

Description

Next Gen Recorder is a video recording library for iOS, tvOS and macOS platforms. It is written from scratch by one of the Everyplay developers who thinks that world still needs a high performance recording system. Everyplay was a gameplay video recording and sharing system with a community of millions of users that got unfortunately shutdown in October 2018.

Next Gen Recorder takes the gameplay recording to the next level.

Key Features

- high performance, 60 fps recording
- fully asynchronous, will not block your game threads
- record any 2d texture, not just the screen
- highlight recording, export the whole gameplay or just the most important parts
- custom watermark, not available in Free version
- easily save the video to the photos or share it through the native share sheet
- session support, record multiple gameplay videos and choose later what video to share
- full access to the recorded video so you may also implement your own sharing/upload code when needed
- custom bit rate, frame rate and video size support
- supports Metal and OpenGL renderers
- support textures in gamma and linear colorspace
- supports Scriptable Render Pipeline (SRP, HDRP, URP)
- supports Unity 5.6.6 and up
- also works in Unity editor on macOS

Minimum OS versions for recording

- iOS 8.0
- tvOS 9.0
- macOS 10.11

Minimum OS versions for sharing

- iOS 8.0 (Photos and ShareSheet)
- tvOS 10.0 (Photos)
- macOS 10.11 (Save video dialog)

Integration

Next Gen Recorder is capable of recording any 2d texture which basically means that you can record anything, whole screen, part of the screen or something that is not even seen on the screen. You can

integrate Next Gen Recorder either automatically or manually. With automatic integration only the main camera gets recorded. To have full control what to record you must integrate manually.

Important

- **All the recording sessions and exported videos are automatically removed when the application is closed.** If you are not sharing the exported video through the sharing API and you want to keep the file, please copy or move it to your preferred location.
- **Next Gen Recorder automatically pauses the recording when application enters background or focus is lost.** So don't implement your own pause/resume logic when entering/returning from the background or when the focus is changed.
- **Automatic integration does not support Scriptable Render Pipeline (SRP).** For SRP you should either do a custom integration or use the Metal Screen Recorder.
- **Always set `Application.targetFrameRate` and Next Gen Recorder's `TargetFrameRate`.** Otherwise you might get weirdly keyframed video.

Automatic integration

By default the integration happens automatically. This works for most simple cases where you have only one camera to be recorded. The video produced with the automatic integration does not include the UI by default. In case you also want to record the UI, please see Recording the UI section of this document. Automatic integration is executed when you call `StartRecording` or `PrepareRecording` for the first time and you have not integrated manually.

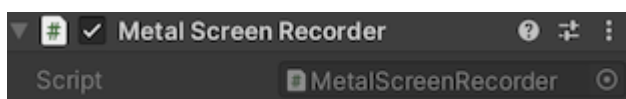
Manual integration

There are multiple ways to integrate Next Gen Recorder. Use Metal Screen Recorder if you are only going to support Metal renderer and you want to record the whole screen with UI. It is the easiest and most performant way to integrate.

If you need to both OpenGL and Metal you can use Virtual Screen. Virtual Screen is a component that is simply dropped on to a camera and then it forces the camera to render to a buffer that Next Gen Recorder can record.

Integration using the Metal Screen Recorder

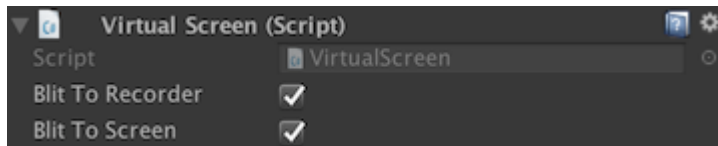
Metal Screen Recorder allows you to record everything seen on the screen and it also supports Scriptable Render Pipeline (SRP). To use the Metal Screen Recorder, just drop it on top of any game object in your scene.



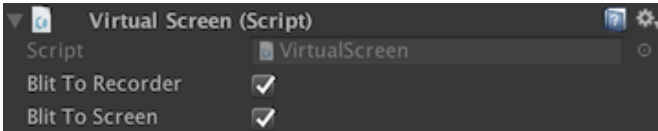
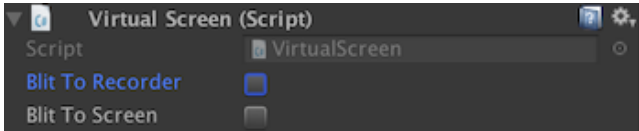
MetalScreenRecorder requires Metal renderer and Unity 2017.3 or greater.

Integration using the Virtual Screen

If you have only one camera it can be recorded by simply adding **Virtual Screen** component on it and making sure both **Blit to Recorder** and **Blit to Screen** are checked. Virtual Screen does not support SRP. For SRP you should either do a custom integration or use the Metal Screen Recorder.



If you have multiple cameras, add **Virtual Screen** to all the cameras that are rendering to the SCREEN and check **Blit to Recorder** and **Blit to Screen** on the LAST camera ONLY. Last camera is the camera that has the greatest **Depth**. **Depth** defines the camera rendering order.

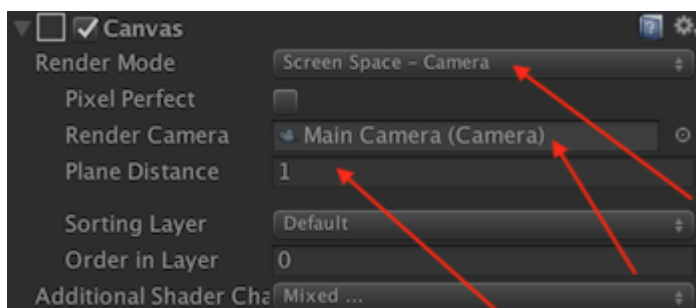
Last Camera	Other Cameras
	

Recording the UI

By default the Unity UI is rendered on a separate layer after the camera and that why it is not captured to the video. There are multiple ways to resolve this. In case you are using image effects and you don't want to apply those effects to the UI you will need a secondary camera. If you are not using image effects you don't need any additional cameras.

Including the UI when using a single camera only

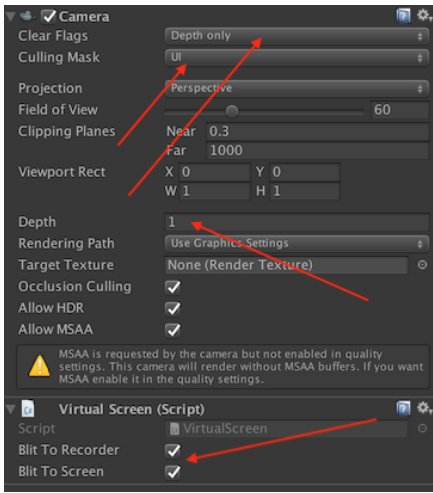
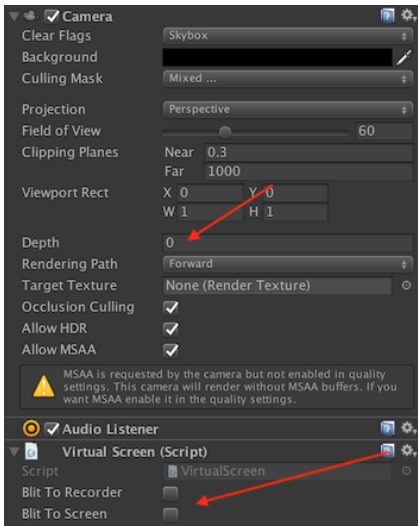
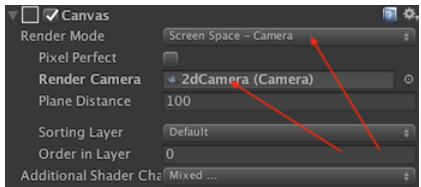
To render the UI with a camera instead of as a separate layer you must set the canvas **Render Mode** from **Screen Space - Overlay** to **Screen Space - Camera**. You must also set the canvas **Render Camera** to your **Main Camera** and **Plane Distance** to **1**. After these changes the recording will also contain the UI. If not, make sure that your camera **Culling Mask** includes the **UI** layer.



Including the UI by using a separate UI camera

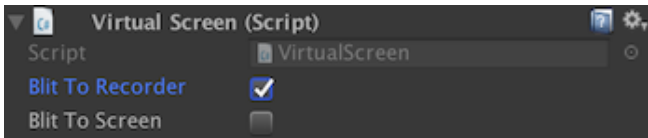
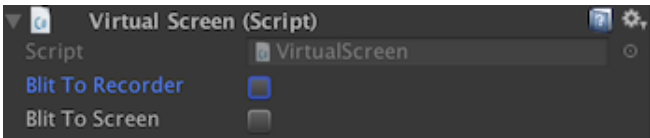
Another option to include the the UI is to create a separate camera for the UI, set it's **Culling Mask** to **UI**, **Depth** to greater than your 3d camera's **Depth** and **Clear Flags** to **Depth Only**. After that set your canvas **Render mode** to **Screen Space - Camera** and **Render Camera** to the camera that you just created. Lastly add **Virtual Screen** to your 2d camera and check both **Blit to Recorder** and

Blit to Screen on it and make sure they both are unchecked for the 3d camera. Also make sure your 3d camera does not contain the **UI** in **Culling Mask**.

2d/UI Camera	Main Camera	2d Canvas
		

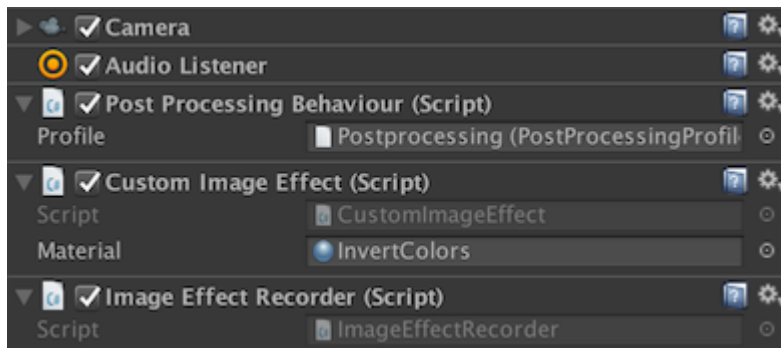
Recording something else

Sometimes you might want to record something that is not visible on the screen at all, for example a goalie camera for a football game or a 3rd person observer camera for a 1st person game. In these cases you just add **Virtual Screen** component to all the cameras you want to record with **Blit to Screen** unchecked and **Blit to Recorder** checked on the LAST camera.

Last Camera	Other Cameras
	

Integration using the Image Effect Recorder

In some special cases you might want to record a camera after or before some certain Image Effect. For this you can use a component called Image Effect Recorder. Just drag it onto your camera. It's good to remember that the order of the components matter. If you want to record all the image effects you must make sure that Image Effect Recorder is the last component on the component list. Image Effect Recorder does not support SRP. For SRP you should either do a custom integration or use the Metal Screen Recorder.



Integration using the Metal Camera Recorder

Metal Camera Recorder allows you to record everything that a camera renders when using Metal renderer and Built-In Render Pipeline. It does not Scriptable Render Pipeline (SRP). To record a camera, drop the Metal Camera Recorder component on top of it.

MetalScreenRecorder requires Metal renderer and Unity 2017.3 or greater.

Custom integration

If none of the above integration methods works for your project you can create your own recording pipeline by inheriting from VideoRecorderBase.

```
public class MyRecorder : Recorder.VideoRecorderBase
{
    ...

    // Set the texture to be recorded

    if (RecordingTexture != myTextureToRecord)
    {
        RecordingTexture = myTextureToRecord;
    }

    ...

    // Blit the actual recording texture to the current encoder target, executed in graphics thread
    // Be sure that the recording texture is available and is not yet discarded, caution with temporary textures

    BlitRecordingTexture();

    ...

    // Or if you use command buffers

    void Awake() {
        CommandBuffer myCommandBuffer = new CommandBuffer();
        CommandBufferBlitRecordingTexture(myCommandBuffer) // Will add custom blit to the c
```

```

ommand buffer
    }

    // Or if you want to capture current render target (Metal only)

    CaptureMetalRenderTarget();

    // Or if you use command buffers

    CommandBufferCaptureMetalRenderTarget(myCommandBuffer);

    ...
}

```

- **IMPORTANT:** When you set the **RecordingTexture** in a custom integration be sure the **texture still exists (and it is not released or discarded) when the actual blit happens**. Setting the **RecordingTexture** only sets the recording source for the next blit and the actual blit happens in rendering thread. At the moment using temporary textures as Next Gen Recorder source (**RenderTarget.GetTemporary**) is not recommended but if you use them please release (**RenderTarget.ReleaseTemporary**) them after everything is rendered. Same applies for discarding render textures. Otherwise you might just get black screen in your recording.

Example – Custom size off-screen camera recording

Version 0.9.10.1 or greater required.

```

using UnityEngine;
using UnityEngine.Rendering;
using pmjo.NextGenRecorder;

[RequireComponent(typeof(Camera))]
public class OffScreenCameraRecorder : Recorder.VideoRecorderBase
{
    public int videoWidth = 640;
    public int videoHeight = 480;

    private RenderTexture m_RenderTexture;
    private Camera m_Camera;
    private CommandBuffer m_CommandBuffer;

    void Awake()
    {
        m_RenderTexture = new RenderTexture(videoWidth, videoHeight, 24, RenderTextureFormat.Default);
        m_RenderTexture.Create();

        m_Camera = GetComponent<Camera>();
        m_Camera.targetTexture = m_RenderTexture;

        if (!Recorder.IsSupported)

```

```

{
    Debug.LogWarning("Next Gen Recorder not supported on this platform");
    return;
}

RecordingTexture = m_RenderTexture;

m_CommandBuffer = new CommandBuffer();
CommandBufferBlitRecordingTexture(m_CommandBuffer);
m_Camera.AddCommandBuffer(CameraEvent.AfterEverything, m_CommandBuffer);
}
}

```

Usage

Next Gen Recorder is really easy to use. This section guides you through all the properties and functions that control the recording and sharing process.

Controlling the recording

To start the recording you just call `StartRecording` and `StopRecording` to stop it. To pause the recording call `PauseRecording` and `ResumeRecording` to continue the recording. Pausing the recording might be useful for example when the user enters the menu during the game.

Example

```

public void OnLevelStarted()
{
    Recorder.StartRecording();
}

public void OnMenuShown()
{
    if (Recorder.IsRecording)
    {
        Recorder.PauseRecording();
    }
}

public void OnMenuHidden()
{
    if (Recorder.IsPaused)
    {
        Recorder.ResumeRecording();
    }
}

public void OnLevelEnded()

```

```
{  
    Recorder.StopRecording();  
}
```

Events

All the calls to the Next Gen Recorder are asynchronous so they will exit immediately when you call them and continue on the background. That why it is highly recommended to rely on events like `RecordingStarted`, `RecordingStopped`, `RecordingPaused`, `RecordingResumed` instead of getters like `IsRecording` since it may result false negative if you call it immediately after `StartRecording`.

Example

```
void OnEnable()  
{  
    Recorder.RecordingStarted += RecordingStarted;  
    Recorder.RecordingPaused += RecordingPaused;  
    Recorder.RecordingResumed += RecordingResumed;  
    Recorder.RecordingStopped += RecordingStopped;  
}  
  
void OnDisable()  
{  
    Recorder.RecordingStarted -= RecordingStarted;  
    Recorder.RecordingPaused -= RecordingPaused;  
    Recorder.RecordingResumed -= RecordingResumed;  
    Recorder.RecordingStopped -= RecordingStopped;  
}  
  
void RecordingStarted(long sessionId)  
{  
    Debug.Log("Recording started, session id " + sessionId);  
}  
  
void RecordingPaused(long sessionId)  
{  
    Debug.Log("Recording paused, session id " + sessionId);  
}  
  
void RecordingResumed(long sessionId)  
{  
    Debug.Log("Recording resumed, session id " + sessionId);  
}  
  
void RecordingStopped(long sessionId)  
{  
    Debug.Log("Recording stopped, session id " + sessionId);  
}
```


All the events have session id parameter to identify the recording session.

Exporting the video

To export a gameplay video you must subscribe to **RecordingExported** event and call **ExportRecordingSession** function. This will start an asynchronous export process that calls **RecordingExported** event when completed.

Example

```
void OnEnable()
{
    Recorder.RecordingExported += RecordingExported;
}

void OnDisable()
{
    Recorder.RecordingExported -= RecordingExported;
}

void ExportLastRecording()
{
    long lastSession = Recorder.GetLastRecordingSession();

    Recorder.ExportRecordingSession(lastSession);
}

void RecordingExported(long sessionId, string path, Recorder.ErrorCode errorCode)
{
    if (errorCode == Recorder.ErrorCode.NoError)
    {
        Debug.Log("Recording exported to " + path + ", session id " + sessionId);
    }
    else
    {
        Debug.Log("Failed to export recording, error code " + errorCode + ", session id " +
sessionId);
    }
}
```

Highlights

Sometimes you might want to share only the best parts of the gameplay. This can be done by marking highlight events like killing a boss during the gameplay. Highlights can be marked by calling **SetHighlight** when the actual highlight event occurs. You may specify a priority for the event and the duration in seconds that you want to store to the highlight video before the event and after the event. **SetHighlight** returns true if highlight was successfully set.

You may also specify an identifier for the highlight. You could for example group all kills with the same highlight identifier to be able to export only kills or use an unique identifier for all events and hand pick which highlights you want to export.

Example

```
int priority = 3;
float preSeconds = 2.0f;
float postSeconds = 3.0f;
// long highlightId = 12345678;
Recorder.SetHighlight(priority, preSeconds, postSeconds);
// Recorder.SetHighlight(priority, preSeconds, postSeconds, highlightId);
```

Exporting a highlight video

To export a highlight video you must subscribe to `HighlightRecordingExported` event and call `ExportRecordingSessionHighlights` function. This will start an asynchronous export process that calls `HighlightRecordingExported` event when completed. You may also define target length for the highlight video so Next Gen Recorder will add highlights starting with the greatest priority to the video until the target length is achieved. If you don't define a target length all highlights will be added to the video.

Example

```
void OnEnable()
{
    Recorder.HighlightRecordingExported += HighlightRecordingExported;
}

void OnDisable()
{
    Recorder.HighlightRecordingExported -= HighlightRecordingExported;
}

void ExportLastHighlightRecording()
{
    long lastSession = Recorder.GetLastRecordingSession();
    float targetVideoLength = 15.0f;

    // Export all highlights
    // Recorder.ExportRecordingSessionHighlights(lastSession, targetVideoLength);

    // Export highlights until video is around 15 seconds
    Recorder.ExportRecordingSessionHighlights(lastSession, targetVideoLength);

    // Export selected highlights
    // long[] highlightIdentifiers = new long[] { 2323, 5234 };
```

```

    // Recorder.ExportRecordingSessionHighlights(lastSession, highlightIdentifiers);
}

void HighlightRecordingExported(long sessionId, string path, Recorder.ErrorCode errorCode)
{
    if (errorCode == Recorder.ErrorCode.NoError)
    {
        Debug.Log("Highlight recording exported to " + path + ", session id " + sessionId);
    }
    else
    {
        Debug.Log("Failed to export highlight recording, error code " + errorCode + ", session id " + sessionId);
    }
}
}

```

Exporting a thumbnail

To export a thumbnail from a recording session you must subscribe to `ThumbnailExported` event and call `ExportRecordingSessionThumbnail` function. This will start an asynchronous thumbnail generation process that calls `ThumbnailExported` event when completed.

You must also set the maximum side width for the generated thumbnail and also the type of the thumbnail as a parameter. Only PNG and JPG types are supported.

Example

```

void OnEnable()
{
    Recorder.ThumbnailExported += ThumbnailExported;
}

void OnDisable()
{
    Recorder.ThumbnailExported -= ThumbnailExported;
}

void ExportLastRecordingThumbnail()
{
    long lastSession = Recorder.GetLastRecordingSession();
    int maxSideWidth = 256;

    Recorder.ExportRecordingSessionThumbnail(lastSession, maxSideWidth, Recorder.ThumbnailType.JPG);
}

void ThumbnailExported(long sessionId, string path, Recorder.ErrorCode errorCode)
{
    if (errorCode == Recorder.ErrorCode.NoError)

```

```

{
    Debug.Log("Thumbnail exported to " + path + ", session id " + sessionId);
}
else
{
    Debug.Log("Failed to export a thumbnail, error code " + errorCode + ", session id "
+ sessionId);
}
}

```

Preparing the recording session

Sometimes you need to be able to start the recording as fast as possible. To make the `StartRecording` call faster you can prepare the recording session by calling `PrepareRecording`. This will make sure all the required buffers are ready when actually starting the recording. If you change any of the video properties like frame rate between `PrepareRecording` and `StartRecording` they will get ignored until the current recording session is stopped.

Example – Preparing the recording session

```
Recorder.PrepareRecording();
```

Controlling the video frame rate

You may control the video frame rate with `TargetFrameRate` and `FrameSkipping`. For best results you should enable `FrameSkipping` when your target frame rate is different than application's target frame rate and disable it when it is the same. If your application runs in different frame rates on different devices or it is not running 60fps it is recommended to set the frame rate like in below example. By default `TargetFrameRate` is 60 and `FrameSkipping` false.

Example – Record using application frame rate

```
Recorder.TargetFrameRate = Application.targetFrameRate;
Recorder.FrameSkipping = false;
```

Sometimes you might want to record the video with lower frame rate to make the video file smaller or to save some cpu time for something else.

Example – Record half frame rate, 60 fps -> 30 fps

```

if(Application.targetFrameRate > 30) {
    Recorder.TargetFrameRate = Application.targetFrameRate / 2;
    Recorder.FrameSkipping = true;
}

```

```
else {  
    Recorder.TargetFrameRate = Application.targetFrameRate;  
    Recorder.FrameSkipping = false;  
}
```

IMPORTANT: By default Unity does not set value for `Application.targetFrameRate` and it is -1 even it really runs 30fps. Always set `Application.targetFrameRate` by yourself and remember to set Next Gen Recorder's `TargetFrameRate` also.

Controlling the video frame time

By default the recorder is in real time mode that means the video frame time is captured when the actual frame blit to the recorder happens. In some rare cases you might want to use a constant frame rate instead. To do this you can disable real time recording by setting `RealtimeRecording` to false. When real time recording is disabled the frame delta time is $1.0 / \text{Recorder.TargetFrameRate}$. It is recommended to not do more than one recorder blit during application frame or otherwise the encoder might choke and or skip the appended frame. Providing custom frame times is not currently supported.

Downscaling the video

By default Next Gen Recorder records using the source texture size that is usually the screen size. Sometimes you might want to downscale the video size to decrease the file size or memory usage or to free cpu time for something else. Use `VideoScale` to set a video scaling factor. The scaling factor is clamped between 0.1 and 1.0.

Example – Downscale 50%

```
Recorder.VideoScale = 0.5f;
```

Many of the new devices have a big native resolution. Recording the full resolution may result low framerate and really huge video files. Some devices like iPad Pro have a resolution that is so big that it simply cannot be recorded 60 fps since the encoder cannot handle that much data in realtime. It is recommended to downscale the resolution or decrease the target framerate to avoid stuttering video.

Example – Downscale 50% if width (or height) is greater than 1920

```
if (Mathf.Max(Screen.width, Screen.height) > 1920) {  
    Recorder.VideoScale = 0.5f;  
}  
else {  
    Recorder.VideoScale = 1.0f;  
}
```

Flipping the video vertically

Sometimes your recording texture might be upside down and you want to flip the video vertically. Keep in mind that texture coordinates are different on OpenGL and Metal and in some cases you might just want flip the video only on Metal or vice versa. To flip the video vertically set `VerticalFlip` property to true.

```
Recorder.VerticalFlip = true;
```

Using custom audio or video bit rate

By default Next Gen Recorder uses Kush Gauge formula with motion factor 1.0 for video bit rate calculation. Sometimes you might want to fine-tune the bit rate for smaller files or for better quality. This can be easily done by using the `CustomVideoBitrate` and `CustomAudioBitrate` callbacks. When a new recording is started, Next Gen Recorder will call the callbacks with current video and audio parameters. Use the parameters to calculate and return your own bit rate.

Example – Set custom video and audio bit rate

```
void Start()
{
    Recorder.CustomVideoBitrate = MyCustomVideoBitrate;
    Recorder.CustomAudioBitrate = MyCustomAudioBitrate;
}

private int MyCustomVideoBitrate(long sessionId, int width, int height, int frameRate)
{
    // float motionFactor = 0.5f; // Low
    float motionFactor = 1.0f; // Medium (default)
    // float motionFactor = 2.0f; // High
    // float motionFactor = 4.0f; // Super
    return Mathf.RoundToInt((frameRate * width * height * motionFactor * 0.07f) * 0.001f) * 1000;
}

private int MyCustomAudioBitrate(long sessionId, int sampleRate, int channelCount)
{
    return 64000 * channelCount;
}
```

Custom watermark

You can use a custom watermark texture which will be rendered on top of the video. Currently it is always rendered on top right corner but there will be an API to position it in the future. Use the `WatermarkTexture` property to set it.

Example

```
public Texture2D myWaterMarkTexture;

void Start()
{
    Recorder.WatermarkTexture = myWaterMarkTexture;
}
```

NOTE: Customizable Watermark feature does not exist in the Free version of Next Gen Recorder.

Audio recording

By default the audio that goes through the AudioListener is recorded. You can disable audio recording by setting `RecordAudio` to false.

Example – Disabling audio recording

```
Recorder.RecordAudio = false;
```

Disk space

For recording to succeed there must be enough disk space to store the video. To avoid a video file corruption you can set `MinDiskSpaceRequired` required. If there is not enough disk space, the recording will not start. `MinDiskSpaceRequired` is expressed in megabytes and is 300 by default. You may also call `IsReadyForRecording` to know if starting the recording is possible.

```
Recorder.MinDiskSpaceRequired = 500;
```

IMPORTANT: All the recording sessions and exported videos are removed when the application is closed. In case the application has been killed by the system the recording sessions are removed when Next Gen Recorder is initialized for the next time. If you are not sharing the exported video through the sharing API and you want to keep the file, please copy or move it to your preferred location.

Color space

By default Next Gen Recorder assumes that the source texture colorspace is Gamma. Use `ColorSpace` property to change it to Linear when needed.

```
// Recorder.ColorSpace = ColorSpace.Gamma;
Recorder.ColorSpace = ColorSpace.Linear;
```

Sharing the video

You can share the video using the native share sheet by calling `ShowShareSheet` or save it to the photo album by calling `SaveToPhotos`. The native share sheet is only supported on iOS. To use the share sheet or photo album feature you must first obtain a path to the video using `ExportRecordingSession` or `ExportRecordingSessionHighlights`.

When using the share sheet you can subscribe to `ShareSheetClosed` event to get notified when the share sheet is closed. This is useful for example when you open a specific sharing UI and you want to know when to hide it.

SaveToPhotos will copy the video to the Camera Roll by default. If you want to copy the video to some specific album you can give the album name as the second parameter. In case the album does not exist, it will be created.

On macOS `ShowShareSheet` and `SaveToPhotos` will open a save to folder dialog instead. On macOS you can also use `SaveToSelectedFolder` that will allow you to give a preferred file name for the video.

```
void OnEnable()
{
    Recorder.RecordingExported += RecordingExported;
    Sharing.ShareSheetClosed += ShareSheetClosed;
    Sharing.SavedToPhotos += SavedToPhotos;
    Sharing.SavedToSelectedFolder += SavedToSelectedFolder;
}

void OnDisable()
{
    Recorder.RecordingExported -= RecordingExported;
    Sharing.ShareSheetClosed -= ShareSheetClosed;
    Sharing.SavedToPhotos -= SavedToPhotos;
    Sharing.SavedToSelectedFolder -= SavedToSelectedFolder;
}

void RecordingExported(long sessionId, string path, Recorder.ErrorCode errorCode)
{
    if (errorCode == Recorder.ErrorCode.NoError)
    {
#ifdef UNITY_TVOS
        Sharing.SaveToPhotos(path);
        // Sharing.SaveToPhotos(path, "My Awesome Album");
#elif UNITY_IPHONE
        Sharing.ShowShareSheet(path);
        // Sharing.SaveToPhotos(path);
        // Sharing.SaveToPhotos(path, "My Awesome Album");
#elif UNITY_STANDALONE_OSX || UNITY_EDITOR_OSX
        // Sharing.SaveToSelectedFolder(path);
        Sharing.SaveToSelectedFolder(path, "MyAwesomeGame.mp4");
#endif
    }
}
```



```

void ShareSheetClosed() {
    Debug.Log("Share sheet was closed");
}

private void SavedToPhotos(Sharing.ErrorCode errorCode)
{
    if (errorCode == Sharing.ErrorCode.NoError)
    {
        Debug.Log("Successfully saved the video to photos");
    }
    else
    {
        Debug.Log("Failed to save the video to photos, error: " + errorCode.ToString());
    }
}

private void SavedToSelectedFolder(Sharing.ErrorCode errorCode)
{
    if (errorCode == Sharing.ErrorCode.NoError)
    {
        Debug.Log("Successfully saved the video to a folder");
    }
    else if (errorCode == Sharing.ErrorCode.Cancelled)
    {
        Debug.Log("User cancelled saving the video to a folder");
    }
    else
    {
        Debug.Log("Failed to save the video to a folder, error: " + errorCode.ToString());
    }
}

```

By default the game engine is not paused when the share sheet is opened, in case you want to pause it pass `true` as second parameter when calling `ShowShareSheet`. If you don't need the sharing functionality you don't need to import NextGenRecorder/Sharing folder. If you already imported it, you can safely remove the folder to decrease your binary size.

Recorder API

All classes are under `pmjo.NextGenRecorder` namespace so remember `using pmjo.NextGenRecorder;`

Properties

```

// Recording state
public static bool IsSupported
public static bool IsReadyForRecording
public static bool IsRecording
public static bool IsPaused

```

```

// Recording options
public static int TargetFrameRate
public static bool FrameSkipping
public static int MinDiskSpaceRequired
public static float VideoScale
public static bool RecordAudio
public static bool RealtimeRecording
public static ColorSpace ColorSpace
public static bool VerticalFlip

// Watermark
public static Texture WaterMark

```

Functions

```

// Preparing the recording session (optional)
public static void PrepareRecording()

// Controlling the recording
public static void StartRecording()
public static void PauseRecording()
public static void ResumeRecording()
public static void StopRecording()

// Marking highlights
public static bool SetHighlight(int priority, int preSeconds, int postSeconds)
public static bool SetHighlight(int priority, int preSeconds, int postSeconds, long highlightId)

// Session handling
public static long[] GetAllRecordingSessions()
public static long GetLastRecordingSession()
public static bool RemoveRecordingSession(long sessionId)

// Video and thumbnail export
public static void ExportRecordingSession(long sessionId)
public static void ExportRecordingSessionHighlights(long sessionId)
public static void ExportRecordingSessionHighlights(long sessionId, float targetLength)
public static void ExportRecordingSessionHighlights(long sessionId, long[] highlightIdentifiers)
public static void ExportRecordingSessionHighlights(long sessionId, long[] highlightIdentifiers, float targetLength)
public static void ExportRecordingSessionThumbnail(long sessionId, int maxSideWidth, ThumbnailType type)

```

Delegates and events

```

// Delegates
public delegate void RecordingStartedDelegate(long sessionId)
public delegate void RecordingPausedDelegate(long sessionId)
public delegate void RecordingResumedDelegate(long sessionId)
public delegate void RecordingStoppedDelegate(long sessionId)

public delegate void RecordingExportedDelegate(long sessionId, string path, ErrorCode errorCode)
public delegate void HighlightsExportedDelegate(long sessionId, string path, ErrorCode errorCode)
public delegate void ThumbnailExportedDelegate(long sessionId, string path, ErrorCode errorCode)

public delegate int CustomVideoBitrateDelegate(long sessionId, int width, int height, int frameRate)
public delegate int CustomAudioBitrateDelegate(long sessionId, int sampleRate, int channelCount)

// Events
public static event RecordingStartedDelegate RecordingStarted
public static event RecordingPausedDelegate RecordingPaused
public static event RecordingResumedDelegate RecordingResumed
public static event RecordingStoppedDelegate RecordingStopped

public static event RecordingExportedDelegate RecordingExported
public static event HighlightsExportedDelegate HighlightRecordingExported
public static event ThumbnailExportedDelegate ThumbnailExported

// Callbacks
public static CustomVideoBitrateDelegate CustomVideoBitrate
public static CustomAudioBitrateDelegate CustomAudioBitrate

```

Error codes

```

public enum ErrorCode
{
    NoError = 0,
    NothingToExport,
    InvalidSession,
    RecordingFailed
}

```

Thumbnail types

```

public enum ThumbnailType
{
    JPG = 0,

```

PNG

```
}
```

Sharing API

All classes are under `pmjo.NextGenRecorder.Sharing` namespace so remember `using`

```
pmjo.NextGenRecorder.Sharing;
```

Delegates and events

```
// Delegates
public delegate void ShareSheetClosedDelegate
public delegate void SavedToPhotosDelegate(ErrorCode errorCode)
public delegate void SavedToSelectedFolderDelegate(ErrorCode errorCode)

// Events
public static event ShareSheetClosedDelegate ShareSheetClosed
public static event SavedToPhotosDelegate SavedToPhotos
public static event SavedToSelectedFolderDelegate SavedToSelectedFolder
```

Functions

```
public static void ShowShareSheet(string path)
public static void ShowShareSheet(string path, bool pauseEngine)
public static void SaveToPhotos(string path)
public static void SaveToPhotos(string path, string albumName)
public static void SaveToSelectedFolder(string path)
public static void SaveToSelectedFolder(string path, string preferredName)
```

Error codes

```
public enum ErrorCode
{
    NoError = 0,
    UnknownError,
    NotSupported,
    FailedToCreateAlbum,
    FileNotFound,
    PermissionError,
    Cancelled
}
```