DESCRIPTION

This plugin allows you to record your webgl gameplay. You can take screenshots using Unity's API, but there is nothing to make a video recording. Other assets on the store allow you to record gameplay but they don't support webgl. This asset was made to fill that gap.

If you are using Unity 2020 or earlier, delete the file RecorderWebGL2021.jslib under MarksAssets\RecorderWebGL\Plugins. This file is only for Unity 2021 or higher. Don't touch the other files.

FEATURES

1. Supports recording webgl games.
2. Optionally records the gameplay with in-game audio. This supports audio from audio sources, from Unity's Video Player, and my own Video Player.
3. Optionally records the gameplay with microphone audio.
4. Optionally records the gameplay with microphone audio and in-game audio.
5. Configurable framerate.
6. You can start, stop, pause, resume.
7. You can download the recorded file in mp4 or webm (depending on the browser).
8. You can export the recorded video in various formats: DataURL, ObjectURL, BinaryData.
9. You can check if recording is supported by the browser.
10. It records in the resolution of Unity's game canvas.
11. It supports all major browsers: Chrome, Safari, Firefox, Opera, Edge.

KNOWN ISSUES/LIMITATIONS

1. Browsers on iPhone require iOS 14.5 or higher.
2. In game audio can only be recorded if it starts playing before you create the recording. If you know when the sound is supposed to play, add silent audio to the file until you want it to go off. If you really can't start playing the audio before you create the recording, there's a hack/workaround. Check out the last section of this documentation.
3. If you have a video with audio that is only supposed to play after the recording starts, you need to play the video beforehand and pause it (after a small delay, say 0.1s), then resume playback when the recording starts.
4. Having a single recording where you switch the source of a video player at runtime is not supported if the sources have audio. Videos with no volume(or muted) are fine. Otherwise, If you know all the videos that will be played, use multiple video players instead.
5. This plugin records Unity's game canvas(don't confuse it with Unity's UI canvas). This means that games that don't happen entirely in the game's canvas can't be recorded by this plugin. If you don't use other plugins you don't have to worry about this. Here are some examples:
   5a) Your camera in Unity has clear flag *depth only* or *solid color* with *alpha 0*. If you are doing this it's probably to show a webcam feed of another plugin you are using, or other

background HTML contents. In the specific case of the webcam feed, you can use DeviceCameraWebGL to stream the feed into Unity and make it recordable.
5b) You have foreground HTML content that overlays Unity's game canvas. A logo or a watermark for example. In this case, recreate the overlay inside Unity itself and remove the HTML overlay using something like display: none.
6. To record the microphone, you **must** host your game on an https server, http won't work.
7. Sometimes the recorded video is saved with the *.x-matroska* extension, instead of *.webm*. To solve this, When creating a new MediaRecorder, specify the *mimetype* with *webm*, like so: *RecorderWebGL.CreateMediaRecorder(createMediaRecorderCallback, new MediaRecorderOptions("video/webm;codecs=vp8,opus")*. This is a chromium issue.
8. If recording with the microphone or in-game audio, you must have at least one enabled audio source with an audio clip attached to it. It can be any dummy audio clip, and the audio source doesn't have to be set to play on awake or anything, it just needs to be enabled on the inspector. If you don't do this, the recording will fail and you may get a runtime error.

HOW TO USE

The flow of the plugin is this:

1. Create a media recorder
2. Start
3. Stop
4. Export to one of the available formats or download.

There is always one media recorder at any given time. Calling the method to create a media recorder multiple times will replace the previous media recorder. Every time you stop the recording, the recording is exported to a blob file that is used internally. This blob is later used to export the recording to a format that can be used in Unity.

If you have my ShareNSaveWebGL plugin, you can use this blob to share the video file using the following code snippet:

```
ShareNSaveWebGL.Share(callback, "Module.RecorderWebGL.mediaRecorderBlob");
```

There are many methods to convert the blob to other formats that you can use. You just need to make sure that the blob already exists before doing so. I will explain below.

CREATING A MEDIA RECORDER

This is the method signature for creating a media recorder. All parameters are optional.

```
public static void CreateMediaRecorder(Action<status> callback = null,
MediaRecorderOptions recorderOptions = null, bool microphone = true, bool
ingameAudio = true, double? frameRequestRate = null, string target = null)
```

Let's go through each parameter

1- Callback: This is the callback that will run after creating the media recording succeeds or fails. It takes a single status parameter that is the following enum

```
public enum status {
            Success = 0,
            NotAllowedError = 1,
            MediaRecorderAPIUnsupportedError = 2
        };
```

Success means the media recorder was successfully created. *NotAllowedError* is an error you get when trying to record using the microphone but the user didn't grant permission to use it. Please note that if you're using an iPhone and not using Safari, you need to enable microphone permissions in the system settings for the browser you are using, so only then you are requested to use the microphone in the target browser, otherwise you will always get a *NotAllowedError*. *MediaRecorderAPIUnsupportedError* is an error you get if the browser doesn't support recording.

2- recorderOptions: This parameter accepts a *MediaRecorderOptions* object, which has the following signature:

```
public                      class                      MediaRecorderOptions
{//https://www.w3.org/TR/mediastream-recording/#mediarecorderoptions-secti
on
            public readonly string mimeType;
            public readonly uint? audioBitsPerSecond;
            public readonly uint? videoBitsPerSecond;
            public readonly uint? bitsPerSecond;
```

```
                public MediaRecorderOptions(string mimeType = null, uint? audioBitsPerSecond = null, uint? videoBitsPerSecond = null, uint? bitsPerSecond = null) {
                this.mimeType = mimeType;
                this.audioBitsPerSecond = audioBitsPerSecond;
                this.videoBitsPerSecond = videoBitsPerSecond;
                this.bitsPerSecond = bitsPerSecond;
            }

        }
```

You can find about each option here: https://www.w3.org/TR/mediastream-recording/#mediarecorder options-section .You can completely ignore this parameter and let the browser choose the options for you. Or you can use this option and just use the null value for suboptions you don't want to configure. The default is null, meaning it lets the browser decide everything.

Here's an example:

```
RecorderWebGL.CreateMediaRecorder(createMediaRecorderCallback, new MediaRecorderOptions("video/webm", 128000));
```

This creates a media recorder with the mime type set to "video/webm" and audioBitsPerSecond to 128000. videoBitsPerSecond and bitsPerSecond are left to the browser to decide because they are null. Please note that even if you specify the options you want, it doesn't mean that the browser will follow it. Think of it as just a hint to the browser. It tells the browser what you want, and the browser might follow suit or not. For example, browsers on iPhone(and other Apple devices), at the time of writing, only export to mp4 files, not webm. Specifying webm will do nothing, it will still export to mp4. If the values for *videoBitsPerSecond*, *audioBitsPerSecond*, and *BitsPerSecond* are too high or too low, the browser will clamp them. *videoBitsPerSecond*, *audioBitsPerSecond*, and *BitsPerSecond* seem to be ignored by Chrome and are always 0 at the time of writing, regardless of the values you provide. Firefox and Safari use these values, however.

Here's an example in javascript for reference.

3- microphone: If this parameter is true it means you want to record the game with microphone audio, false if not. Default is true.

4- ingameAudio: If this parameter is true it means you want to record the game with audio from the game. This supports audio from audio sources, from Unity's Video Player, and my own Video Player. Default is true. You **must** have audio playing before starting to record using this option, otherwise it won't work, it won't even trigger the start callback. See last section of this document.

5- frameRequestRate: The framerate you want the browser to record your gameplay at, using the captureStream method behind the scenes. The value must be positive, if you provide a negative value it is simply converted to positive instead of throwing an error. The browser might follow suit or not. Letting the value as null, which is the default, means a new frame will be captured each time the canvas changes. If you set the value to 0, you will need request the frames manually using the method

```
public static void RequestFrame()
```

For example

```
void Start() {
            RecorderWebGL.CreateMediaRecorder(createMediaRecorderCallback,
null, true, true, 0);//notice that the last parameter is 0
    }

    void Update() {
            RecorderWebGL.RequestFrame();//and here I manually update the
frames
    }
```

6- target: The target that you want to record. If you want to record the game's canvas, leave this *null*(default), which is 99% of the use cases. However, you may want to record some other HTML element. In which case, you pass a string here that will be evaluated by querySelector. Please note that internally, the *captureStream* javascript method is used. When the target is a canvas, which is the default case (Unity's game canvas), it supports all browsers. However, if you try to record something else, like a video element, the browser compatibility is more limited (in short, at the time of writing, browsers running on Apple devices don't support it). An example usage of this parameter is if you want to record the webcam feed when using DeviceCameraWebGL, but you don't want to **display** the webcam feed, that is, you want to record the webcam feed in the background. In this case, you could pass the value "video"(without quotes) for this parameter and it will record the webcam feed, and in Unity you can simply not display the webcam feed and show something else.

STARTING THE MEDIA RECORDER

This is the method signature for starting the media recorder

```
public static void Start(Action callback = null, uint? timeslice = null)
```

There are only 2 parameters:

1. Callback: a parameterless callback that runs after starting the recording. Even though calling start seems to be immediate, it's actually an asynchronous call. So use the callback if you absolutely need to do something after the recording has started, otherwise if perfect precision isn't needed you can ignore this parameter.
2. Timeslice: You can read about this parameter here.

STOPPING THE MEDIA RECORDER

There are several method overrides for stopping the media recorder.

1-

```
public static void Stop(Action callback = null, bool save = false, string
fileName = null)
```

Like the start method, it has a parameterless callback that runs after the media recorder stopped and the recorded blob is available. Unlike the start method, you almost certainly want to use the callback here to do something with the recorded file after it stopped and the blob is available. The *save* parameter is to tell if you want the recorded file to be immediately downloaded, and the *fileName* parameter is to download the file with the name you want(if no name is provided, it downloads with the name “file.mp4” or “file.webm”, depending on the browser).

The example scene uses this version of the Stop method. After you call stop, a download button is enabled so you can download the recorded file.

2-

```
public static void Stop(Action<string> dataURLorObjectURLCallback, bool
isDataURL = true)
```

This method stops the recording and returns a DataURL or ObjectURL through the *dataURLorObjectURLCallback* parameter, which takes a string as input. If dataURL is true, the method returns a DataURL, otherwise it returns an objectURL. This method is useful if you want to play the recording in the game itself, if you want the user to be able to preview the recording, for example. If you have my VideoPlayerWebGL plugin, you can set the video source to be the dataURL or objectURL, and the recorded video will play inside your game(this probably works with Unity's Video Player as well). If you use the objectURL, don't forget to call

```
public static void RevokeObjectURL(string url)
```

afterwards. See https://developer.mozilla.org/en-US/docs/Web/API/URL/revokeObjectURL.

Another use case for this method is opening the recorded video in a new tab to preview it, instead of previewing it in the game itself. You just need to use my LaunchURLWebGL free asset to do this. It will only work with the object url in this case though.

3-

```
public static void Stop(Action<byte[], int> byteArrayCallback)
```

This version of Stop requires a callback that takes a byte array as input (and its size, in bytes, as the second argument). You will get the binary data of the recorded video file. This could be useful if you want to manipulate the video in Unity after recording it and then upload it to an external server, for example.

In addition to the Stop methods, you also have the following ones that serve the same purpose, except for stopping the recording. You can use them if you want to export the recorded video to multiple formats instead of just one. After you call stop to get the video in a specific format, and then later on you want to export to another format the same recording, you'd use one of the following methods.

```
public static void ToDataURL(Action<string> dataURLCallback)

public static void ToObjectURL(Action<string> objectURLCallback)

public static void ToByteArray(Action<byte[], int> byteArrayCallback)
```

DOWNLOADING RECORDED FILE

Call the following method

```
public static void Save(string fileName = null)
```

Or you can use the first version of the stop method with the save parameter set to true. Like the first version of the stop method, fileName is the name you want for the downloaded recording file. (if no name is provided, it downloads with the name "file.mp4" or "file.webm", depending on the browser).

You should call this method on the callback of the stop method, to guarantee that you have the file available(the recorded blob, which is then downloaded). See the example scene. It enables the download button on the Stop's callback, so the blob is guaranteed to be there and ready to be downloaded or exported to another format.

PAUSING AND RESUMING RECORDING

You can pause and resume a recording using the following methods, respectively:

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

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

The difference between pausing and stopping is that pausing does not create a blob that you can use. After pausing you can resume at any time to continue recording on the same file. Both of them have a parameterless callback that you can use if absolute precision is necessary. Otherwise you can ignore the callback because resume and pause happen almost instantly.

OTHER METHODS

1- GetRecordingFileExtension

```
public static string GetRecordingFileExtension()
```

This method returns the file extension of the recorded file. For example, “webm” or “mp4”. Since you don’t know for sure what format the browser will use to record the game, you can use this method for the file name when you want to save the file. It returns *null* after the recording stops and it actually failed and didn't record the file. For example

```
private void stopcallback() {RecorderWebGL.Save("superspecialfile." +
RecorderWebGL.GetRecordingFileExtension());}
```

You need to call this method in the stop callback.

2- IsTypeSupported

```
public static bool IsTypeSupported(string type)
```

If you want to know in advance what format the browser supports, you can check the mime type. For example:

```
IsTypeSupported(“video/webm”);//check if it supports recording in webm
format

IsTypeSupported(“video/mp4”);//check if it supports recording in mp4
format
```

It returns true if the mime type is supported, false if not.

4- Destroy

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

Destroys the media recorder and the current blob if any. After calling this method, you won’t be able to use Start again until you create a new media recorder. Accepts a parameterless callback that you can use to do something right after the media recorder is destroyed

5- GetState

```
public static RecordingState GetState()
```

Returns the current recording state, which is represented by the following enum

```
public enum RecordingState {mediaRecorderDoesntExist = -1,inactive =
0,recording = 1,paused = 2,stopped = 3};
```

It will return *mediaRecorderDoesnExist* if you call this method before creating a media recorder or after destroying it. It will return *inactive* if you created the media recorder but didn't start recording. It will return *recording* if you are currently recording. It will return *paused* if the recording is currently paused. It will return *stopped* if the recording is currently stopped.

6- GetCanvasWidth

```
public static uint GetCanvasWidth()
```

It returns the width of Unity's canvas. This is the same width of the recorded video file, assuming the width of the game doesn't change.

7- GetCanvasHeight

```
public static uint GetCanvasHeight()
```

It returns the height of Unity's canvas. This is the same height of the recorded video file, assuming the height of the game doesn't change.

8- GetVideoBitsPerSecond

```
public static uint GetVideoBitsPerSecond()
```

It returns the VideoBitsPerSecond being used by the media recorder, which might differ from what you provided in the options.

9- GetAudioBitsPerSecond

```
public static uint GetAudioBitsPerSecond()
```

It returns the AudioBitsPerSecond being used by the media recorder, which might differ from what you provided in the options.

10- CanRecord

```
public static bool CanRecord()
```

Returns true if the browser supports recording the game, false if not.

11- Discard

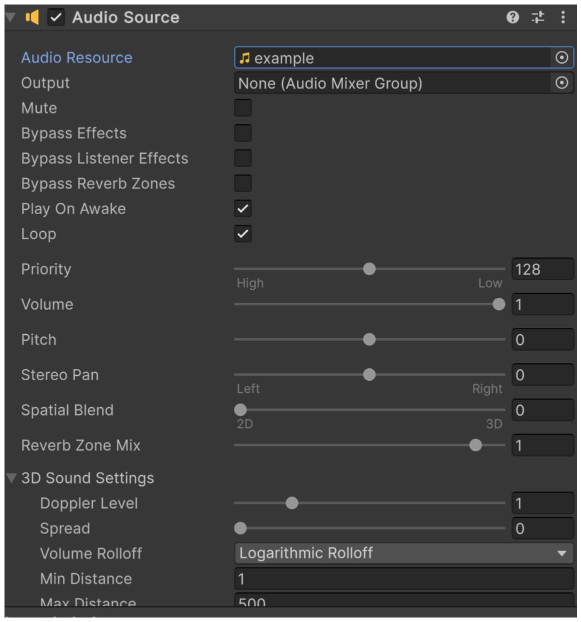
```
public static void Discard()
```

Destroys the blob, if any. Useful if you want to free some memory after exporting and/or downloading the recorded file.

RECORDING IN-GAME AUDIOS

If you want to record in game audios that are not within the limitations of the plugin, you can follow the hack/workaround below.

1. Leave the audio source with Play On Awake and Loop options set, and pitch set to 0



Audio Source

Audio Resource	example
Output	None (Audio Mixer Group)
Mute	
Bypass Effects	
Bypass Listener Effects	
Bypass Reverb Zones	
Play On Awake	✓
Loop	✓
Priority (High – Low)	128
Volume	1
Pitch	0
Stereo Pan (Left – Right)	0
Spatial Blend (2D – 3D)	0
Reverb Zone Mix	1
3D Sound Settings	
Doppler Level	1
Spread	0
Volume Rolloff	Logarithmic Rolloff
Min Distance	1
Max Distance	500

2. When you actually have to play the audio, set the pitch to 1. (audioSource.pitch = 1).
3. When the audio starts looping, set the pitch to 0. Set the pitch to 1 when you need to reuse, and so on and so forth. **Don't** call the pause method on the audioSource, it won't work and the recording will fail.