._instantiated = true;
   }
   else

ARSDK.PlatformBridge.IsPlatformAssistedTrackingSupported(
true);
  }

  protected override void OnDisable()
  {
   base.OnDisable();
   ARSDK.ARBridge.DestroyInstantTracker();
  }

  internal override void OnTargetsLoadedInternal()
  {
  }

  internal override void OnErrorLoadingTargetsInternal(Error
error)
  {
  }

  internal override RecognizedTarget CreateTarget(string name,
long id, float[] modelViewMatrix)
  {
```

```
    return (RecognizedTarget) new InstantTarget(name, id, modelViewMatrix);
   }

   private UnityPlaneConfiguration GetPlaneConfiguration()
   {
    UnityPlaneConfiguration planeConfiguration;
    planeConfiguration.planeDetectionEnabled = this.PlaneDetectionEnabled;
    planeConfiguration.planeFilter = (byte) 0;
    if ((this.PlaneFilter & PlaneFilter.HorizontalUpward) != (PlaneFilter) 0)
     planeConfiguration.planeFilter |= (byte) 1;
    if ((this.PlaneFilter & PlaneFilter.HorizontalDownward) != (PlaneFilter) 0)
     planeConfiguration.planeFilter |= (byte) 2;
    if ((this.PlaneFilter & PlaneFilter.Vertical) != (PlaneFilter) 0)
     planeConfiguration.planeFilter |= (byte) 4;
    if ((this.PlaneFilter & PlaneFilter.Arbitrary) != (PlaneFilter) 0)
     planeConfiguration.planeFilter |= (byte) 8;
    planeConfiguration.convexHullEnabled = this.ConvexHullEnabled;
    return planeConfiguration;
   }

   internal void PlanesChanged(HashSet<Plane> currentPlanes)
   {
    foreach (RecognizedTarget currentPlane in currentPlanes)
    {
     if (!Math.CheckMatrix(currentPlane.ModelViewMatrix))
      return;
    }
    InstantTrackable instantTrackable = (InstantTrackable) null;
    if (this._registeredTrackables.Count == 1)
     instantTrackable = this._registeredTrackables.First<KeyValuePair<Trackable, HashSet<RecognizedTarget>>>().Key as InstantTrackable;
    if (!Object.op_Inequality((Object) instantTrackable, (Object) null))
     return;
    this._cachedPlanesToAdd.Clear();
    this._cachedPlanesToRemove.Clear();
    foreach (Plane trackedPlane in this._trackedPlanes)
    {
     if (!currentPlanes.Contains(trackedPlane))
      this._cachedPlanesToRemove.Add(trackedPlane);
    }
    foreach (Plane currentPlane in currentPlanes)
    {
     if (!currentPlane.IsKnown)
     {
      this._cachedPlanesToAdd.Add(currentPlane);
      currentPlane.IsKnown = true;
     }
    }
    foreach (Plane lostPlane in this._cachedPlanesToRemove)
    {
     this._trackedPlanes.Remove(lostPlane);
     instantTrackable.OnPlaneLostInternal(lostPlane);
    }
    foreach (Plane recognizedPlane in this._cachedPlanesToAdd)
    {
     this._trackedPlanes.Add(recognizedPlane);

instantTrackable.OnPlaneRecognizedInternal(recognizedPlane);
    }
    TransformProperties baseCamera = this.ConvertProjectionMatrix(this._sceneCamera.get_fieldOfView());
    if (this._ARCamera.StaticCamera)
    {
     foreach (Plane trackedPlane in this._trackedPlanes)
     {
      GameObject drawable = trackedPlane.Drawable;
      this.UpdateDrawableForStaticCamera(baseCamera, trackedPlane.ModelViewMatrix, 1f, ((Component) instantTrackable).get_transform(), drawable.get_transform()).ApplyGlobal(drawable.get_transform());
     }
    }
    else
    {
     if (this._anchorTarget == null)
      return;
     TransformProperties positionForTarget1 = this.ComputeCameraPositionForTarget(baseCamera, this._anchorTarget.ModelViewMatrix, 1f, ((Component) instantTrackable).get_transform(), Vector3.get_zero(), Quaternion.get_identity());
     foreach (Plane trackedPlane in this._trackedPlanes)
     {
      GameObject drawable = trackedPlane.Drawable;
      TransformProperties positionForTarget2 = this.ComputeCameraPositionForTarget(baseCamera, trackedPlane.ModelViewMatrix, 1f, ((Component) instantTrackable).get_transform(), Vector3.get_zero(), Quaternion.get_identity());
      TransformProperties transformation = this.UpdateDrawable(positionForTarget1, positionForTarget2, drawable.get_transform());
      instantTrackable.OnPlaneTrackedInternal(trackedPlane, transformation);
     }
    }
   }

   [Serializable]
   public class OnStateChangedEvent : UnityEvent<InstantTrackingState>
   {
    public OnStateChangedEvent()
    {
     base.\u002Ector();
    }
   }

   [Serializable]
   public class OnErrorEvent : UnityEvent<Error>
   {
    public OnErrorEvent()
    {
     base.\u002Ector();
    }
   }

   [Serializable]
   public class OnScreenConversionComputedEvent : UnityEvent<bool, Vector2, Vector3>
   {
    public OnScreenConversionComputedEvent()
    {
     base.\u002Ector();
    }
   }

   [Serializable]
   public class OnPointCloudRequestFinishedEvent : UnityEvent<Vector3[]>
   {
    public OnPointCloudRequestFinishedEvent()
    {
     base.\u002Ector();
    }
   }
  }
}
```

Instant Trackable

```
using System;
using UnityEngine;
using UnityEngine.Events;

namespace AR
```

```
{
  public class InstantTrackable : Trackable
  {
    [HideInInspector]
    public InstantTrackable.OnInitializationStartedEvent OnInitializationStarted = new InstantTrackable.OnInitializationStartedEvent();
    [HideInInspector]
    public InstantTrackable.OnInitializationStoppedEvent OnInitializationStopped = new InstantTrackable.OnInitializationStoppedEvent();
    [HideInInspector]
    public InstantTrackable.OnSceneRecognizedEvent OnSceneRecognized = new InstantTrackable.OnSceneRecognizedEvent();
    [HideInInspector]
    public InstantTrackable.OnSceneLostEvent OnSceneLost = new InstantTrackable.OnSceneLostEvent();
    public InstantTrackable.OnPlaneRecognizedEvent OnPlaneRecognized = new InstantTrackable.OnPlaneRecognizedEvent();
    public InstantTrackable.OnPlaneTrackedEvent OnPlaneTracked = new InstantTrackable.OnPlaneTrackedEvent();
    public InstantTrackable.OnPlaneLostEvent OnPlaneLost = new InstantTrackable.OnPlaneLostEvent();
    [SerializeField]
    [HideInInspector]
    private GameObject _planeDrawable;

    public GameObject PlaneDrawable
    {
      get
      {
        return this._planeDrawable;
      }
      set
      {
        this._planeDrawable = value;
      }
    }

    internal override bool IncludesTargetName(string targetName)
    {
      return true;
    }

    internal override void OnTargetRecognized(RecognizedTarget recognizedTarget)
    {
      base.OnTargetRecognized(recognizedTarget);
      InstantTarget instantTarget = recognizedTarget as InstantTarget;
      if (recognizedTarget.Name == "instant_target")
        this.OnSceneRecognized.Invoke(instantTarget);
      else
        this.OnInitializationStarted.Invoke(instantTarget);
    }

    internal override void OnTargetLost(RecognizedTarget recognizedTarget)
    {
      InstantTarget instantTarget = recognizedTarget as InstantTarget;
      if (recognizedTarget.Name == "instant_target")
        this.OnSceneLost.Invoke(instantTarget);
      else
        this.OnInitializationStopped.Invoke(instantTarget);
      base.OnTargetLost(recognizedTarget);
    }

    internal void OnPlaneRecognizedInternal(Plane recognizedPlane)
    {
      this.OnRecognizedTargetRecognized((RecognizedTarget) recognizedPlane, this._planeDrawable);
      this.OnPlaneRecognized.Invoke(recognizedPlane);
    }

    internal void OnPlaneTrackedInternal(Plane trackedPlane, TransformProperties transformation)
    {
      this.OnRecognizedTargetTracked((RecognizedTarget) trackedPlane, transformation);
      this.OnPlaneTracked.Invoke(trackedPlane);
    }

    internal void OnPlaneLostInternal(Plane lostPlane)
    {
      this.OnRecognizedTargetLost((RecognizedTarget) lostPlane);
      this.OnPlaneLost.Invoke(lostPlane);
    }

    [Serializable]
    public class OnInitializationStartedEvent : UnityEvent<InstantTarget>
    {
      public OnInitializationStartedEvent()
      {
        base.\u002Ector();
      }
    }

    [Serializable]
    public class OnInitializationStoppedEvent : UnityEvent<InstantTarget>
    {
      public OnInitializationStoppedEvent()
      {
        base.\u002Ector();
      }
    }

    [Serializable]
    public class OnSceneRecognizedEvent : UnityEvent<InstantTarget>
    {
      public OnSceneRecognizedEvent()
      {
        base.\u002Ector();
      }
    }

    [Serializable]
    public class OnSceneLostEvent : UnityEvent<InstantTarget>
    {
      public OnSceneLostEvent()
      {
        base.\u002Ector();
      }
    }

    [Serializable]
    public class OnPlaneRecognizedEvent : UnityEvent<Plane>
    {
      public OnPlaneRecognizedEvent()
      {
        base.\u002Ector();
      }
    }

    [Serializable]
    public class OnPlaneTrackedEvent : UnityEvent<Plane>
    {
      public OnPlaneTrackedEvent()
      {
        base.\u002Ector();
      }
    }

    [Serializable]
    public class OnPlaneLostEvent : UnityEvent<Plane>
    {
      public OnPlaneLostEvent()
      {
        base.\u002Ector();
```

```
    }
   }
  }
}
```

Instant Tracking State

```
namespace AR
{
 public enum InstantTrackingState
 {
  Initializing,
  Tracking,
 }
}
```

Instant Track Plane Orientation

```
namespace AR
{
 public enum InstantTrackingPlaneOrientation
 {
  Horizontal,
  Vertical,
  Custom,
 }
}
```

Background Camera

```
using UnityEngine;

namespace AR
{
 internal class BackgroundCamera : MonoBehaviour
 {
  internal ARCamera ARCamera;

  private void Awake()
  {
   M0 component = ((Component) this).GetComponent<Camera>();
   ((Camera) component).set_depth(Camera.get_main().get_depth() - 1f);
   ((Camera) component).set_cullingMask(0);
  }

  private void OnPreRender()
  {
   ARSDK.PlatformBridge.OnPreRender();
  }

  private void OnPostRender()
  {
   if (this.ARCamera.HasInputPlugins && !this.ARCamera.RequestsCameraFrameRendering)
    return;
   GL.Viewport(new Rect(0.0f, 0.0f, (float) Screen.get_width(), (float) Screen.get_height()));
   GL.PushMatrix();
   if (ARSDK.PlatformBridge.IsEditor)
    this.ARCamera.SetGL();
   else if (SystemInfo.get_graphicsDeviceType() == 16 || SystemInfo.get_graphicsDeviceType() == 2)
    GL.LoadPixelMatrix(0.0f, 1f, 0.0f, 1f);
   else
    GL.LoadPixelMatrix(0.0f, 1f, 1f, 0.0f);
   Graphics.DrawTexture(new Rect(0.0f, 0.0f, 1f, 1f), this.ARCamera.CameraTexture);
   GL.PopMatrix();
  }

  public BackgroundCamera()
  {
   base.\u002Ector();
  }
 }
}
```

Camera Error Codes

```
namespace AR
{
 internal enum CameraErrorCodes
 {
  ManualFocusDistanceIncorrectMode = 11001, // 0x00002AF9
  ManualFocusDistanceAndroidCamera1 = 11002, // 0x00002AFA
  ManualFocusDistanceInvalid = 11003, // 0x00002AFB
  AutoFocusRestrictionOnAndroid = 11004, // 0x00002AFC
  RuntimePropertyAccessedBeforeInitialization = 11005, // 0x00002AFD
  InitializationPropertyAccessedAfterInitialization = 11006, // 0x00002AFE
 }
}
```

AR Error

```
using System;
using System.Runtime.InteropServices;

namespace AR
{
 public class Error
 {
  internal const string CameraErrorDomain = "com.AR.unity.camera";
  internal const string PluginErrorDomain = "com.AR.unity.plugin";

  public int Code { get; private set; }

  public string Domain { get; private set; }

  public string Message { get; private set; }

  public Error UnderlyingError { get; private set; }

  internal Error(int errorCount, IntPtr errorData)
  {
   Error error1 = this;
   for (int index = 0; index < errorCount; ++index)
   {
    BridgeError structure = (BridgeError) Marshal.PtrToStructure(new IntPtr(errorData.ToInt64() + (long) (index * Marshal.SizeOf(typeof (BridgeError)))), typeof (BridgeError));
    error1.Code = structure.Code;
    error1.Domain = structure.Domain;
    error1.Message = structure.Message;
    if (index != errorCount - 1)
    {
     Error error2 = new Error();
     error1.UnderlyingError = error2;
     error1 = error2;
    }
    else
     error1.UnderlyingError = (Error) null;
   }
  }

  internal Error(int errorCode, string errorDomain, string errorMessage)
  {
   this.Code = errorCode;
   this.Domain = errorDomain;
   this.Message = errorMessage;
   this.UnderlyingError = (Error) null;
  }

  private Error()
  {
   this.Code = 0;
   this.Domain = (string) null;
   this.Message = (string) null;
   this.UnderlyingError = (Error) null;
  }
```

```
  }
}
```

Platform Base

```
using UnityEngine;

namespace AR
{
 public abstract class PlatformBase
 {
  protected static PlatformBase _instance;

  public static PlatformBase Instance
  {
   get
   {
    return PlatformBase._instance;
   }
   protected set
   {
    PlatformBase._instance = value;
   }
  }

  public abstract void LoadImage(Texture2D texture, byte[] data);

  public abstract byte[] EncodeToPNG(Texture2D texture);

  public abstract UnityVersion GetUnityVersion();
 }

}
```

Unity AR Bridge

```
using System;
using System.Runtime.InteropServices;
using UnityEngine;

namespace AR
{
 [StaticARBridgeInitializer]
 [StaticPlatformBridgeInitializer]
 internal class UnityARBridge : IARBridge, IPlatformBridge
 {
  internal float[] GenericTrackingMatrices = new float[16]
  {
   0.005390511f,
   0.7855874f,
   0.6187273f,
   0.0f,
   -0.9999313f,
   -0.002205342f,
   0.01151174f,
   0.0f,
   0.01040799f,
   -0.6187468f,
   0.7855215f,
   0.0f,
   0.002648768f,
   0.06339091f,
   -2.233013f,
   1f
  };
  internal float[] ObjectTrackingMatrix = new float[16]
  {
   -0.998f,
   0.041f,
   -0.047f,
   0.0f,
   0.059f,
   0.862f,
   -0.504f,
   0.0f,
   0.02f,
   -0.506f,
   -0.862f,
   0.0f,
   -0.101f,
   -0.427f,
   2.657f,
   1f
  };
  internal float[] PhysicalTargetHeights = new float[0];
  internal long[] TargetIDs = new long[0];
  internal string TrackedTargets = " ";
  internal int TargetCount = 1;
  private string _trackerManagerName = "TrackerManager";
  private bool _isObjectTrackingRunning;

  static UnityARBridge()
  {
   BridgeFactory.RegisterARBridgeConstructor("Unity",
(BridgeFactory.ARBridgeConstructor) (() => (IARBridge) new
UnityARBridge()));
   BridgeFactory.RegisterPlatformBridgeConstructor("Unity",
(BridgeFactory.PlatformBridgeConstructor) (() =>
(IPlatformBridge) new UnityARBridge()));
  }

  void IARBridge.InitializeCSharpFunctions(IntPtr
csharpFunctions)
  {
  }

  void IARBridge.InstantiateARNativeSDK(string licenseKey,
IntPtr platformComponent, IntPtr sdkMutex)
  {
  }

  void IARBridge.DestroyARNativeSDK()
  {
  }

  void IARBridge.StartARNativeSDK()
  {
  }

  void IARBridge.StopARNativeSDK()
  {
  }

  void IARBridge.InstantiateTargetCollectionResource(string
path, long id)
  {
  }

  void IARBridge.InstantiateCloudRecognitionService(string
clientToken, string groupId, string targetCollectionId, int region,
string customServerURL, long id)
  {
  }

  void
IARBridge.InstantiateImageTrackerWithTargetCollectionResou
rce(string targetsForExtendedTracking, string
physicalTargetImageHeights, int
maximumNumberOfConcurrentTrackableTargets, int
rangeExtension, int trackerEfficiencyMode, long
targetCollectionResourceId)
  {
   this._isObjectTrackingRunning = false;
  }

  void
IARBridge.InstantiateImageTrackerWithCloudRecognitionServ
ice(string targetsForExtendedTracking, string
physicalTargetImageHeights, int rangeExtension, int
trackerEfficiencyMode, long cloudRecognitionServiceId)
  {
   this._isObjectTrackingRunning = false;
  }

  void
IARBridge.InstantiateObjectTrackerWithTargetCollectionReso
```

```
urce(string targetsForExtendedTracking, int trackerEfficiencyMode, long targetCollectionResourceId)
    {
      this._isObjectTrackingRunning = true;
    }

    void IARBridge.InstantiateInstantTracker(bool smartEnabled, float deviceHeightAboveGround, float planeOrientationAngle, int trackerEfficiencyMode, UnityPlaneConfiguration planeConfiguration)
    {
      this._isObjectTrackingRunning = false;
    }

    void IARBridge.StopExtendedImageTracking()
    {
    }

    void IARBridge.StopExtendedObjectTracking()
    {
    }

    void IARBridge.DestroyTargetCollectionResource(long id)
    {
    }

    void IARBridge.DestroyCloudRecognitionService(long id)
    {
    }

    void IARBridge.DestroyImageTracker()
    {
    }

    void IARBridge.DestroyInstantTracker()
    {
    }

    void IARBridge.DestroyObjectTracker()
    {
    }

    void IARBridge.Recognize(long id)
    {
    }

    void IARBridge.StartContinuousRecognition(double interval, long id)
    {
    }

    void IARBridge.StopContinuousRecognition(long id)
    {
    }

    void IARBridge.SetState(int state)
    {

GameObject.Find(this._trackerManagerName).SendMessage("InstantTrackerChangedState", (object) state.ToString());
    }

    void IARBridge.SetDeviceHeightAboveGround(float height)
    {
    }

    void IARBridge.SetTrackingPlaneOrientation(float planeOrientationAngle)
    {
    }

    bool IARBridge.CanStartTracking()
    {
      return true;
    }

    void IARBridge.ConvertScreenCoordinate(Vector2 screenCoordinate)
    {

GameObject.Find(this._trackerManagerName).SendMessage("ScreenCoordinateConversionFailed", (object) (screenCoordinate.x.ToString() + "|" + (object) (float) screenCoordinate.y));
    }

    void IARBridge.RequestPointCloud()
    {
      float[] numArray = new float[12]
      {
        0.0f,
        0.0f,
        0.0f,
        1f,
        0.0f,
        0.0f,
        0.0f,
        1f,
        0.0f,
        0.0f,
        0.0f,
        1f
      };
      GCHandle gcHandle = GCHandle.Alloc((object) numArray, GCHandleType.Pinned);
      try
      {
        IntPtr num = gcHandle.AddrOfPinnedObject();

GameObject.Find(this._trackerManagerName).SendMessage("PointCloudRequestFinished", (object) ((numArray.Length / 3).ToString() + "|" + num.ToString()));
      }
      finally
      {
        if (gcHandle.IsAllocated)
          gcHandle.Free();
      }
    }

    void IARBridge.TransferPointCloud(IntPtr source, IntPtr destination, uint length)
    {
      try
      {
        byte[] numArray = new byte[(int) length];
        Marshal.Copy(source, numArray, 0, (int) length);
        Marshal.Copy(numArray, 0, destination, (int) length);
      }
      catch (Exception ex)
      {
        Debug.LogError((object) ex);
      }
    }

    void IARBridge.SaveCurrentInstantTarget(string path)
    {
    }

    void IARBridge.LoadInstantTarget(long resourceId, int expansionPolicy)
    {
    }

    string IARBridge.GetTrackedTargets()
    {
      return this.TrackedTargets;
    }

    long[] IARBridge.GetTargetIDs()
    {
      return this.TargetIDs;
    }

    float IARBridge.GetFov()
    {
      return 57f;
    }
```

```
float IARBridge.GetCameraToSurfaceAngle()
{
  return 0.0f;
}

float[] IARBridge.GetModelViewMatrices()
{
  if (this._isObjectTrackingRunning)
    return this.ObjectTrackingMatrix;
  return this.GenericTrackingMatrices;
}

long[] IARBridge.GetPlaneIDs()
{
  return new long[0];
}

PlaneData IARBridge.GetPlaneData(long identifier)
{
  return new PlaneData();
}

float[] IARBridge.GetPlaneMatrices()
{
  return this.GenericTrackingMatrices;
}

float[] IARBridge.GetPlaneConvexHull(long identifier)
{
  return new float[0];
}

float IARBridge.GetImageTargetScale(string targetName, long targetId, int componentIndex)
{
  return 1f;
}

float IARBridge.GetObjectTargetScale(string targetName, int componentIndex)
{
  return 1f;
}

float IARBridge.GetPhysicalTargetHeight(string targetName, long targetId)
{
  for (int index = 0; index < this.TargetIDs.Length; ++index)
  {
    if (this.TargetIDs[index] == targetId)
      return this.PhysicalTargetHeights[index];
  }
  return 298f;
}

void IARBridge.RegisterPlugin(string identifier, bool hasInputModule, bool requestsCameraFrameRendering)
{
}

void IARBridge.NotifyNewCameraFrame(string identifier, UnityFrame unityFrame, bool invertFrame)
{
}

void IARBridge.SetInputModuleCameraToSurfaceAngle(string identifier, float cameraToSurfaceAngle)
{
}

void IARBridge.DeregisterPlugin(string identifier)
{
}

string IARBridge.GetSDKBuildConfiguration()
{
  return "release";
}

string IARBridge.GetSDKBuildNumber()
{
  return "local";
}

string IARBridge.GetSDKBuildDate()
{
  return "Fri Oct 14 09:22:54 2016";
}

float IARBridge.ComputeDistanceBetweenTargets(string firstImageTargetName, long firstImageTargetID, string secondImageTargetName, long secondImageTargetID)
{
  return 0.0f;
}

float IARBridge.ComputeCameraDistanceToTarget(string imageTargetName, long imageTargetID)
{
  return 0.0f;
}

long IARBridge.GetRenderFrameId()
{
  return -1;
}

bool IPlatformBridge.IsUsingRemote
{
  get
  {
    return false;
  }
  set
  {
  }
}

bool IPlatformBridge.IsEditor
{
  get
  {
    return true;
  }
}

void IPlatformBridge.InitializeCSharpFunctions(IntPtr csharpFunctions)
{
}

IntPtr IPlatformBridge.InstantiatePlatform(string temporaryDirectory)
{
  return IntPtr.Zero;
}

IntPtr IPlatformBridge.GetSDKMutex()
{
  return IntPtr.Zero;
}

void IPlatformBridge.SetLicenseKey(string licenseKey)
{
}

void IPlatformBridge.Start()
{
}

void IPlatformBridge.DestroyPlatform()
{
}

Texture IPlatformBridge.GetRenderTexture()
{
```

```
    return (Texture) Texture2D.get_blackTexture();
  }

  void IPlatformBridge.UpdateRenderTexture(Texture texture)
  {
  }

  void IPlatformBridge.SurfaceChanged(uint width, uint height)
  {
  }

  void IPlatformBridge.CameraToSurfaceAngleChanged(float
angle)
  {
  }

  void IPlatformBridge.SetMetadata(float cameraFieldOfView,
uint frameWidth, uint frameHeight)
  {
  }

  void IPlatformBridge.OnPreRender()
  {
  }

  void IPlatformBridge.Update()
  {
  }

  void IPlatformBridge.UpdateOrientation(ScreenOrientation
orientation)
  {
  }

  void IPlatformBridge.NotifyNewInputFrame(ulong frameId,
IntPtr frameData)
  {
  }

  void IPlatformBridge.NotifyNewDeviceRotationEvent(float[]
rotation)
  {
  }

  void
IPlatformBridge.IsPlatformAssistedTrackingSupported(bool
internalCall)
  {

TrackerManager.IsPlatformAssistedTrackingAvailableInternal(
SmartAvailability.Unsupported, internalCall);
  }

  void IPlatformBridge.SetCaptureDeviceResolution(int
resolutionMode)
  {
  }

  void IPlatformBridge.SetCaptureDeviceFramerate(int
framerateMode)
  {
  }

  int IPlatformBridge.GetCaptureDevicePosition()
  {
    return 0;
  }

  void IPlatformBridge.SetCaptureDevicePosition(int
newDevicePosition)
  {
  }

  bool IPlatformBridge.IsCaptureFocusModeSupported(int
focusMode)
  {
    return false;
  }

  int IPlatformBridge.GetCaptureFocusMode()
  {
    return 0;
  }

  void IPlatformBridge.SetCaptureFocusMode(int focusMode)
  {
  }

  bool IPlatformBridge.IsCaptureExposureModeSupported(int
focusMode)
  {
    return false;
  }

  int IPlatformBridge.GetCaptureExposureMode()
  {
    return 0;
  }

  void IPlatformBridge.SetCaptureExposureMode(int
exposureMode)
  {
  }

  bool IPlatformBridge.IsManualFocusAvailable()
  {
    return false;
  }

  float IPlatformBridge.GetManualFocusDistance()
  {
    return 0.0f;
  }

  void IPlatformBridge.SetManualFocusDistance(float
manualFocusDistance)
  {
  }

  bool
IPlatformBridge.IsCaptureAutoFocusRestrictionSupported()
  {
    return false;
  }

  int IPlatformBridge.GetCaptureAutoFocusRestriction()
  {
    return 0;
  }

  void IPlatformBridge.SetCaptureAutoFocusRestriction(int
newAutoFocusRestriction)
  {
  }

  bool IPlatformBridge.IsCaptureZoomLevelSupported(float
zoomLevel)
  {
    return false;
  }

  float IPlatformBridge.GetCaptureZoomLevel()
  {
    return 0.0f;
  }

  void IPlatformBridge.SetCaptureZoomLevel(float
newZoomLevel)
  {
  }

  float IPlatformBridge.GetCaptureMaxZoomLevel()
  {
    return 1f;
  }

  bool IPlatformBridge.IsCaptureFlashModeSupported(int
flashMode)
```

```
    {
      return false;
    }

    int IPlatformBridge.GetCaptureFlashMode()
    {
      return 0;
    }

    void IPlatformBridge.SetCaptureFlashMode(int newFlashMode)
    {
    }

    bool IPlatformBridge.IsFocusAtPointOfInterestSupported()
    {
      return false;
    }

    void IPlatformBridge.FocusAtPointOfInterest(float pointOfInterestX, float pointOfInterestY)
    {
    }

    bool IPlatformBridge.IsExposeAtPointOfInterestSupported()
    {
      return false;
    }

    void IPlatformBridge.ExposeAtPointOfInterest(float pointOfInterestX, float pointOfInterestY, int exposureMode)
    {
    }

    void IPlatformBridge.EnableCamera2(bool enabled)
    {
    }
  }
}
```

Assembly Info

```
using System.Reflection;
using System.Runtime.CompilerServices;

[assembly: AssemblyTitle("ARUnityPlugin")]
[assembly: AssemblyDescription("Unity Plugin for Augmented Reality AR SDK")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("AR GmbH")]
[assembly: AssemblyProduct("Unity Plugin for Augmented Reality AR SDK")]
[assembly: AssemblyCopyright("© 2015-2017 AR GmbH")]
[assembly: AssemblyTrademark("")]
[assembly: InternalsVisibleTo("ARUnityEditor")]
[assembly: InternalsVisibleTo("UWPBridge")]
[assembly: InternalsVisibleTo("MacOSEditorBridge")]
[assembly: InternalsVisibleTo("WindowsEditorBridge")]
[assembly: AssemblyVersion("8.1.0.0")]
```

Dock UI

```
using UnityEngine;

public class DockUI : MonoBehaviour
{
    public RectTransform ButtonsBackground;
    public RectTransform ButtonsParent;

    private void Awake() {
        Update();
    }

    private void Update() {
        float currentRatio = (float)Screen.width / (float)Screen.height;
        if (currentRatio > 1.0f) {
            ButtonsBackground.sizeDelta = new Vector2(0f, -200f);
            ButtonsParent.sizeDelta = new Vector2(-600f, 0f);
        } else {
            ButtonsBackground.sizeDelta = new Vector2(0f, -206f);
            ButtonsParent.sizeDelta = new Vector2(0f, 0f);
        }
    }
}
```

Grid Renderer

```
using UnityEngine;

public class GridRenderer : MonoBehaviour
{
    public static Color DefaultTargetColor = new Color(1.0f, 0.525f, 0, 1.0f);
    private static Color GridColor = new Color(1.0f, 1.0f, 1.0f, 1.0f);
    private static Color MainLineColor = GridColor * 0.9f;
    private static Color UnitLineColor = GridColor * 0.75f;

    public Color TargetColor = DefaultTargetColor;

    private const int NumberOfLinesOnEitherSide = 50;
    private const float LineSpacing = 0.25f;
    private const int IntervalBetweenMainLines = 10;
    private const float TargetSize = 8.0f;

    private Material _lineMaterial;
    private Plane[] _cameraPlanes;

    private void Start() {
        InitializeLineMaterial();
    }

    private void OnRenderObject() {
        // Because the AR SDK uses a secondary camera to render the camera frame, we need to make sure to render the grid only on the main camera.
        if (Camera.current == Camera.main) {
            RenderGrid();
        }
    }

    private void InitializeLineMaterial() {
        var shader = Shader.Find("Hidden/Internal-Colored");
        _lineMaterial = new Material(shader);
        _lineMaterial.hideFlags = HideFlags.HideAndDontSave;

        _lineMaterial.SetInt("_SrcBlend", (int)UnityEngine.Rendering.BlendMode.SrcAlpha);
        _lineMaterial.SetInt("_DstBlend", (int)UnityEngine.Rendering.BlendMode.OneMinusSrcAlpha);

        _lineMaterial.SetInt("_Cull", (int)UnityEngine.Rendering.CullMode.Off);
        _lineMaterial.SetInt("_ZWrite", 0);
    }

    private void RenderGrid() {
        _lineMaterial.SetPass(0);
        _cameraPlanes = GeometryUtility.CalculateFrustumPlanes(Camera.main);

        GL.Begin(GL.LINES);

        RenderGrid(Vector3.forward, Vector3.right);
        RenderGrid(Vector3.right, Vector3.forward);

        RenderLine(Vector3.forward, Vector3.zero, TargetColor, TargetSize);
        RenderLine(Vector3.right, Vector3.zero, TargetColor, TargetSize);

        GL.End();
    }

    private void RenderGrid(Vector3 mainAxis, Vector3 perpendicular) {
        for (int i = -NumberOfLinesOnEitherSide; i <= NumberOfLinesOnEitherSide; ++i) {
            var color = UnitLineColor;
```

```
            if (i == 0) {
                color = GridColor;
            } else if (i % IntervalBetweenMainLines == 0) {
                color = MainLineColor;
            }
            float intensity = 1.0f - Mathf.Abs(i * 2.0f) / NumberOfLinesOnEitherSide;
            color.a *= intensity;
            if (color.a > 0.01f) {
                RenderLine(mainAxis, perpendicular * i * LineSpacing, color, (float)NumberOfLinesOnEitherSide / 8.0f);
            }
        }
    }

    private void RenderLine(Vector3 axis, Vector3 offset, Color color, float length) {
        RenderLine(GetTransparent(color), axis * (-length) + offset, color, offset);
        RenderLine(color, offset, GetTransparent(color), axis * length + offset);
    }

    private void RenderLine(Color startColor, Vector3 startPosition, Color endColor, Vector3 endPosition) {
        Color clippedEndColor = Color.Lerp(startColor, endColor, Clip(startPosition, ref endPosition));
        Color clippedStartColor = Color.Lerp(startColor, endColor, 1.0f - Clip(endPosition, ref startPosition));

        GL.Color(clippedStartColor);
        GL.Vertex(startPosition);
        GL.Color(clippedEndColor);
        GL.Vertex(endPosition);
    }

    private float Clip(Vector3 start, ref Vector3 end) {
        // We clip the vertices to the camera planes because on some Android devices clipping of GL_LINES doesn't work correctly.
        Vector3 v = end - start;
        float magnitude = v.magnitude;
        Vector3 vNorm = v / magnitude;

        var startRay = new Ray(start, vNorm);
        float minDistance = magnitude;

        foreach (var plane in _cameraPlanes) {
            if (Vector3.Dot(vNorm, plane.normal) < 0) {
                float enter;
                if (plane.Raycast(startRay, out enter)) {
                    if (enter < minDistance) {
                        minDistance = enter;
                    }
                }
            }
        }

        end = startRay.GetPoint(minDistance);
        return minDistance / magnitude;
    }

    private Color GetTransparent(Color color) {
        color.a = 0.0f;
        return color;
    }
}
```

Instant Tracking Controller

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;
using System.Collections.Generic;
using AR;
using System;

using Plane = UnityEngine.Plane;

public class InstantTrackingController : SampleController
{
    public GameObject ButtonDock;
    public GameObject InitializationControls;
    public Text HeightLabel;

    public InstantTracker Tracker;

    public Button ResetButton;

    public List<Button> Buttons;
    public List<GameObject> Models;

    public Text MessageBox;

    public Image ActivityIndicator;
    public Image InitializationButton;
    public Sprite InitializationOn;
    public Sprite InitializationOff;

    public Color EnabledColor = new Color(0.2f, 0.75f, 0.2f, 0.8f);
    public Color DisabledColor = new Color(1.0f, 0.2f, 0.2f, 0.8f);

    private MoveController _moveController;
    private GridRenderer _gridRenderer;

    private HashSet<GameObject> _activeModels = new HashSet<GameObject>();
    private InstantTrackingState _currentState = InstantTrackingState.Initializing;
    public InstantTrackingState CurrentState {
        get { return _currentState; }
    }
    private bool _isTracking = false;

    public HashSet<GameObject> ActiveModels {
        get {
            return _activeModels;
        }
    }

    private void Awake() {
        Application.targetFrameRate = 60;

        _moveController = GetComponent<MoveController>();
        _gridRenderer = GetComponent<GridRenderer>();
    }

    protected override void Start() {
        base.Start();
        QualitySettings.shadowDistance = 4.0f;


        MessageBox.text = "Starting the SDK";
        // The AR SDK needs to be fully started before we can query for platform assisted tracking support
        // SDK initialization happens during start, so we wait one frame in a coroutine
        StartCoroutine(CheckPlatformAssistedTrackingSupport());
    }

    private IEnumerator CheckPlatformAssistedTrackingSupport() {
        yield return null;
        if (Tracker.SMARTEnabled) {

            Tracker.IsPlatformAssistedTrackingSupported((SmartAvailability smartAvailability) => {
                UpdateTrackingMessage(smartAvailability);
            });
        }
    }

    private void UpdateTrackingMessage(SmartAvailability smartAvailability) {
        if (Tracker.SMARTEnabled) {
            string sdk;
```

```
        if (Application.platform == RuntimePlatform.Android)
{
            sdk = "ARCore";
        }       else        if        (Application.platform        ==
RuntimePlatform.IPhonePlayer) {
            sdk = "ARKit";
        } else {
            return;
        }

        switch (smartAvailability) {
            case SmartAvailability.IndeterminateQueryFailed: {
                MessageBox.text = "Platform support query failed.
Running without platform assisted tracking support.";
                break;
            }
            case SmartAvailability.CheckingQueryOngoing: {
                MessageBox.text = "Platform support query
ongoing.";
                break;
            }
            case SmartAvailability.Unsupported: {
                MessageBox.text = "Running without platform
assisted tracking support.";
                break;
            }
            case SmartAvailability.SupportedUpdateRequired:
            case SmartAvailability.Supported: {
                string runningWithMessage = "Running with
platform assisted tracking support (" + sdk + ").";

                if (_currentState == InstantTrackingState.Tracking)
{
                    MessageBox.text = runningWithMessage;
                } else {
                    MessageBox.text = runningWithMessage + "\n
Move your phone around until the target turns green, which is
when you can start tracking.";
                }
                break;
            }
        }
    } else {
        MessageBox.text = "Running without platform assisted
tracking support.";
    }
  }

  protected override void Update() {
    base.Update();
    if (_currentState == InstantTrackingState.Initializing) {
      if (Tracker.CanStartTracking()) {
        _gridRenderer.TargetColor = Color.green;
        InitializationButton.GetComponent<Image>().sprite =
InitializationOn;
      } else {
        _gridRenderer.TargetColor                          =
GridRenderer.DefaultTargetColor;
        InitializationButton.GetComponent<Image>().sprite =
InitializationOff;
      }
    } else {
      _gridRenderer.TargetColor                            =
GridRenderer.DefaultTargetColor;
      InitializationButton.GetComponent<Image>().sprite    =
InitializationOff;
    }
  }

  #region UI Events
  public void OnInitializeButtonClicked() {
    Tracker.SetState(InstantTrackingState.Tracking);
  }

  public void OnHeightValueChanged(float newHeightValue)
{
    HeightLabel.text = string.Format("{0:0.##} m",
newHeightValue);
    Tracker.DeviceHeightAboveGround = newHeightValue;
  }

  public void OnBeginDrag (int modelIndex) {
    if (_isTracking) {
      // Create object
      GameObject modelPrefab = Models[modelIndex];
      Transform model = Instantiate(modelPrefab).transform;
      _activeModels.Add(model.gameObject);
      // Set model position at touch position
      var                cameraRay                =
Camera.main.ScreenPointToRay(Input.mousePosition);
      Plane p = new Plane(Vector3.up, Vector3.zero);
      float enter;
      if (p.Raycast(cameraRay, out enter)) {
        model.position = cameraRay.GetPoint(enter);
      }

      // Set model orientation to face toward the camera
      Quaternion          modelRotation          =
Quaternion.LookRotation(Vector3.ProjectOnPlane(-
Camera.main.transform.forward, Vector3.up), Vector3.up);
      model.rotation = modelRotation;

      _moveController.SetMoveObject(model);
    }
  }

  public void OnResetButtonClicked() {
    Tracker.SetState(InstantTrackingState.Initializing);
    ResetButton.gameObject.SetActive(false);
  }
  #endregion

  #region Tracker Events
  public void OnSceneRecognized(InstantTarget target) {
    SetSceneActive(true);
  }

  public void OnSceneLost(InstantTarget target) {
    SetSceneActive(false);
  }

  private void SetSceneActive(bool active) {
    foreach (var button in Buttons) {
      button.interactable = active;
    }

    foreach (var model in _activeModels) {
      model.SetActive(active);
    }

    ActivityIndicator.color = active ? EnabledColor :
DisabledColor;

    _gridRenderer.enabled = active;
    _isTracking = active;
  }

  public void OnStateChanged(InstantTrackingState newState)
{
    _currentState = newState;
    if (newState == InstantTrackingState.Tracking) {
      if (InitializationControls != null) {
        InitializationControls.SetActive(false);
      }
      ButtonDock.SetActive(true);
      ResetButton.gameObject.SetActive(true);
    } else {
      foreach (var model in _activeModels) {
        Destroy(model);
      }
      _activeModels.Clear();

      if (InitializationControls != null) {
        InitializationControls.SetActive(true);
      }
      ButtonDock.SetActive(false);
    }
  }
```

```
    /// <summary>
    /// Used when augmentations are loaded from disk.
    /// Please see SaveInstantTarget and LoadInstantTarget for more information.
    /// </summary>
    internal void LoadAugmentation(AugmentationDescription augmentation) {
        GameObject modelPrefab = Models[augmentation.ID];
        Transform model = Instantiate(modelPrefab).transform;
        _activeModels.Add(model.gameObject);

        model.localPosition = augmentation.LocalPosition;
        model.localRotation = augmentation.LocalRotation;
        model.localScale = augmentation.LocalScale;

        model.gameObject.SetActive(false);
    }

    public void OnError(Error error) {
        PrintError("Instant Tracker error!", error);
    }
    #endregion
}
```

Load Instant Target

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using UnityEngine;
using UnityEngine.UI;
using AR;

public class LoadInstantTarget : MonoBehaviour {
    public InstantTracker Tracker;
    public InstantTrackingController Controller;
    public Text InfoMessage;
    public Button LoadButton;

    public void OnLoadButtonPressed() {
        LoadTarget();
        LoadScene();
    }

    public void OnChangedState(InstantTrackingState state) {
        if (state == InstantTrackingState.Tracking) {
            LoadButton.gameObject.SetActive(false);
        } else {
            LoadButton.gameObject.SetActive(true);
        }
    }

    /// <summary>
    /// Loads the instant target from the disk, without any augmentations.
    /// </summary>
    private void LoadTarget() {
        // A TargetCollectionResource is needed to manage file loading.
        var targetCollectionResource = new TargetCollectionResource();
        // UseCustomURL is used to specify that the file is not inside the "StreamingAssets" folder
        targetCollectionResource.UseCustomURL = true;
        // The "file://" is used to indicate that the file is located on disk, and not on a server.
        targetCollectionResource.TargetPath = "file://" + Application.persistentDataPath + "/InstantTarget.wto";
        var configuration = new InstantTargetRestorationConfiguration();
        configuration.ExpansionPolicy = InstantTargetExpansionPolicy.Allow;
        Tracker.LoadInstantTarget(targetCollectionResource, configuration, LoadSuccessHandler, LoadErrorHandler);
    }

    /// <summary>
    /// Loads all augmentations from disk.
    /// </summary>
    private void LoadScene() {
        try {
            string json = File.ReadAllText(Application.persistentDataPath + "/InstantScene.json");
            var sceneDescription = JsonUtility.FromJson<SceneDescription>(json);

            foreach (var augmentation in sceneDescription.Augmentations) {
                Controller.LoadAugmentation(augmentation);
            }
        } catch (Exception ex) {
            InfoMessage.text = "Error loading scene augmentations.";
            Debug.LogError("Error loading augmentations: " + ex.Message);
        }
    }

    private void LoadSuccessHandler(string path) {
        InfoMessage.text = "The instant target was successfully loaded from path: " + path;
    }

    private void LoadErrorHandler(Error error) {
        InfoMessage.text = "The following error occurred when loading the instant target. " +
            "Error code: " + error.Code + " domain: " + error.Domain + " message: " + error.Message;
    }
}
```

Move Controller

```
using UnityEngine;

public class MoveController : MonoBehaviour
{
    private Transform _activeObject = null;

    private Vector3 _startObjectPosition;
    private Vector2 _startTouchPosition;
    private Vector2 _touchOffset;

    private InstantTrackingController _controller;

    private void Start() {
        _controller = GetComponent<InstantTrackingController>();
    }

    public Transform ActiveObject {
        get {
            return _activeObject;
        }
    }

    public void SetMoveObject(Transform newMoveObject) {
        if (_controller.ActiveModels.Contains(newMoveObject.gameObject)) {
            _activeObject = newMoveObject;
            _startObjectPosition = _activeObject.position;
            _startTouchPosition = Input.GetTouch(0).position;
            _touchOffset = Camera.main.WorldToScreenPoint(_startObjectPosition);
        }
    }

    void Update () {
        if (Input.touchCount > 0) {
            Touch touch = Input.GetTouch(0);
            RaycastHit hit;

            if (_activeObject == null) {
                if (Physics.Raycast(Camera.main.ScreenPointToRay(touch.position), out hit)) {
```

```
            SetMoveObject(hit.transform);
          }
        }

        if (_activeObject != null) {
          var screenPosForRay = (touch.position -
_startTouchPosition) + _touchOffset;
          Ray cameraRay =
Camera.main.ScreenPointToRay(screenPosForRay);
          Plane p = new Plane(Vector3.up, Vector3.zero);

          float enter;
          if (p.Raycast(cameraRay, out enter)) {
            var position = cameraRay.GetPoint(enter);

            position.x = Mathf.Clamp(position.x, -15.0f, 15.0f);
            position.y = 0.0f;
            position.z = Mathf.Clamp(position.z, -15.0f, 15.0f);

            _activeObject.position =
Vector3.Lerp(_activeObject.position, position, 0.25f);
          }
        }
      } else {
        _activeObject = null;
      }
    }
}
```

Plane Renderer

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using AR;

public class PlaneRenderer : MonoBehaviour {

  /* How thick the faded border should be. A positive value
fades outside the existing bounds, while a negative value fades
inside the bounds. */
  public float FadeDistance = 0.1f;
  public Material RenderPlaneMaterial;

  private Dictionary<long, GameObject> _renderPlanes = new
Dictionary<long, GameObject>();

  public void OnPlaneRecognized(AR.Plane recognizedPlane)
{
    var renderPlane = new GameObject("Plane");

renderPlane.transform.SetParent(recognizedPlane.Drawable.tra
nsform);

    var mesh = new Mesh();
    renderPlane.AddComponent<MeshFilter>().mesh = mesh;
    renderPlane.AddComponent<MeshRenderer>().material =
new Material(RenderPlaneMaterial);

    UpdateMesh(renderPlane, mesh, recognizedPlane);

    _renderPlanes.Add(recognizedPlane.ID, renderPlane);
  }

  public void OnPlaneTracked(AR.Plane trackedPlane) {
    GameObject renderPlane;
    if (_renderPlanes.TryGetValue(trackedPlane.ID, out
renderPlane)) {
      UpdateMesh(renderPlane,
renderPlane.GetComponent<MeshFilter>().sharedMesh,
trackedPlane);
    } else {
      Debug.LogError("Could not find tracked plane with ID:
" + trackedPlane.ID);
    }
  }

  public void OnPlaneLost(AR.Plane lostPlane) {
    GameObject renderPlane;
    if (_renderPlanes.TryGetValue(lostPlane.ID, out
renderPlane)) {
      _renderPlanes.Remove(lostPlane.ID);
      Destroy(renderPlane);
    } else {
      Debug.LogError("Could not find lost plane with ID: " +
lostPlane.ID);
    }
  }

  private void UpdateMesh(GameObject renderPlane, Mesh
mesh, AR.Plane plane) {
    int vertexCount = plane.ConvexHull.Length / 2;
    List<Vector3> convexHull = new
List<Vector3>(vertexCount);
    for (int i = 0; i < vertexCount; ++i) {
      convexHull.Add(new Vector3(plane.ConvexHull[i * 2],
0.0f, plane.ConvexHull[i * 2 + 1]));
    }

    var inset = ComputeInset(convexHull);

    /* The sign of the fadeDistance decides which polygon is
outside, and which one is inside. */
    if (FadeDistance > 0) {
      UpdateMeshGeometry(mesh, convexHull, inset);
    } else {
      UpdateMeshGeometry(mesh, inset, convexHull);
    }

    Color32 color = Color.white;
    switch (plane.PlaneType) {
      case PlaneType.HorizontalUpward:
      case PlaneType.HorizontalDownward: {
        color = new Color32(0, 0, 255, 102);
        break;
      }
      case PlaneType.Vertical: {
        color = new Color32(237, 33, 200, 102);
        break;
      }
      case PlaneType.Arbitrary: {
        color = new Color32(64, 237, 33, 102);
        break;
      }
    }


renderPlane.GetComponent<MeshRenderer>().sharedMaterial.
SetColor("_Color", color);
  }

  private void UpdateMeshGeometry(Mesh mesh,
List<Vector3> insidePolygon, List<Vector3> outsidePolygon)
{
    mesh.Clear();
    var vertices = new List<Vector3>();
    var indices = new List<int>();
    var uvs = new List<Vector2>();
    var colors = new List<Color32>();

    /* Add the points from the inside polygon. These points
have full opacity. */
    for (int i = 0; i < insidePolygon.Count; ++i) {
      vertices.Add(insidePolygon[i]);
      uvs.Add(new Vector2(insidePolygon[i].x,
insidePolygon[i].z));
      colors.Add(Color.white);
    }

    /* Add the points from the outside polygon. These points
have zero opacity to provide the fade out effect. */
    for (int i = 0; i < outsidePolygon.Count; ++i) {
      vertices.Add(outsidePolygon[i]);
      uvs.Add(new Vector2(outsidePolygon[i].x,
outsidePolygon[i].z));
      colors.Add(new Color(1.0f, 1.0f, 1.0f, 0.0f));
    }
```

```
        /* Add the indices for the convex hull, which is rendered as a triangle fan. */
        for (int i = 2; i < insidePolygon.Count; ++i) {
            indices.Add(0);
            indices.Add(i - 1);
            indices.Add(i);
        }

        /* Add the indices for the faded border, which is rendered as a triangle strip, using points from both the outside polygon, as well as from the inside polygon. */
        for (int i = 0; i < outsidePolygon.Count; ++i) {
            indices.Add(i + insidePolygon.Count);
            indices.Add((i + 1) % outsidePolygon.Count + insidePolygon.Count);
            indices.Add(i);

            indices.Add(i);
            indices.Add((i + 1) % outsidePolygon.Count + insidePolygon.Count);
            indices.Add((i + 1) % insidePolygon.Count);
        }

        mesh.SetVertices(vertices);
        mesh.SetIndices(indices.ToArray(), MeshTopology.Triangles, 0);
        mesh.SetUVs(0, uvs);
        mesh.SetColors(colors);
    }

    /* Computes an inset for the polygon described by convexHull.
    * A positive fadeDistance results in an inset that is outside the original convexHull.
    * A negative fadeDistance results in an inset that is inside the original convexHull.
    * It assumes that the points in the convexHull are arranged in a counter-clockwise fashion.
    *
    * How it works is that for each point, it takes each connecting line adjacent to it and translates them perpendicularly by fadeDistance.
    * The new point is found at the intersection of these lines.
    *
    *          intersection
    *            point
    *              |                        +------- prevOffsetDir
    *              |                        |
    *              v                        v
    *              * - - - - ---------------------------
    *              .                  ^
    *              .                  |
    *              .              positive inset
    *              .                distance
    *              .                  |
    *              .                  v
    *              .            *-------------------------
    *              /            / ^                 ^
    *             /            /  |                 |
    *            /            /   original          +------- prevDir
    *           /            /    point (currPoint)
    *          /            /
    *         /            /
    *        /            / <---- nextDir
    *       /            /
    *
    */
    private List<Vector3> ComputeInset(List<Vector3> convexHull) {
        var inset = new List<Vector3>();

        for (int i = 0; i < convexHull.Count; ++i) {

            /* For each point, we need to also consider the points adjacent to it. */
            var prevPoint = convexHull[(convexHull.Count + i - 1) % convexHull.Count];
            var currPoint = convexHull[i];
            var nextPoint = convexHull[(i + 1) % convexHull.Count];

            /* Compute the vectors describing the lines between the current point and the adjacent points. */
            var prevDir = currPoint - prevPoint;
            var nextDir = nextPoint - currPoint;

            /* Compute the perpendicular vectors to the lines computed above.
            * This assumes the points are arranged in a counter-clockwise fashion. If that is not the case, the sign needs to be inverted!
            */
            var prevPerp = new Vector3(-prevDir.z, 0.0f, prevDir.x).normalized;
            var nextPerp = new Vector3(-nextDir.z, 0.0f, nextDir.x).normalized;

            /* Move each point in the perpendicular direction we computed before and
            * connect them. This creates two new lines (*OffsetDir) which correspond to the original
            * lines computed above, but offset by fadeDistance.
            */
            var prevOffsetStart = prevPoint + prevPerp * FadeDistance;
            var prevOffsetEnd = currPoint + prevPerp * FadeDistance;

            var nextOffsetStart = currPoint + nextPerp * FadeDistance;
            var nextOffsetEnd = nextPoint + nextPerp * FadeDistance;

            var prevOffsetDir = prevOffsetEnd - prevOffsetStart;
            var nextOffsetDir = nextOffsetEnd - nextOffsetStart;

            /* Compute the intersection point.
            * When using the cross product, only the y component is considered. Because all points are on the XZ plane,
            * the cross product will either point directly up or directly down.
            */
            Vector3 intersection;
            float cross = Vector3.Cross(prevOffsetDir, nextOffsetDir).y;
            if (Mathf.Approximately(cross, 0.0f)) {
                /* This means that the lines are parallel to each other, so the intersection point is just one of the points
                * on the offset line. */
                intersection = prevOffsetEnd;
            } else {
                float a = Vector3.Cross(nextOffsetEnd - prevOffsetStart, nextOffsetDir).y / cross;
                intersection = prevOffsetStart + a * prevOffsetDir;
            }
            inset.Add(intersection);
        }

        return inset;
    }
}
```

Save Instant Target

```
using System;
// using System.Collections;
// using System.Collections.Generic;
// using System.IO;
// using UnityEngine;
// using UnityEngine.UI;
// using AR;

// public class SaveInstantTarget : MonoBehaviour {
    // public InstantTracker Tracker;
    // public InstantTrackingController Controller;
    // public Text InfoMessage;
    // public Button SaveButton;
```

```
    // public void OnChangedState(InstantTrackingState state) {
        Only enable the save button when tracking
        // if (state == InstantTrackingState.Tracking) {
            // SaveButton.interactable = true;
        // } else {
            // SaveButton.interactable = false;
        // }
    // }

    // public void OnSaveButtonPressed() {
        // if (Controller.CurrentState == InstantTrackingState.Tracking) {
            // SaveTarget();
            // SaveScene();
        // }
        // else {
            // InfoMessage.text = "Instant targets can only be saved when the tracker is in the tracking state.";
        // }
    // }

    / <summary>
    / Saves the instant target to disk. This only saves the point cloud representation of the scene,
    / without any augmentations.
    / </summary>
    // private void SaveTarget() {
        // var path = Application.persistentDataPath + "/InstantTarget.wto";
        // Tracker.SaveCurrentInstantTarget(path, SaveSuccessHandler, SaveErrorHandler);
        // InfoMessage.text = "Saving instant target to: " + path;
    // }

    / <summary>
    / Saves all augmentations to disk. Each augmentation is represented by an AugmentationDescription and
    / the entire scene is serialized as a SceneDescription.
    / </summary>
    // private void SaveScene() {
        // var sceneDescription = new SceneDescription();

        // foreach (var augmentation in Controller.ActiveModels) {
            // int id = GetAugmentationId(augmentation);
            // if (id == -1) {
                Early return in case we cannot find the ID of an augmentation.
                // Debug.LogError("Could not find ID for augmentation " + augmentation.name);
                // return;
            // } else {
                // sceneDescription.Augmentations.Add(new AugmentationDescription(id, augmentation.transform));
            // }
        // }

        // try {
            // string json = JsonUtility.ToJson(sceneDescription);
            // File.WriteAllText(Application.persistentDataPath + "/InstantScene.json", json);
        // } catch (Exception ex) {
            // InfoMessage.text = "Error saving scene augmentations.";
            // Debug.LogError("Error saving augmentations: " + ex.Message);
        // }
    // }

    // private void SaveSuccessHandler(string path) {
        // InfoMessage.text = "The instant target was successfully saved at path: " + path;
    // }

    // private void SaveErrorHandler(Error error) {
        // InfoMessage.text = "The following error occurred when saving the instant target. " +
            // "Error code: " + error.Code + " domain: " + error.Domain + " message: " + error.Message;
    // }

    / <summary>
    / Utility method that searches for the ID of an augmentation.
    / InstantTrackingController.Models is a list of all the augmentation that can be added to the scene.
    / The ID is simply the index in this list.
    / </summary>
    / <param name="augmentation">The GameObject for which the ID is required.</param>
    / <returns>Index of the augmentation in InstantTrackingController.Models. Returns -1 if not found.</returns>
    // private int GetAugmentationId(GameObject augmentation) {
        // for (int i = 0; i < Controller.Models.Count; ++i) {
            Because instantiated prefabs have the "(Clone)" suffix applied, we only check if the names
            start the same way.
            // if (augmentation.name.StartsWith(Controller.Models[i].name)) {
                // return i;
            // }
        // }
        // return -1;
    // }
// }
```

Scale Controller

```
using UnityEngine;

public class ScaleController : MonoBehaviour
{
    public float MinScale = 0.1f;
    public float MaxScale = 0.5f;

    private InstantTrackingController _controller;
    private Transform _activeObject = null;

    private Vector3 _touch1StartGroundPosition;
    private Vector3 _touch2StartGroundPosition;
    private Vector3 _startObjectScale;

    private void Start () {
        _controller = GetComponent<InstantTrackingController>();
    }

    private void Update () {
        if (Input.touchCount >= 2) {
            Touch touch1 = Input.GetTouch(0);
            Touch touch2 = Input.GetTouch(1);
            Transform hitTransform;

            if (_activeObject == null) {
                if (GetTouchObject(touch1.position, out hitTransform)) {
                    SetTouchObject(hitTransform);
                } else if (GetTouchObject(touch2.position, out hitTransform)) {
                    SetTouchObject(hitTransform);
                } else if (GetTouchObject((touch1.position + touch2.position) / 2, out hitTransform)) {
                    SetTouchObject(hitTransform);
                }

                if (_activeObject != null) {
                    _touch1StartGroundPosition = GetGroundPosition(touch1.position);
                    _touch2StartGroundPosition = GetGroundPosition(touch2.position);
                    _startObjectScale = _activeObject.localScale;
                }
            }

            if (_activeObject != null) {
                var touch1GroundPosition = GetGroundPosition(touch1.position);
                var touch2GroundPosition = GetGroundPosition(touch2.position);
```

```
            float startMagnitude = (_touch1StartGroundPosition -
_touch2StartGroundPosition).magnitude;
            float currentMagnitude = (touch1GroundPosition -
touch2GroundPosition).magnitude;

            _activeObject.localScale = _startObjectScale *
(currentMagnitude / startMagnitude);

            if (_activeObject.localScale.x < MinScale) {
               _activeObject.localScale = new Vector3(MinScale,
MinScale, MinScale);
            }

            if (_activeObject.localScale.x > MaxScale) {
               _activeObject.localScale = new Vector3(MaxScale,
MaxScale, MaxScale);
            }
         }
      } else {
         _activeObject = null;
      }
   }

   private bool GetTouchObject(Vector2 touchPosition, out
Transform hitTransform) {
      var                touchRay                =
Camera.main.ScreenPointToRay(touchPosition);
      touchRay.origin -= touchRay.direction * 100.0f;

      RaycastHit hit;
      if (Physics.Raycast(touchRay, out hit)) {
         hitTransform = hit.transform;
         return true;
      }

      hitTransform = null;
      return false;
   }

   private Vector3 GetGroundPosition(Vector2 touchPosition) {
      var groundPlane = new Plane(Vector3.up, Vector3.zero);
      var                touchRay                =
Camera.main.ScreenPointToRay(touchPosition);
      float enter;
      if (groundPlane.Raycast(touchRay, out enter)) {
         return touchRay.GetPoint(enter);
      }
      return Vector3.zero;
   }

   private void SetTouchObject(Transform newObject) {
      if
(_controller.ActiveModels.Contains(newObject.gameObject)) {
         _activeObject = newObject;
      }
   }
}
```

Scene Description

```
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// Simple struct used to serialize augmentations to disk.
/// </summary>
[System.Serializable]
public struct AugmentationDescription {
   /// <summary>
   /// Corresponds to the index of this augmentation in
InstantTrackingController.Models
   /// </summary>
   public int ID;
   public Vector3 LocalPosition;
   public Quaternion LocalRotation;
   public Vector3 LocalScale;

   public AugmentationDescription(int id, Transform transform)
{
      ID = id;
      LocalPosition = transform.localPosition;
      LocalRotation = transform.localRotation;
      LocalScale = transform.localScale;
   }
}

[System.Serializable]
public class SceneDescription {
   public List<AugmentationDescription> Augmentations =
new List<AugmentationDescription>();
}
```

Scene Picking Controller

```
using UnityEngine;
using UnityEngine.UI;
using AR;
using System.Collections;
using System.Collections.Generic;

public class ScenePickingController : SampleController
{
   public InstantTracker Tracker;
   public GameObject Augmentation;

   public Button ToggleStateButton;
   public Text ToggleStateButtonText;
   public Text MessageBox;

   public GameObject HeightSlider;
   public Text HeightLabel;

   private InstantTrackingState _currentTrackerState =
InstantTrackingState.Initializing;
   private bool _changingState = false;
   private GridRenderer _gridRenderer;
   private List<GameObject> _augmentations = new
List<GameObject>();
   private InstantTrackable _trackable;
   private bool _isTracking = false;

   private void Awake() {
      Application.targetFrameRate = 60;

      _gridRenderer = GetComponent<GridRenderer>();
      _gridRenderer.enabled = false;

      _trackable                                    =
Tracker.GetComponentInChildren<InstantTrackable>();

Tracker.OnScreenConversionComputed.AddListener(OnScreen
ConversionComputed);
   }

   protected override void Start() {
      base.Start();

      MessageBox.text = "Starting the SDK";
      // The AR SDK needs to be fully started before we can
query for ARKit / ARCore support
      // SDK initialization happens during start, so we wait one
frame in a coroutine
      StartCoroutine(CheckPlatformAssistedTrackingSupport());
   }

      private                                       IEnumerator
CheckPlatformAssistedTrackingSupport() {
      yield return null;
      if (Tracker.SMARTEnabled) {

Tracker.IsPlatformAssistedTrackingSupported((SmartAvailabili
ty smartAvailability) => {
            UpdateTrackingMessage(smartAvailability);
         });
      }
   }

   private void UpdateTrackingMessage(SmartAvailability
smartAvailability) {
```

```
      if (Tracker.SMARTEnabled) {
        string sdk;
        if (Application.platform == RuntimePlatform.Android)
{
          sdk = "ARCore";
        } else if (Application.platform ==
RuntimePlatform.IPhonePlayer) {
          sdk = "ARKit";
        } else {
          return;
        }

        switch (smartAvailability) {
          case SmartAvailability.IndeterminateQueryFailed: {
            MessageBox.text = "Platform support query failed.
Running without platform assisted tracking support.";
            break;
          }
          case SmartAvailability.CheckingQueryOngoing: {
            MessageBox.text = "Platform support query
ongoing.";
            break;
          }
          case SmartAvailability.Unsupported: {
            MessageBox.text = "Running without platform
assisted tracking support.";
            break;
          }
          case SmartAvailability.SupportedUpdateRequired:
          case SmartAvailability.Supported: {
            string runningWithMessage = "Running with
platform assisted tracking support (" + sdk + ").";

            if (_currentTrackerState ==
InstantTrackingState.Tracking) {
              MessageBox.text = runningWithMessage;
            } else {
              MessageBox.text = runningWithMessage + "\n
Move your phone around until the target turns green, which is
when you can start tracking.";
            }
            break;
          }
        }
      } else {
        MessageBox.text = "Running without platform assisted
tracking support.";
      }
    }

    protected override void Update() {
      base.Update();

      if (_isTracking && Input.GetMouseButtonUp(0)) {
        Tracker.ConvertScreenCoordinate(Input.mousePosition);
      }

      if (_currentTrackerState ==
InstantTrackingState.Initializing) {
        if (Tracker.CanStartTracking()) {
          _gridRenderer.TargetColor = Color.green;
        } else {
          _gridRenderer.TargetColor =
GridRenderer.DefaultTargetColor;
        }
      } else {
        _gridRenderer.TargetColor =
GridRenderer.DefaultTargetColor;
      }
    }


    public void OnStateChanged(InstantTrackingState newState)
{
      _currentTrackerState = newState;

      if (_currentTrackerState ==
InstantTrackingState.Initializing) {
        ToggleStateButtonText.text = "Start Tracking";
        HeightSlider.SetActive(true);
      } else {
        ToggleStateButtonText.text = "Start Initialization";
        HeightSlider.SetActive(false);
      }

      _changingState = false;
    }

    public void OnScreenConversionComputed(bool success,
Vector2 screenCoordinate, Vector3 pointCloudCoordinate) {
      if (success) {
        var newAugmentation =
GameObject.Instantiate(Augmentation, _trackable.transform) as
GameObject;
        // The pointCloudCoordinate values are in the local
space of the trackable.
        newAugmentation.transform.localPosition =
pointCloudCoordinate;
        newAugmentation.transform.localScale = Vector3.one;
        _augmentations.Add(newAugmentation);
      }
    }

    public void OnToggleStateButtonPressed() {
      if (!_changingState) {

        if (_currentTrackerState ==
InstantTrackingState.Initializing) {
          if (Tracker.CanStartTracking()) {
            ToggleStateButtonText.text = "Switching State...";
            _changingState = true;
            Tracker.SetState(InstantTrackingState.Tracking);
          }
        } else {
          // Clear all the previous augmentations
          foreach (var augmentation in _augmentations) {
            Destroy(augmentation);
          }
          _augmentations.Clear();

          ToggleStateButtonText.text = "Switching State...";
          _changingState = true;
          Tracker.SetState(InstantTrackingState.Initializing);
        }
      }
    }

    public void OnInitializationStarted(InstantTarget target) {
      SetSceneEnabled(true);
    }

    public void OnInitializationStopped(InstantTarget target) {
      SetSceneEnabled(false);
    }

    public void OnSceneRecognized(InstantTarget target) {
      SetSceneEnabled(true);
      _isTracking = true;
    }

    public void OnSceneLost(InstantTarget target) {
      SetSceneEnabled(false);
      _isTracking = false;
    }

    private void SetSceneEnabled(bool enabled) {
      _gridRenderer.enabled = enabled;
      // Because the InstantTrackable has the Auto Toggle
Visibility option enabled
      // and because all the augmentations are set as children to it,
we don't need to hide them.
    }

    public void OnHeightValueChanged(float newHeightValue)
{
      HeightLabel.text = string.Format("{0:0.##} m",
newHeightValue);
      Tracker.DeviceHeightAboveGround = newHeightValue;
```

```
    }

    public void OnError(Error error) {
        _changingState = false;
        ToggleStateButtonText.text = "Start Tracking";
        PrintError("Instant Tracker error!", error);
    }
}
```

Bed Button

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BedButton : MonoBehaviour {
    public GameObject ShowedButton;
                // Use this for initialization
                public void OnClicked () {
        if (ShowedButton.activeSelf == false)
        {
            ShowedButton.SetActive(true);
        } else
        {
            ShowedButton.SetActive(false);
        }
                }

    public void ResetClicked()
    {
        ShowedButton.SetActive(false);
    }

    // Update is called once per frame
    void Update () {

                }
}
```

Creator Info

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CreatorInfo : MonoBehaviour {
    public GameObject CreatorInfoPanel;
    // Use this for initialization
    void Start () {

                }

    public void OnCreatorInfoButtonPressed()
    {
        CreatorInfoPanel.SetActive(true);
    }
    public void OnCreatorInfoDoneButtonPressed()
    {
        CreatorInfoPanel.SetActive(false);
    }
    // Update is called once per frame
    void Update () {

                }
}
```

Rotation Controller

```
using UnityEngine;

public class RotationController : MonoBehaviour
{
    private InstantTrackingController _controller;
    private Transform _activeObject = null;

    private Vector3 _touch1StartGroundPosition;
    private Vector3 _touch2StartGroundPosition;
    private Vector3 _startObjectRotation;
    private bool rotating = false;
    public const float TOUCH_ROTATION_WIDTH = 1;
    public const float TOUCH_ROTATION_MINIMUM = 1;
    Vector2 startVector = Vector2.zero;
    private void Start()
    {
        _controller = GetComponent<InstantTrackingController>();
    }

    private void Update()
    {
        if (Input.touchCount >= 2)
        {
            Touch touch1 = Input.GetTouch(0);
            Touch touch2 = Input.GetTouch(1);
            Transform hitTransform;

            if (_activeObject == null)
            {
                if (GetTouchObject(touch1.position, out hitTransform))
                {
                    SetTouchObject(hitTransform);
                }
                else if (GetTouchObject(touch2.position, out hitTransform))
                {
                    SetTouchObject(hitTransform);
                }
                else if (GetTouchObject((touch1.position + touch2.position) / 2, out hitTransform))
                {
                    SetTouchObject(hitTransform);
                }

                if (_activeObject != null)
                {
                    _touch1StartGroundPosition = GetGroundPosition(touch1.position);
                    _touch2StartGroundPosition = GetGroundPosition(touch2.position);
                    _startObjectRotation = _activeObject.localEulerAngles;
                }
            }

            if (_activeObject != null)
            {
                var touch1GroundPosition = GetGroundPosition(touch1.position);
                var touch2GroundPosition = GetGroundPosition(touch2.position);
                if (!rotating)
                {
                    startVector = touch2GroundPosition - touch1GroundPosition;
                    rotating = startVector.sqrMagnitude > TOUCH_ROTATION_WIDTH;
                }
                else
                {
                    Vector2 currVector = Input.touches[1].position - Input.touches[0].position;
                    float angleOffset = Vector2.Angle(startVector, currVector);
                    if (angleOffset > TOUCH_ROTATION_MINIMUM)
                    {
                        Vector3 LR = Vector3.Cross(startVector, currVector); // z > 0 left rotation, z < 0 right rotation
                        float a = _activeObject.localEulerAngles.y;
                        Vector3 b = _activeObject.localEulerAngles;
                        if (LR.z > 0)
                            a += angleOffset;
                        else if (LR.z < 0)
                            a -= angleOffset;

                        _activeObject.localEulerAngles = new Vector3(_activeObject.localEulerAngles.x,
```

```
_activeObject.localEulerAngles.y + a,
_activeObject.localEulerAngles.z);
                    startVector = currVector;
                }
            }
        }
    }
    else
    {
        rotating = false;
        _activeObject = null;
    }
}

    private bool GetTouchObject(Vector2 touchPosition, out
Transform hitTransform)
    {
        var touchRay =
Camera.main.ScreenPointToRay(touchPosition);
        touchRay.origin -= touchRay.direction * 100.0f;

        RaycastHit hit;
        if (Physics.Raycast(touchRay, out hit))
        {
            hitTransform = hit.transform;
            return true;
        }

        hitTransform = null;
        return false;
    }

    private Vector3 GetGroundPosition(Vector2 touchPosition)
    {
        var groundPlane = new Plane(Vector3.up, Vector3.zero);
        var touchRay =
Camera.main.ScreenPointToRay(touchPosition);
        float enter;
        if (groundPlane.Raycast(touchRay, out enter))
        {
        android:required="true" />

<!-- do not support small resolution screens -->
<supports-screens
        android:anyDensity="true"
        android:largeScreens="true"
        android:normalScreens="true"
        android:smallScreens="false"
        android:xlargeScreens="true" />

<application
        android:label="@string/app_name"
        android:icon="@drawable/app_icon">
    <activity
android:name="com.unity3d.player.UnityPlayerActivity"

android:configChanges="fontScale|keyboard|keyboardHidden|l
ocale|mnc|mcc|navigation|orientation|screenLayout|screenSize|s
mallestScreenSize|uiMode|touchscreen">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category
android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>

    <meta-data android:name="com.google.ar.core"
android:value="required" />
</application>
</manifest>
```

```
            return touchRay.GetPoint(enter);
        }
        return Vector3.zero;
    }

    private void SetTouchObject(Transform newObject)
    {
        if
(_controller.ActiveModels.Contains(newObject.gameObject))
        {
            _activeObject = newObject;
        }
    }
}
```

Android Manifest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.AR.unity_plugin">
<uses-sdk android:minSdkVersion="19"
android:targetSdkVersion="24" />

<uses-permission
android:name="android.permission.CAMERA" />
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STO
RAGE" />
<uses-permission
android:name="android.permission.INTERNET" />

<!-- Tell the system this app requires OpenGL ES 2.0. -->
<uses-feature
        android:glEsVersion="0x00020000"
        android:required="true" />

<!-- rear facing cam -->
<uses-feature
        android:name="android.hardware.camera"
```