DESCRIPTION

This plugin allows you to access the device's camera and stream its contents into a render texture. The reason I made this asset is because there are several issues with the native webcam texture implementation:

1. You can't switch the camera on Android.
2. The camera doesn't start on Safari.
3. The screen tears when you switch from portrait to landscape.

My plugin addresses all issues above and works on all major browsers: Chrome, Safari, Firefox, Opera, Edge.

You can access the front camera, the rear camera, you can check which cameras the device has to begin with. You can start, stop, update the render texture, check if the camera is active, check if you are on desktop.

Please note that my asset requires you to host your game on an https server, not http. Http won't work. In addition, my asset doesn't do anything in play mode, only in the actual build.

If you get a black screen, please check if

1. You are hosting your game on an https server, not http. Use something like https-localhost for testing purposes.
2. You don't have another tab using the webcam.
3. The *Existing Video* option is off. Only use this option if you have a video feed (in HTMLVideoElement form) from another plugin. More about this in the **HOW TO HIJACK AN EXISTING VIDEO ELEMENT** section of this document.

HOW DOES IT WORK?

First, you need to familiarize yourself with basic HTML knowledge. Read about the DOM, a video element. My plugin works by accessing the device's camera using the getUserMedia api. The camera's feed is stored in a video element, and my plugin can

stream its contents to a Unity Render texture. My plugin can access the device's camera, create a video element to store its feed and stream it to a render texture in Unity, or hijack an existing video element created by some other plugin and stream its contents to a render texture.

The main class is DeviceCameraWebGL.cs. DeviceCameraWebGLPrefab.cs was created for the prefab under the prefab folder. Unless you have very specific requirements (which I know some of you do, it's why I rewrote the asset from scratch), using my prefab will be enough. Check out the example scene.

HOW TO USE

First I'll explain the most frequent usage, which is to create the video element yourself, access the device's camera, and stream its contents to a render texture in Unity.

CREATING THE VIDEO ELEMENT

To create a video element, use the following method

```
public static void createVideoElement(Action<uint, uint> callback = null,
string css = "pointer-events: none; width: 1px; height: 1px; z-index:
-1000; position: absolute; left: 0; top: 0;")
```

The first parameter is a callback that takes 2 unsigned integers. This callback will run every time the video is resized, which happens when you switch the phone's orientation for example. It passes the video's width and the video's height, respectively. This information is crucial to know the aspect ratio of the camera feed and display it correctly in Unity. Check out the example scene and the DeviceCameraWebGLPrefab.cs file. Unless you know what you're doing, there is no reason to change the css parameter. It's there to make the video in the DOM not interactive and effectively invisible. Please note that using other css styles to hide the video such as display:none or visibility:hidden are not cross browser solutions. And setting the width and height to 0 fails to work on Safari. If you still want to play with the css, you can change it though.

The created video will have the id stored in the *id* public variable

```
public static string id = "DeviceCameraWebGL";
```

Internally, all methods retrieve the created(or hijacked) video using the querySelector api and the selector stored in the *selector* public variable

```
public static string selector = $"#{id}";
```

For the most frequent use case, you don't need to worry about the id and selector at all.

ACCESSING THE DEVICE'S CAMERA

To access the device's camera, you have the start method

```
public static void start(string constraints, Action<status> callback =
null)
```

It uses the getUserMedia api under the hood. You can check all available constraints here. The constraints can be used in conjunction with the keywords "min", "max", "ideal", "exact". See this example. The first parameter is a json string representing these constraints, and the second parameter is a callback that runs after accessing the device's camera succeeds or fails. The callback takes a *status* parameter, which is the following enum

```
public enum status {

            Success = 0,

            AbortError = 1,

            NotAllowedError  = 2,

            NotFoundError = 3,

            NotReadableError = 4,

            OverconstrainedError = 5,

            SecurityError = 6,
```

```
            TypeError = 7

        };
```

You can check what each error is about here.

Unless you want to play with the constraints and have very specific requirements, you can use the following methods instead that are much easier and straightforward:

```
public static void startFrontCamera(Action<status> callback = null)
```

This method calls the start method to access the front camera, using the following constraints:

```
$"{{\"video\": {{\"width\": {{\"min\": 1280, \"ideal\": 1920, \"max\": 2560}},\"height\": {{\"min\": 720, \"ideal\": 1080, \"max\": 1440}}, \"facingMode\": \"user\"}}}}"
```

Please notice that the keys from the key/value pairs are enclosed in double quotes: “video”, “facingMode”. You absolutely **must** enclose the keys in double quotes, not single quotes, otherwise it won’t work. “user” is a value, not a key, but it’s enclosed in double quotes as well because it’s a string(if it was a number for example, it wouldn’t be).

To access the rear camera, you have this method:

```
public static void startRearCamera(Action<status> callback = null)
```

Which calls *start* with the following constraints

```
$"{{\"video\": {{\"width\": {{\"min\": 1280, \"ideal\": 1920, \"max\": 2560}},\"height\": {{\"min\": 720, \"ideal\": 1080, \"max\": 1440}}, \"facingMode\": \"environment\"}}}}"
```

Under the hood, this method accesses the device’s camera and stores its feed in the video element that was created or hijacked, using the querySelector api and the selector stored in the *selector* public variable.

STREAMING THE VIDEO CONTENTS INTO A RENDER TEXTURE IN UNITY

Now that you created the video element and accessed the device's camera, you need to stream the feed to Unity in a render texture.

Just use the following method

```
public static void updateTexture(IntPtr texturePtr)
```

, which takes a texture pointer as a parameter (doesn't need to be a render texture, but it's easier to use). You should always check if the camera feed is available to update the texture using the *canUpdateTexture*(it's 0 if you can't update, 1 if you can).

```
public static readonly byte canUpdateTexture = 0;
```

Don't be fooled, even though it's a readonly field, it is actually being written into and read from in the javascript side. Then on the Unity side you can do something like what the DeviceCameraWebGLPrefab.cs does:

```
void Update() {

        if (DeviceCameraWebGL.canUpdateTexture != 0) {


DeviceCameraWebGL.updateTexture(targetTexture.GetNativeTexturePtr());

        }

    }
```

And this is pretty much it. Now I'll get into some other useful methods.

STOPPING THE DEVICE'S CAMERA

```
public static void stop(Action callback = null)
```

You can pass a callback that will run once the camera stops.

CHECK IF THE CAMERA IS RUNNING

```
public static bool isPlaying()
```

Returns true if it is, false if not.

CHECK IF ON DESKTOP

```
public static bool isDesktop()
```

Returns true if it is, false if it isn't. This is useful if you want to have a button that switches between front and rear camera, but only want to show the button if you have both camera's available.

SWITCH CAMERAS

```
public static void switchCamera(Action<status> callback = null)
```

This method checks the current camera that is active and switches to the other one. If the user camera is active, it switches to the rear camera. If the rear camera is active, it switches to the user camera. If no camera is active to switch, the front camera is activated.

CHECK CURRENTLY ACTIVE CAMERA

```
public static string getCurrentFacingMode()
```

Returns "environment" if the rear camera is active, "user" if the front camera is active, and "none" if no camera is active.

CHECK AVAILABLE MEDIA INPUT AND OUTPUT ON DEVICE

```
public static void getDevices(Action<MediaDeviceInfo[]> devices)
```

This method uses the enumerateDevices api. It takes a callback which takes an array of MediaDeviceInfo as a parameter, where MediaDeviceInfo is

```
public class MediaDeviceInfo {

            public string deviceId;

            public string groupId;

            public string kind;

            public string label;

        }
```

See here for more information. Please note that this method won't work on Safari if you call it before you start the camera. This method is useful in cases where more than 2 cameras are available and you want to activate a specific one using the deviceId constraint. Something like "`{ video: { deviceId: myPreferredCameraDeviceId } }`" for example.

HOW TO HIJACK AN EXISTING VIDEO ELEMENT

It's very easy. You just need to update the selector to match the video you want to hijack with the querySelector. Example:

```
DeviceCameraWebGL.selector = "video";
```

After that, you need to wait for the video element to be available, using the

```
public static void waitForElement(Action callback)
```

method. It takes a parameterless callback as a parameter that runs once the video element is found. In the callback, you call the following methods

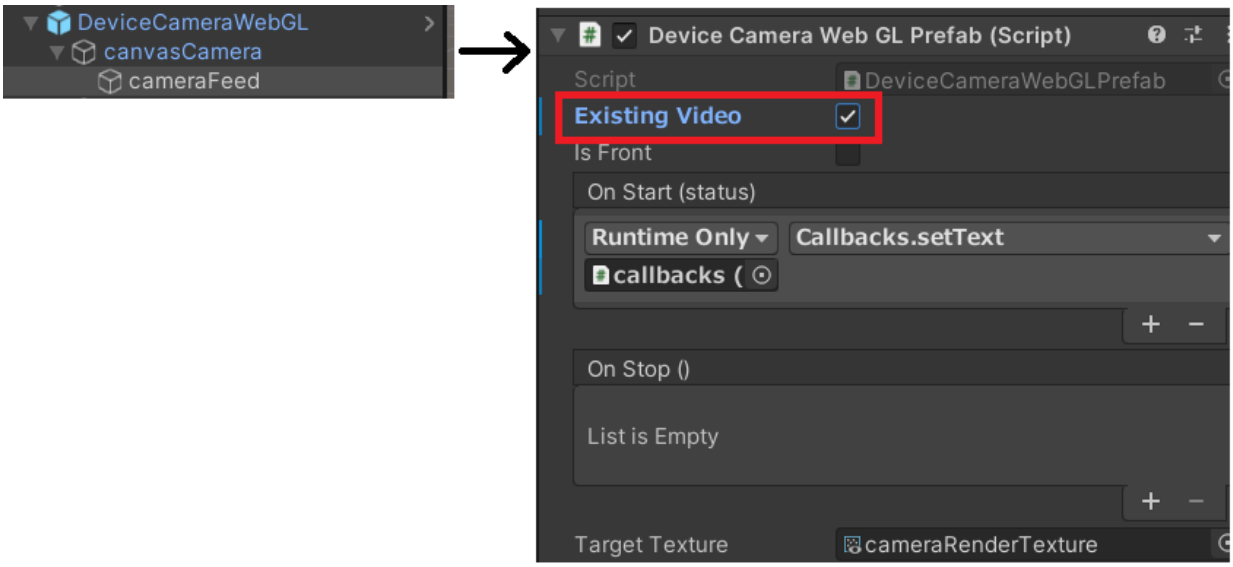
```
public static void assignCSSToVideo(string css = "pointer-events: none;
width: 1px; height: 1px; z-index: -1000; position: absolute; left: 0; top:
0;")
```

```
public static void assignPlayingEventToVideo(Action<status> callback =
null)

public static void assignResizeEventToVideo(Action<uint, uint> callback)
```

The first method's purpose is the same as the css parameter for the createVideoElement method. The second method hooks the "canUpdateTexture" property to the video element and takes as a parameter a callback that works the same as the callback for the start method. The third and last method is to run a callback when the video resizes, the same purpose as the first parameter of the createVideoElement method. You can check an example implementation of all this in the DeviceCameraWebGLPrefab.cs

The available prefab already makes all of this super easy. If you want to hijack an existing video element, just set the *existing video* field in the inspector to true, otherwise leave it false(default).



KNOWN ISSUES

1- If you think the webcam's color is "washed out", there are 2 ways you can solve this:

1. Go to Edit->Project Settings...->Player->Other Settings->Rendering->Color Space->

Select Gamma

2. If it's still washed out or you really want Linear Color space, go to Assets->MarksAssets->DeviceCameraWebGL->Plugins->DeviceCameraWebGL.jslib. Open the file and go to the function *DeviceCameraWebGL_updateTexture*. There are **2 lines** (one inside an *if*, another inside an *else*) like this *GLctx.texImage2D(GLctx.TEXTURE_2D, 0, GLctx.RGBA, GLctx.RGBA, GLctx.UNSIGNED_BYTE, video);*.

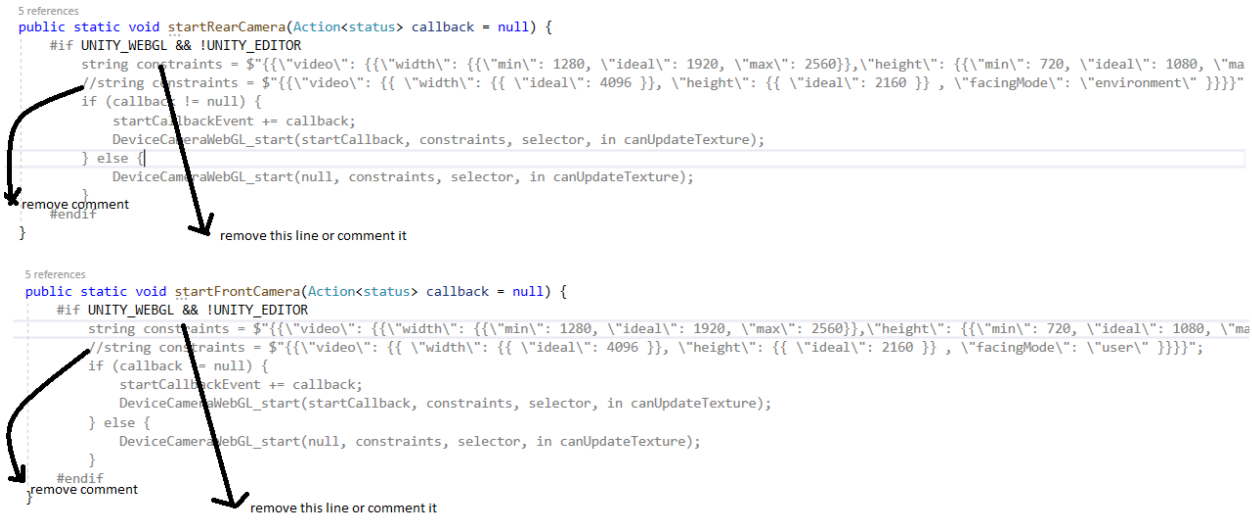
```
DeviceCameraWebGL_updateTexture: function(selector, texturePtr) {
    selector = UTF8ToString(selector);
    const video = document.querySelector(selector);
    if (!video) return false;
    if (!(video.videoWidth > 0 && video.videoHeight > 0)) return false;

    if (video.previousUploadedWidth != video.videoWidth || video.previousUploadedHeight != video.videoHeight)
        GLctx.deleteTexture(GL.textures[texturePtr]);
        GL.textures[texturePtr] = GLctx.createTexture();
        GL.textures[texturePtr].name = texturePtr;
        var prevTex = GLctx.getParameter(GLctx.TEXTURE_BINDING_2D);
        GLctx.bindTexture(GLctx.TEXTURE_2D, GL.textures[texturePtr]);
        GLctx.pixelStorei(GLctx.UNPACK_FLIP_Y_WEBGL, true);
        GLctx.texParameteri(GLctx.TEXTURE_2D, GLctx.TEXTURE_WRAP_S, GLctx.CLAMP_TO_EDGE);
        GLctx.texParameteri(GLctx.TEXTURE_2D, GLctx.TEXTURE_WRAP_T, GLctx.CLAMP_TO_EDGE);
        GLctx.texParameteri(GLctx.TEXTURE_2D, GLctx.TEXTURE_MIN_FILTER, GLctx.LINEAR);
        GLctx.texImage2D(GLctx.TEXTURE_2D, 0, GLctx.RGBA, GLctx.RGBA, GLctx.UNSIGNED_BYTE, video);
        GLctx.pixelStorei(GLctx.UNPACK_FLIP_Y_WEBGL, false);
        GLctx.bindTexture(GLctx.TEXTURE_2D, prevTex);
        video.previousUploadedWidth = video.videoWidth;
        video.previousUploadedHeight = video.videoHeight
    } else {
        GLctx.pixelStorei(GLctx.UNPACK_FLIP_Y_WEBGL, true);
        var prevTex = GLctx.getParameter(GLctx.TEXTURE_BINDING_2D);
        GLctx.bindTexture(GLctx.TEXTURE_2D, GL.textures[texturePtr]);
        GLctx.texImage2D(GLctx.TEXTURE_2D, 0, GLctx.RGBA, GLctx.RGBA, GLctx.UNSIGNED_BYTE, video);
        GLctx.pixelStorei(GLctx.UNPACK_FLIP_Y_WEBGL, false);
        GLctx.bindTexture(GLctx.TEXTURE_2D, prevTex)
    }
},
```

Replace them with *GLctx.texImage2D(GLctx.TEXTURE_2D, 0, GLctx.SRGB8_ALPHA8, GLctx.RGBA, GLctx.UNSIGNED_BYTE, video);*. This option is significantly slower though.

2- Hijacking a video element may still not give you full control over it such as switching the camera or stopping it. The reason is because usually the video element will be being controlled by something else that created it (another library), and as such it might resist giving up control of the video and override the commands of my plugin. The hijacking only guarantees that the contents of the video goes to a Unity texture so that you can do whatever you want with it, such as taking screenshots, recording, applying effects, etc.

3- If you think the camera resolution is low, you can go to the DeviceCameraWebGL.cs file, on the *startRearCamera* method and *startFrontCamera* method, there are commented out constraints, that you can uncomment and delete the other:



However please note that by doing this the experience can get significantly slower, specially on lower end devices. Alternatively, you can use the Start method directly with your own custom constraints if the above doesn’t work for you.