Table of Contents

Quick Start

The example code provided in this quick start guide is for educational and demonstration purposes only. It may not represent best practices for production use. This quick start was last updated for **UXR.QuestCamera v3.1.0**.

Breaking Changes Notice

If you've just updated the package to v3.0.0 or later, it's recommended to check the migration guide for information on breaking changes from v2.6.1, including async disposal patterns, renamed types (e.g., `CaptureSessionObject<T>` → `CapturePipeline<T>`), and nullable-aware APIs.

Setup

Dependencies

UXR.QuestCamera uses an AAR plugin to access the Camera2 API and requires the **External Dependency Manager for Unity (EDM4U)** package to handle native dependencies. If you have Firebase or Google SDKs in your project, you likely have it installed. If not, you can see the installation steps here: https://github.com/googlesamples/unity-jar-resolver?tab=readme-ov-file#getting-started

Unity Version Compatibility

This package uses some `Awaitable` methods for switching between threads for frame processing. Since `Awaitable` was only added in Unity 6, you will have to install **com.utilities.async** by Stephen Hodgson on older versions of Unity. You can see the installation steps here: https://github.com/RageAgainstThePixel/com.utilities.async

AndroidManifest.xml

> **NOTE**
>
> You can skip this step if you're using the Meta XR Core SDK v81 or higher by enabling the ‘Enabled Passthrough Camera Access’ setting in your `OVR Manager` instance and regenerating your AndroidManifest using the SDK's tools.

Add the following to your project's `AndroidManifest.xml` file:

```xml
<uses-feature android:name="android.hardware.camera2.any" android:required="true"/>
<uses-permission android:name="horizonos.permission.HEADSET_CAMERA"
android:required="true"/>
```

The `HEADSET_CAMERA` permission is required by Horizon OS for apps to access the headset cameras. You must request it at runtime before using any of this package's APIs, like so:

```
if (!Permission.HasUserAuthorizedPermission(UCameraManager.HeadsetCameraPermission))
    Permission.RequestUserPermission(UCameraManager.HeadsetCameraPermission);
```

Usage

Device Support

The Passthrough Camera API is restricted to the Quest 3 family and newer devices on Horizon OS version >= 74. Check if the current device is supported with the `CameraSupport.IsSupported` static property.

Choosing the Camera

`UCameraManager` allows you to access `CameraInfo` objects and create `CameraDevice` objects. It's a persistent singleton, so add it to a GameObject in the first scene it's referenced in, and it can be used in any scene loaded thereafter.

`UCameraManager` creates a `CameraInfo` object for each camera the native plugin detects. Each `CameraInfo` then exposes the supported resolutions and intrinsic information of the physical camera device. You can get an array of all detected cameras in `UCameraManager.Cameras`, or request a camera associated with the left or right eye using `UCameraManager.GetCamera(CameraInfo.CameraEye)`.

Even though `CameraInfo` is an `IDisposable`, `UCameraManager` manages the disposal of the objects returned by the above APIs. You can get independently managed `CameraInfo` objects using `UCameraManager.GetCameraInfos()`.

Opening the Camera

You can open a camera device using `UCameraManager.OpenCamera(cameraId)`. Pass the `CameraInfo` object to this function (it is implicitly converted to the ID of the camera it represents). This method returns a `CameraDevice?`.

`CameraDevice` is a wrapper for the native Camera2 `CameraDevice` class. It allows you to create capture sessions, which provide actual images from the camera. It takes a bit for the camera device to open, so you need to wait for it using `CameraDevice.WaitForInitialization()` (Coroutine) or `CameraDevice.WaitForInitializationAsync()` (Task).

The task returns a boolean confirming that the camera opened successfully. When using the coroutine method, use the `CurrentState` property after yielding to confirm the camera opened. To get specific error reasons, check logcat or add listeners to `CameraDevice.OnDeviceDisconnected` and `CameraDevice.OnDeviceErred`.

If the camera couldn't open, release its native resources by awaiting `camera.DisposeAsync()`. You can yield async calls in coroutines using the package-provided `.Yield()` extension.

Creating a Capture Session

After opening a camera device, you can start a capture session. Keep in mind that you can't close the camera device while the capture session is still active.

You can create two kinds of capture sessions: continuous and on-demand. A continuous session streams a sequence of frames to Unity, each converted from YUV to RGBA. If you don't need a live feed, you can save resources by using an on-demand capture session. On-demand sessions only process frames when explicitly requested.

To create a new continuous session, use `CameraDevice.CreateContinuousCaptureSession(resolution)`. To create an on-demand session, use `CameraDevice.CreateOnDemandCaptureSession(resolution)`. Supported resolutions for the camera are exposed in `CameraInfo.SupportedResolutions` as an array of Unity's `Resolution` objects. The last value in the array is usually the highest resolution.

These methods return `CapturePipeline<ContinuousCaptureSession>?` and `CapturePipeline<OnDemandCaptureSession>?` objects respectively. Each contains the session object (`CapturePipeline<T>.CaptureSession`) and a YUV-to-RGBA texture converter (`CapturePipeline<T>.TextureConverter`).

As with `CameraDevice`, wait for the session to open using `CaptureSession.WaitForInitialization()` or `CaptureSession.WaitForInitializationAsync()`, and check `CurrentState` when using the coroutine method. If the session could not be started successfully, release its native resources by awaiting `CapturePipeline<T>.DisposeAsync()`. For error details, check logcat or add listeners to `CaptureSession.OnSessionConfigurationFailed` and `CaptureSession.OnSessionRequestFailed`.

Once started, you'll get frames from the camera in an ARGB32 `RenderTexture` stored in `TextureConverter.FrameRenderTexture`. For on-demand sessions, the `RenderTexture` remains black until you call `CaptureSession.RequestCapture()`, which can be called any number of times. The method returns a boolean indicating whether the request succeeded. You can then await `TextureConverter.GetNextFrameAsync()` to get the requested frame, along with the capture timestamp.

When you're done with the session, dispose of it by awaiting `CapturePipeline<T>.DisposeAsync()`. This disposes both the texture converter and capture session simultaneously. You can then dispose of the camera device, ensuring you close the capture session *before* closing the camera.

See Unity's `RenderTexture` documentation for information on reading pixel data to the CPU: https://docs.unity3d.com/6000.0/Documentation/ScriptReference/RenderTexture.html

Capture Templates

UXR.QuestCamera supports a subset of Camera2's capture templates. By default, `CameraDevice.CreateContinuousCaptureSession` uses TEMPLATE_PREVIEW, and `OnDemandCaptureSession.RequestCapture()` uses TEMPLATE_STILL_CAPTURE. You can change them by specifying one of the templates defined in the `CaptureTemplate` enum.

Releasing Resources

Make sure to dispose of all camera resources *immediately* after you finish using them so the native camera device and capture session are properly closed. You can also force closure synchronously, for example in `OnApplicationQuit`, where Unity won't wait for async methods:

```
Task.WhenAll(
    captureSession.DisposeAsync().AsTask(),
    cameraDevice.DisposeAsync().AsTask()
).Wait();

Debug.Log("Synchronously closed resources.");
```

Using the `await using` Pattern

If you can use C#'s `await using` statement, you can simplify the entire process significantly. For example:

```
public async Task TakePicture()
{
    if (UCameraManager.Instance.GetCamera(CameraInfo.CameraEye.Left) is not
CameraInfo cameraInfo)
    {
        Debug.LogError("Could not get camera info!");
        return;
    }

    await using CameraDevice? cameraDevice = UCameraManager.Instance.OpenCamera(cameraInfo);
    if (cameraDevice == null || !await cameraDevice.WaitForInitializationAsync())
    {
        Debug.LogError("Could not open camera!");
        return;
    }

    Resolution resolution = cameraInfo.SupportedResolutions[^1];
    await using CapturePipeline<OnDemandCaptureSession>? capturePipeline =
cameraDevice.CreateOnDemandCaptureSession(resolution);
    if (capturePipeline == null || !await
capturePipeline.CaptureSession.WaitForInitializationAsync())
    {
        Debug.LogError("Could not open capture session!");
        return;
    }

    if (!capturePipeline.CaptureSession.RequestCapture())
    {
        Debug.LogError("Could not capture frame!");
        return;
    }

    (RenderTexture texture, long timestamp) = await
capturePipeline.TextureConverter.GetNextFrameAsync();
    // Process the RenderTexture here!
}
```

Better Performance in OpenGL

If your app uses the OpenGL Graphics API, you can use `SurfaceTextureCaptureSession` and `OnDemandSurfaceTextureCaptureSession` (in the `Uralstech.UXR.QuestCamera.SurfaceTextureCapture` namespace) instead of `ContinuousCaptureSession` and `OnDemandCaptureSession`. This can improve performance since SurfaceTexture-based sessions use low-level OpenGL shaders for YUV-to-RGBA conversion.

They're also simpler to use, as they don't require additional components like texture converters or frame forwarders. Both provide a read-only `Texture` property that stores the camera images.

You can create them by calling `CameraDevice.CreateSurfaceTextureCaptureSession()` or `CameraDevice.CreateOnDemandSurfaceTextureCaptureSession()`, like so:

```
CameraDevice camera = ...;
Resolution resolution = ...;

// Create a capture session with the camera at the chosen resolution.
SurfaceTextureCaptureSession? session =
camera.CreateSurfaceTextureCaptureSession(resolution);
if (session == null) { /* Handle error */ }
yield return session.WaitForInitialization();

// Check if it opened successfully
if (session.CurrentState != NativeWrapperState.Opened)
{
    Debug.LogError("Could not open camera session!");

    // Release camera and session resources.
    yield return session.DisposeAsync().Yield();
    yield return camera.DisposeAsync().Yield();
    yield break;
}

// Set the image texture.
_rawImage.texture = session.Texture;
```

Example Script

```
using System.Collections;
using UnityEngine;
using UnityEngine.Android;
using UnityEngine.UI;
using Uralstech.UXR.QuestCamera;

public class CameraTest : MonoBehaviour
{
    [SerializeField] private RawImage _rawImage;

    private IEnumerator Start()
    {
        // Check if the current device is supported.
        if (!CameraSupport.IsSupported)
        {
            Debug.LogError("Device does not support the Passthrough Camera API!");
            yield break;
        }

        // Check for permission.
        if (!Permission.HasUserAuthorizedPermission(UCameraManager.HeadsetCameraPermission))
        {
            // If the has not yet given the permission, request it and exit out of
this function.
            Permission.RequestUserPermission(UCameraManager.HeadsetCameraPermission);
            yield break;
        }

        // Get a camera device.
        CameraInfo? currentCamera =
UCameraManager.Instance.GetCamera(CameraInfo.CameraEye.Left);
        if (currentCamera == null)
        {
            Debug.LogError("No camera available!");
            yield break;
        }
        // Get the supported resolutions of the camera and choose the highest resolution.
        Resolution highestResolution = default;
        foreach (Resolution resolution in currentCamera.SupportedResolutions)
        {
            if (resolution.width * resolution.height > highestResolution.width
* highestResolution.height)
                highestResolution = resolution;
        }

        // Open the camera.
        CameraDevice? camera = UCameraManager.Instance.OpenCamera(currentCamera);
        if (camera == null)
        {
            Debug.LogError("Could not open camera!");
            yield break;
        }

        yield return camera.WaitForInitialization();

        // Check if it opened successfully
        if (camera.CurrentState != NativeWrapperState.Opened)
        {
            Debug.LogError("Could not open camera!");

            // Very important, this frees up any resources held by the camera.
            yield return camera.DisposeAsync().Yield();
            yield break;
        }

        // Create a capture session with the camera, at the chosen resolution.
        CapturePipeline<ContinuousCaptureSession>? sessionPipeline =
camera.CreateContinuousCaptureSession(highestResolution);
        if (sessionPipeline == null)
        {
            Debug.LogError("Could not create session!");
            yield return camera.DisposeAsync().Yield();
            yield break;
        }

        yield return sessionPipeline.CaptureSession.WaitForInitialization();

        // Check if it opened successfully
        if (sessionPipeline.CaptureSession.CurrentState != NativeWrapperState.Opened)
        {
            Debug.LogError("Could not open camera session!");
            // Both of these are important for releasing the camera and session resources.
            yield return sessionPipeline.DisposeAsync().Yield();
            yield return camera.DisposeAsync().Yield();
            yield break;
        }

        // Set the image texture.
        _rawImage.texture = sessionPipeline.TextureConverter.FrameRenderTexture;

        // Optional: Dispose at end of use (e.g., in StopCoroutine or OnDestroy)
        // yield return sessionPipeline.DisposeAsync().Yield();
        // yield return camera.DisposeAsync().Yield();
    }
}
```

Sample - Digit Recognition with Unity Inference Engine

The package contains a Computer Vision sample that uses an MNIST trained model to recognize handwritten digits, through the Camera API.

Package Dependencies

This sample requires the Unity Inference Engine package (`com.unity.ai.inference`) and was built with version 2.3.0 of the package.

Advanced Samples

This page contains some samples for advanced use-cases, like custom texture converters or multi-camera streaming.

Custom Texture Converters

The texture converter in `CapturePipeline<T>.TextureConverter` allows you to easily change the conversion compute shader to custom ones. All you have to do is set `CapturePipeline<T>.TextureConverter.Shader` to your shader. You can also change the compute shader for all new capture sessions by changing `UCameraManager.YUVToRGBAComputeShader`.

For example, the following compute shader ignores the U and V values of the YUV stream to provide a Luminance-only image:

```
#pragma kernel CSMain

// Input buffers (read-only)
ByteAddressBuffer YBuffer;
ByteAddressBuffer UBuffer;
ByteAddressBuffer VBuffer;

// Row strides
uint YRowStride;
uint UVRowStride;

// Pixel strides
uint UVPixelStride;

// Image dimensions
uint TargetWidth;
uint TargetHeight;

// Output texture (read-write)
RWTexture2D<float4> OutputTexture;

// Helper function to get a byte from a ByteAddressBuffer.
//  buffer: The ByteAddressBuffer.
//  byteIndex: The *byte* index (offset) into the buffer.
uint GetByteFromBuffer(ByteAddressBuffer buffer, uint byteIndex)
{
    // Calculate the 32-bit word offset (each word is 4 bytes).
    uint wordOffset = byteIndex / 4;

    // Load the 32-bit word containing the byte.
    uint word = buffer.Load(wordOffset * 4); // MUST multiply by 4 for ByteAddressBuffer.Load()

    // Calculate the byte position *within* the word (0, 1, 2, or 3).
    uint byteInWord = byteIndex % 4;

    // Extract the correct byte using bit shifts and masking.
    return (word >> (byteInWord * 8)) & 0xFF;
}

[numthreads(8, 8, 1)]
void CSMain(uint3 id : SV_DispatchThreadID)
{
    if (id.x >= TargetWidth || id.y >= TargetHeight)
        return;

    // The YUV stream is flipped, so we have to un-flip it.
    uint flippedY = TargetHeight - 1 - id.y;

    // Index of Y value in buffer.
    uint yIndex = flippedY * YRowStride + id.x;
    uint yValue = GetByteFromBuffer(YBuffer, yIndex);

    float3 luminance = float3(yValue, yValue, yValue) / 255.0;
    OutputTexture[id.xy] = float4(luminance.rgb, 1.0);
}
```

Multiple Streams From One Camera

By adding multiple texture converters to the same request, you can emulate the effect of having more than one image stream from a single camera. For example, you can have one converter stream the camera image as-is, and another streaming with a simple Sepia post-processing effect:

```
// Create a capture session with the camera, at the chosen resolution.
CapturePipeline<ContinuousCaptureSession> capturePipeline =
camera.CreateContinuousCaptureSession(highestResolution);
if (capturePipeline == null...

yield return capturePipeline.CaptureSession.WaitForInitialization();

// Check if it opened successfully.
if (capturePipeline.CaptureSession.CurrentState...

// Set the image texture.
_rawImage.texture = capturePipeline.TextureConverter.FrameRenderTexture;

// Create a new YUVToRGBAConverter.
```

```
YUVToRGBAConverter secondary = new YUVToRGBAConverter(highestResolution);

// Assign it a different shader.
secondary.Shader = _postProcessShader;

// Link the capture session and the converter.
capturePipeline.CaptureSession.OnFrameReady += secondary.OnFrameReady;

// Set the second image to the post processed RenderTexture.
_rawImagePostProcessed.texture = secondary.FrameRenderTexture;
```

YUV To RGBA Converter With Sepia Effect

```
#pragma kernel CSMain

// Input buffers (read-only)
ByteAddressBuffer YBuffer;
ByteAddressBuffer UBuffer;
ByteAddressBuffer VBuffer;

// Row strides
uint YRowStride;
uint UVRowStride;

// Pixel strides
uint UVPixelStride;

// Image dimensions
uint TargetWidth;
uint TargetHeight;

// Output texture (read-write)
RWTexture2D<float4> OutputTexture;

// Helper function to get a byte from a ByteAddressBuffer.
//  buffer: The ByteAddressBuffer.
//  byteIndex: The *byte* index (offset) into the buffer.
uint GetByteFromBuffer(ByteAddressBuffer buffer, uint byteIndex)
{
    // Calculate the 32-bit word offset (each word is 4 bytes).
    uint wordOffset = byteIndex / 4;

    // Load the 32-bit word containing the byte.
    uint word = buffer.Load(wordOffset * 4); // MUST multiply by 4 for ByteAddressBuffer.Load()

    // Calculate the byte position *within* the word (0, 1, 2, or 3).
    uint byteInWord = byteIndex % 4;

    // Extract the correct byte using bit shifts and masking.
    return (word >> (byteInWord * 8)) & 0xFF;
}

[numthreads(8, 8, 1)]
void CSMain(uint3 id : SV_DispatchThreadID)
{
    if (id.x >= TargetWidth || id.y >= TargetHeight)
        return;

    // The YUV stream is flipped, so we have to un-flip it.
    uint flippedY = TargetHeight - 1 - id.y;

    // Index of Y value in buffer.
    uint yIndex = flippedY * YRowStride + id.x;
    uint yValue = GetByteFromBuffer(YBuffer, yIndex);

    float3 luminance = float3(yValue, yValue, yValue) / 255.0;

    // --- Post-processing (Sepia Tone) ---
    float4 color = float4(luminance.rgb, 1.0);

    //Simple Sepia.  Could also do a vignette, bloom, etc. here.
    float4 sepiaColor;
    sepiaColor.r = dot(color.rgb, float3(0.393, 0.769, 0.189));
    sepiaColor.g = dot(color.rgb, float3(0.349, 0.686, 0.168));
    sepiaColor.b = dot(color.rgb, float3(0.272, 0.534, 0.131));
    sepiaColor.a = 1.0;

    OutputTexture[id.xy] = sepiaColor;
}
```

Migrating to UXR.QuestCamera V3

UXR.QuestCamera v3 introduces a soft rewrite of the package with several breaking changes. It's highly recommended to update to v3, as it includes fixes for many crashes and stutters. This page documents all changes to the **public API** in v3.0.0 and v3.1.0 compared to v2.6.1.

Basic Sample

This script shows the most important changes in V3 compared to V2:

```
// Open the camera.
CameraDevice camera = UCameraManager.Instance.OpenCamera(currentCamera);
if (camera == null)
{
    Debug.LogError("Could not open camera!");
    yield break;
}

yield return camera.WaitForInitialization();

// Check if it opened successfully
if (camera.CurrentState != NativeWrapperState.Opened)
{
    Debug.LogError("Could not open camera!");

    // Very important, this frees up any resources held by the camera.

    // V2: camera.Destroy();
    // V3: Yield to DisposeAsync() using extension instead of calling Destroy().
    yield return camera.DisposeAsync().Yield();
    yield break;
}

// Create a capture session with the camera, at the chosen resolution.
// V2: CreateContinuousCaptureSession returns CaptureSessionObject<ContinuousCaptureSession>
// V3: CreateContinuousCaptureSession returns CapturePipeline<ContinuousCaptureSession>

CapturePipeline<ContinuousCaptureSession> sessionObject =
camera.CreateContinuousCaptureSession(highestResolution);
if (sessionObject == null)
{
    Debug.LogError("Could not open camera session!");
    yield return camera.DisposeAsync().Yield(); // V3 dispose
    yield break;
}
```

```
yield return sessionObject.CaptureSession.WaitForInitialization();

// Check if it opened successfully
if (sessionObject.CaptureSession.CurrentState != NativeWrapperState.Opened)
{
    Debug.LogError("Could not open camera session!");

    // Both of these are important for releasing the camera and session resources.

    // V2: sessionObject.Destroy();
    // V3: Yield to DisposeAsync().
    yield return sessionObject.DisposeAsync().Yield();

    yield return camera.DisposeAsync().Yield(); // V3 dispose
    yield break;
}

// Set the image texture.
_rawImage.texture = sessionObject.TextureConverter.FrameRenderTexture;
```

CameraDevice

`CameraDevice` is no longer a `MonoBehavior` and now implements `AndroidJavaProxy` and `IAsyncDisposable`.

- **Removed**
    - `Release()`, `Destroy()` — replaced by `DisposeAsync()`.
    - `IsActiveAndUsable`
- **Changed**
    - `OnDeviceOpened`: `Action<string>` — parameter is the ID of the opened camera.
    - `OnDeviceClosed`: `Action<string?>` — parameter is the ID of the closed camera, or `null` if it failed to open.
    - `OnDeviceErred`: `Action<string?, ErrorCode>` — parameters are the ID of the erred camera (or `null` if it failed to open) and the error code.
    - `OnDeviceDisconnected`: `Action<string>` — parameter is the ID of the disconnected camera.
    - `WaitForInitialization()` now returns a `WaitUntil` object and throws `ObjectDisposedException` if the `CameraDevice` was disposed at the time of calling.
    - `WaitForInitializationAsync()` now accepts an optional `CancellationToken` and throws `ObjectDisposedException` if the `CameraDevice` was disposed at the time of calling.
    - `CreateContinuousCaptureSession()` and `CreateOnDemandCaptureSession()` now return `CapturePipeline<ContinuousCaptureSession>?` and `CapturePipeline<OnDemandCaptureSession>?`,

respectively, and throw `ObjectDisposedException` if the `CameraDevice` was disposed at the time of calling.
    - `CreateSurfaceTextureCaptureSession()` and `CreateOnDemandSurfaceTextureCaptureSession()` now have nullable return types and throw `ObjectDisposedException` if the `CameraDevice` was disposed at the time of calling.
    - `CameraId` is no longer a property and is now a cached value.
- **v3.1.0 Specific Changes**
    - `WaitForInitializationAsync()` now returns `Task<bool>` representing the open state of the device.

CaptureSessionObject

`CaptureSessionObject<T>` has been replaced by [`CapturePipeline<T>`](), which implements `IAsyncDisposable`.

- **Removed**
    - `GameObject`
    - `CameraFrameForwarder` — functionality moved to `ContinuousCaptureSession`.
    - `Destroy()` — replaced by `DisposeAsync()`.

ContinuousCaptureSession

[`ContinuousCaptureSession`]() is no longer a `MonoBehavior` and now implements `AndroidJavaProxy` and `IAsyncDisposable`.

- **Removed**
    - `Release()` — replaced by `DisposeAsync()`.
    - `IsActiveAndUsable`
- **Changed**
    - `OnSessionConfigurationFailed`: `Action<bool>` — parameter indicates whether the failure was caused by a camera access or security exception.
    - `OnSessionConfigured`, `OnSessionRequestSet`, `OnSessionRequestFailed`: `Action`
    - `WaitForInitialization()` now returns a `WaitUntil` object and throws `ObjectDisposedException` if the `ContinuousCaptureSession` was disposed at the time of calling.
    - `WaitForInitializationAsync()` now accepts an optional `CancellationToken` and throws `ObjectDisposedException` if the `ContinuousCaptureSession` was disposed at the time of calling.
- **v3.1.0 Specific Changes**

    - `WaitForInitializationAsync()` now returns `Task<bool>` representing the open state of the session.

OnDemandCaptureSession

`OnDemandCaptureSession` inherits from `ContinuousCaptureSession` and includes the same breaking changes, plus the following:

- **Changed**
    - `RequestCapture()` now throws `ObjectDisposedException` if the `OnDemandCaptureSession` was disposed at the time of calling.

YUVToRGBAConverter

`YUVToRGBAConverter` is no longer a `MonoBehavior` and now implements `IDisposable`.

- **Removed**
    - `Release()` — replaced by `Dispose()`.
    - `OnFrameProcessedWithTimestamp` — replaced with new `OnFrameProcessed`.
    - `SetupCameraFrameForwarder()` — replaced with new constructor `YUVToRGBAConverter(Resolution)`.
    - `CameraFrameForwarder`
- **Changed**
    - `Shader` is now a nullable-aware property.
    - `OnFrameProcessed`: `Action<RenderTexture, long>` — parameters for the frame's `RenderTexture` and the capture's timestamp, in nanoseconds.

SurfaceTextureCaptureSession

`SurfaceTextureCaptureSession` has been moved to the `Uralstech.UXR.QuestCamera.SurfaceTextureCapture` namespace, no longer inherits from `ContinuousCaptureSession`, and now implements `AndroidJavaProxy` and `IAsyncDisposable`.

- **Removed**
    - `Release()` — replaced by `DisposeAsync()`.
    - `Resolution` — use `Texture.width` and `Texture.height` instead.
    - `IsActiveAndUsable`
- **Changed**

  - `Texture` is now a read-only field.
  - `OnSessionConfigurationFailed`: `Action<bool>` — parameter indicates whether the failure was caused by a camera access or security exception.
  - `OnSessionConfigured`, `OnSessionRequestSet`, `OnSessionRequestFailed`: `Action`
  - `WaitForInitialization()` now returns a `WaitUntil` object and throws `ObjectDisposedException` if the `SurfaceTextureCaptureSession` was disposed at the time of calling.
  - `WaitForInitializationAsync()` now accepts an optional `CancellationToken` and throws `ObjectDisposedException` if the `SurfaceTextureCaptureSession` was disposed at the time of calling.
- **v3.1.0 Specific Changes**
  - `WaitForInitializationAsync()` now returns `Task<bool>` representing the open state of the session.

OnDemandSurfaceTextureCaptureSession

`OnDemandSurfaceTextureCaptureSession` (moved to the `Uralstech.UXR.QuestCamera.SurfaceTextureCapture` namespace) inherits from `SurfaceTextureCaptureSession` and includes the same breaking changes, plus the following:

- **Changed**
  - `RequestCapture(Action<Texture2D>)` is now `bool RequestCapture(Action<Texture2D, long>)` where the `long` callback parameter is the capture timestamp, returning the success of the capture, and throws `ObjectDisposedException` if the `OnDemandSurfaceTextureCaptureSession` was disposed at the time of calling.
  - `RequestCapture()` now returns a `WaitUntil?` object (`null` when the capture fails) and throws `ObjectDisposedException` if the `OnDemandSurfaceTextureCaptureSession` was disposed at the time of calling.
  - `RequestCaptureAsync()` is now `Awaitable<(Texture2D?, long)> RequestCaptureAsync()` (`Texture2D` is `null` when the capture fails), accepts an optional `CancellationToken`, and throws `ObjectDisposedException` if the `OnDemandSurfaceTextureCaptureSession` was disposed at the time of calling.
- **v3.1.0 Specific Changes**
  - `RequestCaptureAsync()` now returns `Task<(Texture2D?, long)>`.

CameraInfo

`CameraInfo` is now a record type and implements `IDisposable`.

- **Changed**

    - `CameraEye` now defines the following enumeration values:
        - `Unknown = -1`
        - `Left = 0`
        - `Right = 1`
    - `CameraSource` now defines the following enumeration values:
        - `Unknown = -1`
        - `PassthroughRGB = 0`
    - `LensPoseTranslation` is now nullable (`Vector3?`).
    - `LensPoseRotation` is now nullable (`Quaternion?`).
    - `Intrinsics` is now nullable (`CameraIntrinsics?`).
    - `NativeCameraCharacteristics` is now managed by the `CameraInfo` instance — **do not** dispose it manually.
    - `CameraIntrinsics` is now a record type.
    - All properties are now read-only fields with cached values.

CameraFrameForwarder (Removed)

`CameraFrameForwarder` has been removed. Its functionality has been moved to `ContinuousCaptureSession`.

- **Changed**
    - `OnFrameReady`: `Action<IntPtr, IntPtr, IntPtr, int, int, int, long>` — moved to `ContinuousCaptureSession`.
      See the `OnFrameReady` documentation for parameter details.

CaptureTemplate

- **Removed**
    - `ZeroShutterLag` — not compatible with capture sessions created by the plugin.

UCameraManager

All methods and properties in `UCameraManager` are now nullable-aware with no breaking changes.