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, it is recommended to check the *changelogs* for information on breaking changes.

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 >= `2.2.0`

AndroidManifest.xml

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 `CameraDevice.WaitForInitialization()`, or on Unity 6 and above, await `CameraDevice.WaitForInitializationAsync()`.

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 calling `CameraDevice.Destroy`.

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 `CaptureSessionObject<ContinuousCaptureSession>` and `CaptureSessionObject<OnDemandCaptureSession>` respectively, which contain the session object (`ContinuousCaptureSession` or `OnDemandCaptureSession`), a YUV to RGBA texture converter, the native-to-unity frame forwarder, and the `GameObject` that they are components of.

Yield `CaptureSessionObject.CaptureSession.WaitForInitialization()` or await `CaptureSessionObject.CaptureSession.WaitForInitializationAsync()` and check `CaptureSessionObject.CaptureSession.CurrentState`, just like with `CameraDevice`. If the capture session could not be started successfully, release the native resources and `ComputeBuffer`s held by it by calling `CaptureSessionObject.Destroy()`.

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

The first few frames after the session has been started will be black. I've found that waiting around 0.1 seconds after the session has started and then requesting the capture or accessing the image works.

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.

Releasing Resources

It is highly recommended to release or destroy all `CameraDevice`s and `CaptureSessionObject`s *immediately* after you have finished using them, as not doing so may result in the app not closing quickly. They will automatically be released and destroyed by Unity, like when a new scene is loaded, but do not rely on it!

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);

        // 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);
        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.
            camera.Destroy();
            yield break;
        }

        // Create a capture session with the camera, at the chosen resolution.
```

```
        CaptureSessionObject<ContinuousCaptureSession> sessionObject =
camera.CreateContinuousCaptureSession(highestResolution);
        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.
            sessionObject.Destroy();
            camera.Destroy();
            yield break;
        }

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

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 `CaptureSessionObject.TextureConverter` allows you to easily change the conversion compute shader to custom ones. All you have to do is set `CaptureSessionObject.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.
CaptureSessionObject<ContinuousCaptureSession> sessionObject =
camera.CreateContinuousCaptureSession(highestResolution);
yield return sessionObject.CaptureSession.WaitForInitialization();

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

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

// Create a new YUVToRGBAConverter to the current GameObject.
YUVToRGBAConverter secondary = gameObject.AddComponent<YUVToRGBAConverter>();
```

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

// Setup the camera forwarder, which will forward the camera frames in native memory to
the converter.
secondary.SetupCameraFrameForwarder(sessionObject.CameraFrameForwarder, resolution);

// 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;
}
```