Table of Contents

Quick Start

Please note that the code provided in this page is *purely* for learning purposes and is far from perfect. Remember to null-check all responses!

Breaking Changes Notice

If you've just updated the package to v3.0.0 or later, it is 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 Android native AAR plugin, written in Kotlin, to access the Android Camera2 API. The plugin thus requires the External Dependency Manager for Unity (EDM4U) package to add the android.hardware.camera2 package. If your project uses Firebase or Google SDKs, it is likely already installed. If not, you can see the installation steps here: https://github.com/googlesamples/unity-jar-resolver?tab=readme-ov-file#getting-started

Unity Compatibility

The package uses some `Awaitable` methods for switching between the JNI and Unity 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

Installation

To install UXR.QuestCamera:

- Go to `Project Settings` in your Unity project
- Under `Package Manager`, in `Scoped Registries`, create a new registry with the following settings:
  - Name: `OpenUPM`
  - URL: `https://package.openupm.com`
  - Scope(s): `com.uralstech`
- Click `Apply`
- Open the package manager and go to `My Registries` -> `OpenUPM`
- Install UXR.QuestCamera >= `3.0.0`

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:

```
<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 be able to access the headset cameras. You will have to request it at runtime, like so:

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

Usage

Device Support

Since the Passthrough Camera API is restricted to the Quest 3 family and Horizon OS version >= 74, you have to check if it is supported by the current device before doing anything. Just check the static property `CameraSupport.IsSupported` before doing anything.

Choosing the Camera

`UCameraManager` is the script that will allow you to open and close camera devices. It is a persistent singleton, so add it to the first scene that is loaded in your app, so that it can be referenced from all other scripts at all times.

You can add the `QuestCameraManager` prefab, which contains an instance of `UCameraManager`, along with the default YUV to RGBA conversion compute shader, by right clicking on the Hierarchy and clicking on `Quest Camera Manager`, under `Quest Camera`.

To start getting images, you first have to open the camera device. You can have multiple cameras open at the same time, as long as the system allows it.

You can get an array of all available cameras in `UCameraManager.Cameras`, or you can request the camera nearest to the left or right eye of the user by calling `UCameraManager.GetCamera(CameraInfo.CameraEye)`.

The information for each camera, like supported resolutions, position and rotation relative to the headset, focal length, and more are stored in `CameraInfo` objects.

Once you've chosen a camera, you have to select a resolution for the images. You can get them from `CameraInfo.SupportedResolutions`. For example, you can get the highest supported resolution with the following code:

```
Resolution highestResolution = default;
foreach (Resolution resolution in cameraInfo.SupportedResolutions)
{
    if (resolution.width * resolution.height > highestResolution.width
* highestResolution.height)
        highestResolution = resolution;
}
```

Now you're ready to open the camera and start a capture session!

Opening the Camera

You can open the camera by calling `UCameraManager.OpenCamera(cameraId)`. You can pass the previous `CameraInfo` object into this function, as `CameraInfo` will implicitly return its camera's ID as a string.

The function returns a `CameraDevice?` object, which is a wrapper for the native Camera2 `CameraDevice` class. You should then wait for the camera to open. To do so, yield `camera.WaitForInitialization()` (returns `WaitUntil`), or on Unity 6 and above, await `camera.WaitForInitializationAsync()` (supports `CancellationToken`).

After this, you can check the state of the camera by accessing its `CurrentState` property. If it is `NativeWrapperState.Opened`, the camera is ready for use. Otherwise, it means the camera could not be opened successfully. For error details, check logcat or add listeners to `CameraDevice.OnDeviceDisconnected` and `CameraDevice.OnDeviceErred`.

If the camera could not be opened successfully, release its native resources by awaiting `camera.DisposeAsync()`. You can also yield these dispose calls by using the `.Yield()` extension provided by the package.

Creating a Capture Session

You can create two kinds of capture session: continuous and on-demand. A continuous capture session will send each frame recorded by the camera to Unity, and convert it to RGBA. If you don't need the continuous stream of frames, you can save on resources by using an on-demand capture session. On-demand capture sessions will only send camera frames to Unity when requested to do so. Other than that, both function the exact same way.

To create a new continuous capture session, call `CameraDevice.CreateContinuousCaptureSession(resolution)` with the previously chosen resolution. To create an on-demand capture session, call `CameraDevice.CreateOnDemandCaptureSession(resolution)` instead.

They return `CapturePipeline<ContinuousCaptureSession>?` and `CapturePipeline<OnDemandCaptureSession>?` respectively, which contain the session object (`CaptureSession` property), a YUV to RGBA texture converter (`TextureConverter` property), and implement `IAsyncDisposable` for closure and cleanup.

Yield `pipeline.CaptureSession.WaitForInitialization()` or await `pipeline.CaptureSession.WaitForInitializationAsync()` and check `pipeline.CaptureSession.CurrentState`, just like with `CameraDevice`. If the capture session could not be started successfully, release the native resources by awaiting/yielding `pipeline.DisposeAsync()`.

Once started successfully, you will receive the frames from the camera in an ARGB32 format `RenderTexture` as `pipeline.TextureConverter.FrameRenderTexture`. For `OnDemandCaptureSession`s, the `RenderTexture` will remain black until you call `pipeline.CaptureSession.RequestCapture()`, which can be called any number of times (throws `ObjectDisposedException` if disposed).

See the documentation for `RenderTexture` on how to get its pixel data to the CPU: https://docs.unity3d.com/6000.0/Documentation/ScriptReference/RenderTexture.html

Capture Templates

Camera2 allows you to set capture templates for capture requests. `CameraDevice.CreateContinuousCaptureSession()` and `OnDemandCaptureSession.RequestCapture()` also allow you to do so. By default, continuous captures use TEMPLATE_PREVIEW, which is suitable for camera preview windows, and on-demand captures use TEMPLATE_STILL_CAPTURE, which is suitable for still image capture. You can change them by specifying one of the templates defined in the `CaptureTemplate` enum. Note: `ZeroShutterLag` is no longer supported.

Releasing Resources

Make sure to dispose all `CameraDevice`s and `CapturePipeline`s *immediately* after you have finished using them, so that the native camera device and capture session are closed. You can also try to force closure synchronously like in `OnApplicationQuit`, where Unity won't wait for async methods like so:

```
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:

```
// This example requires Unity 6.0 or greater.
public async Awaitable 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()
!= NativeWrapperState.Opened)
    {
        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() != NativeWrapperState.Opened)
    {
        Debug.LogError("Could not open capture session!");
        return;
    }

    TaskCompletionSource<RenderTexture> taskCompletion = new();
    void OnFrameProcessed(RenderTexture texture, long timestamp) =>
taskCompletion.SetResult(texture);

    capturePipeline.TextureConverter.OnFrameProcessed += OnFrameProcessed;
    if (!capturePipeline.CaptureSession.RequestCapture())
    {
        Debug.LogError("Could not capture frame!");
        taskCompletion.SetCanceled();
        return;
    }

    RenderTexture texture = await taskCompletion.Task;
    // 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`, respectively. This can improve performance as the SurfaceTexture-based sessions use low-level OpenGL shaders to convert the camera image from YUV to RGBA. They are also much simpler to use as they do not have any other associated components, like texture converters, or frame forwarders. Both have a read-only `Texture` property, which will store the camera images.

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

```
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!");

    // Both of these are important for releasing the 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 System.Threading.Tasks;
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 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.

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.

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.

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.

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.

UCameraManager

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

CaptureTemplate

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