Unity PSG Player

This is a library for synthesis PSG (Programmable Sound Generator) sound sources such as retro game consoles on Unity. The performance data is MML (Music Macro Language) text, which describes musical notation in text, allowing easy creation of music data. The library is designed to produce expressions similar to those of NES sound generators (except DPCM).

- The sound generator can produce four types of square waves, a triangle wave, and two types of noise. The triangle wave is a 4-bit waveform.
- Performance expressions include sweep, LFO (vibrato), and volume envelope.

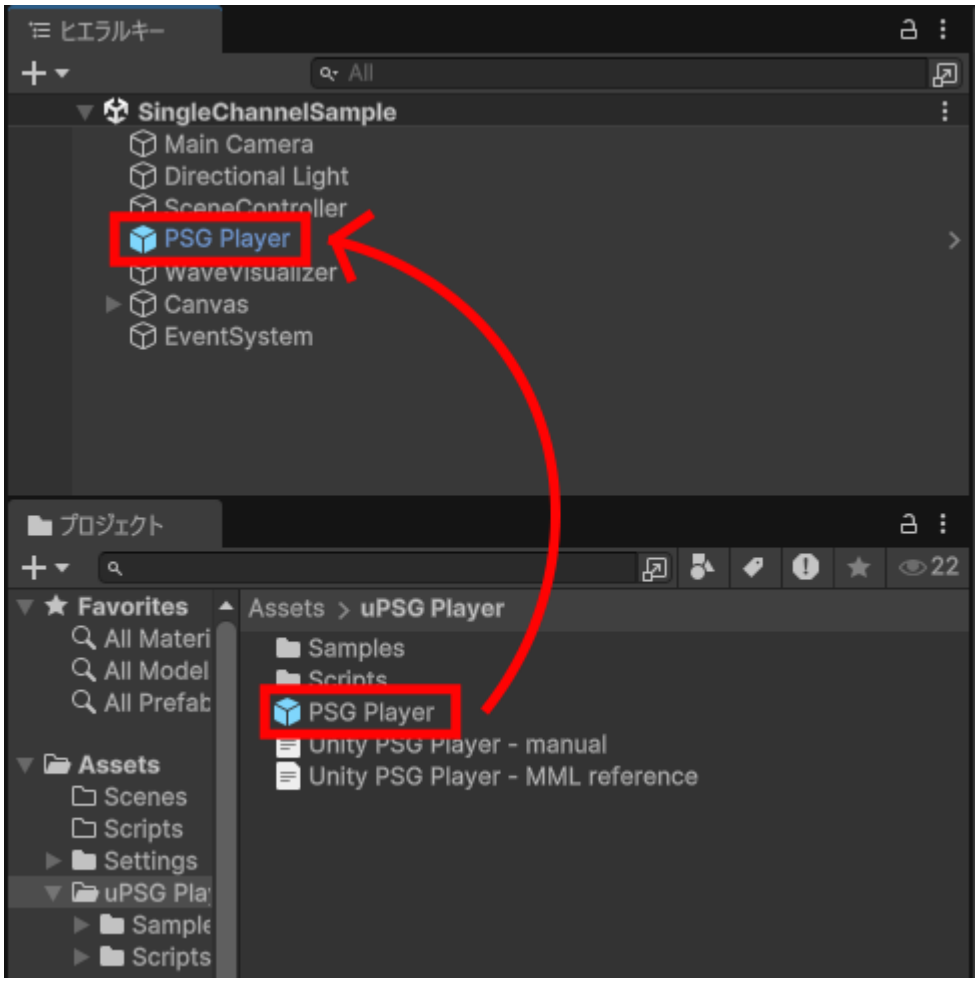
> I'm not aiming for a perfect recreation of the Famicom sound source. (It's too much trouble.)
> My goal is solely to produce sounds using only Unity, without preparing separate sound clips.

The PSG Player produces monophonic (single-note) sound. To play NES 4-note polyphonic sounds (excluding DPCM), you can achieve this by preparing multiple PSG Players and playing them simultaneously.

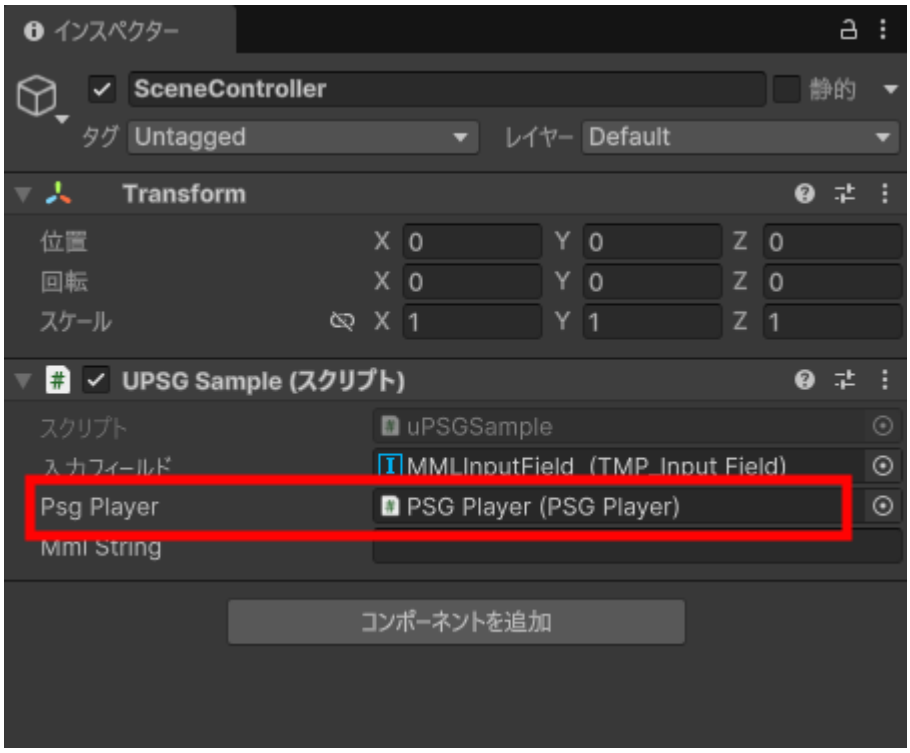
Quick Guide

Basic Usage

1. Place the PSG Player prefab in the Hierarchy.



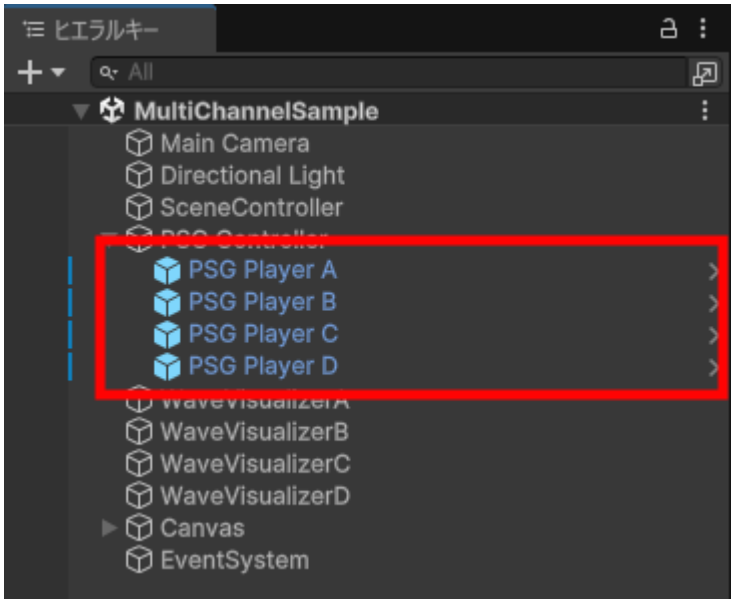
2. Prepare a PSGPlayer class variable in the script you are operating, and attach the PSG Player object you have placed.



3. The MML written in the mmlString variable of the PSGPlayer will be played by Play() function. For details on MML, refer to the MML Reference.

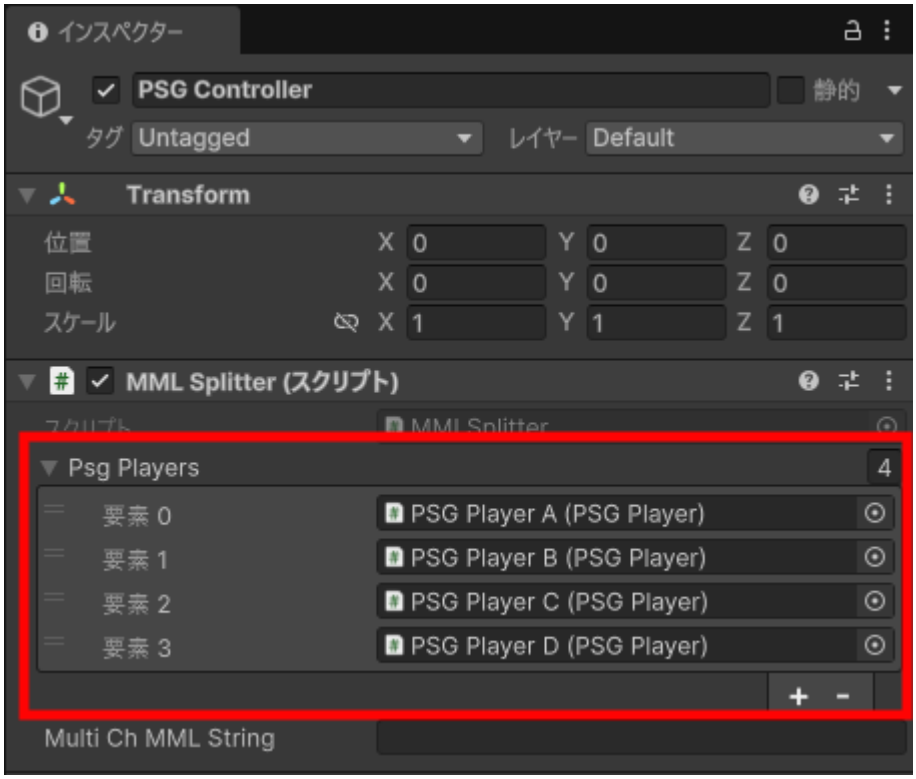
Multi-channel Usage

1. Place the PSG Player prefab according to the required number of channels.

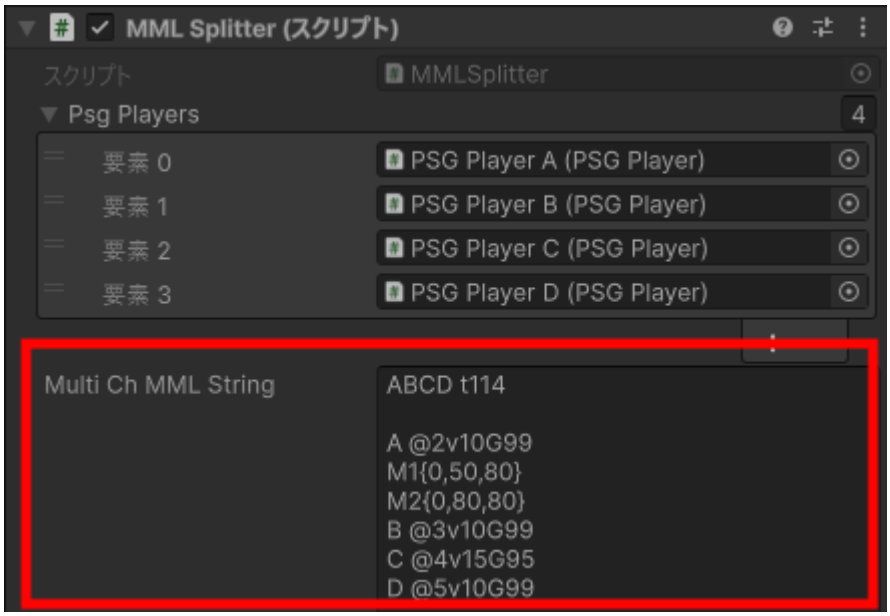


2. Attach the MMLSplitter script to the appropriate game object.

3. Assign the placed PSG Player to the psgPlayers of the MMLSplitter from the Inspector.



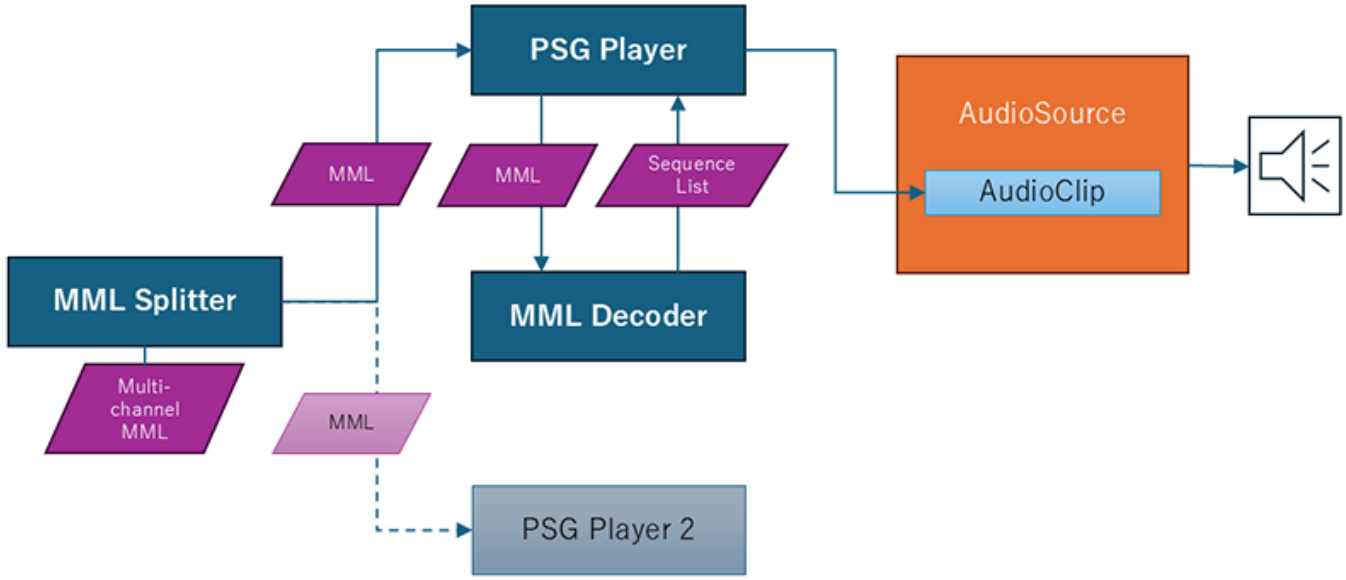
4. Place the MML into the multiChMMLString variable of MMLSplitter, distribute the MML to each channel using the SplitMML() function, and play it using the PlayAllChannels() function.



Composition

The Unity PSG Player consists of:

- The "**PSG Player**" that synthesizes sound according to sequence data
- The "**MML Decoder**" that converts MML to sequence data
- The "**MML Splitter**" that splits MML



MML Splitter is not required for single sound use.
An AudioSource component is also required to play the sound generated by PSG Player.

Summary

PSG Player synthesizes monophonic waveforms: square waves (pulse waves), triangle waves, and noise. The generated sound is attached as an AudioClip to an AudioSource resource and streamed. During streaming, the AudioClip sequentially requests the necessary buffer data, and PSG Player calculates the waveform data for each request.

PSG Player sequentially processes sequence data from a List array to calculate pitch and volume. Conversion from MML to sequence data is performed by the MML Decoder.

Additionally, use the MML Splitter to distribute multi-channel MML. It processes each line of the MML and determines the destination channel based on the characters at the beginning of the line. For details, refer to the MML Reference. Besides distributing MML, the MML Splitter can also centralize control over each PSG Player.

When playing sound effects using PSG, retro game consoles would interrupt the playback of one BGM channel to play sound effects in real time. However, the PSG Player buffers pre-loaded sounds, making it impossible to play sound effects with precise timing.
To play sound effects in PSG Player similarly to retro game consoles, prepare a separate PSG Player instance dedicated solely to sound effects and mute one of the BGM channels.

When using PSG Player in multi-channel mode, using the AudioMixer makes it easier to adjust volume levels and other settings.

About Sample rate

PSG Player allows you to set the sample rate for generated AudioClips. The sample rate is the number of samples per second. Since processing occurs for each sample, increasing the sample rate also increases CPU load.

Additionally, the sample rate affects the upper limit of the frequency of the generated sound.
The number of samples required per wavelength varies depending on the tone, as follows:

Tone	Samples
Square wave	2 samples
25%(75%) Pulse wave	4 samples
12.5% Pulse wave	8 samples
Triangle wave	32 samples

The frequency of sound that can be produced without issues is determined by dividing the sample rate by the number of samples mentioned above, ~~so the highest pitch achievable, especially for a triangle wave, will be lower~~.

`v0.9.2beta` Slightly modified the triangle wave generation logic, enabling it to produce high frequencies comparable to those of a square wave.

- This represents the upper limit for achieving the expected pitch; higher notes will cause the tone to deteriorate.
  At the default sampling rate, proper sound reproduction is only guaranteed up to approximately the seventh octave (o7).
- Due to unique noise processing, lower sample rates will cause volume to decrease at high frequencies.

Musical Expression

PSG Player allows you to use volume envelopes, sweeps, and LFOs for performance expression.
However, noise sounds and sounds generated by frequency specification will disable sweep and LFO effects that alter pitch.

About Rendering

`v0.9.6beta`

PSG Player has been developed assuming the use of the Stream from AudioClip.Create().
Stream playback distributes the load, so even long sequences like background music can be played with reasonably good responsiveness.
However, since buffering occurs when Create() is called, there was some concern about playing timing-sensitive sound effects.

Rendering outputs the entire sequence as waveform data for an AudioClip.
By preparing this waveform data in advance, you can achieve performance equivalent to playing a standard audio file.
However, rendering is computationally intensive, so converting long time performances may cause the program to freeze.

Using asynchronous rendering allows you to return to the main thread at regular intervals, thereby distributing the load.
Instead, the time required to complete rendering will increase.
We recommend using **streams**, **rendering**, and **asynchronous rendering** appropriately depending on the use case.

Note that during rendering, the sequence loop command (MML loop “L”) is disabled.
(There's this mysterious Unity behavior where loop points embedded in audio files get applied to AudioClips, but there's no way to add them when creating them manually...)

By the way, in WebGL, the Stream from AudioClip.Create() is unsupported (it plays but the buffer doesn't update), but rendered AudioClips can be played.

PSG Player script refference

List of Variables & Public Functions

Variables

---

mmlDecoder

```
[SerializeField] private MMLDecoder mmlDecoder;
```

Register the MML Decoder component that converts MML into sequence data.

---

audioSource

```
[SerializeField] private AudioSource audioSource;
```

Register the AudioSource that will receive the generated AudioClip.

---

sampleRate

```
public int sampleRate = 32000;
```

Sets the sample rate for the AudioClip.
The default is 32000Hz (32kHz).

---

audioClipSizeMilliSec

```
public int audioClipSizeMilliSec = 1000;
```

Sets the length of the AudioClip.
The default is 1000 milliseconds (1 second).

---

a4Freq

```
public float a4Freq = 440f;
```

Set the frequency of the A note in the fourth octave (o4a), which serves as the standard for the musical scale.
The default is 440Hz.
This variable can be changed using MML commands.

---

tickPerNote

```
public int tickPerNote = 960;
```

Set the resolution to one beat (quarter note).
Note duration is converted to the number of ticks based on this resolution, and the actual note length (in seconds) is calculated from the tempo and this resolution.

(Example: An eighth note corresponds to 480 ticks. At a tempo of 120, the note duration is
60[sec] / 120[notePerMin] * 480[tick] / 960[tickPerNote] = 0.25[sec]
which equals 0.25 seconds.)

---

programChange

```
public int programChange;
```

The number of the tone to be generated.

Number	Wave form

Number	Wave form
0	Pulse wave（12.5%）
1	Pulse wave（25%）
2	Square wave（50%）
3	Pulse wave（75%）
4	Triangle wave
5	Noise
6	Short-cycle noise

This variable can be changed using MML commands.

---

seqListIndex

```
[SerializeField] private int seqListIndex = 0;
```

Current position in sequence data processing.
Mainly displayed for debugging purposes.

---

seqList

```
[SerializeField] private List<SeqEvent> seqList = new();
```

A List array of sequence data.
Mainly displayed for debugging purposes.

---

mmlString

```
[Multiline] public string mmlString = "";
```

This is the MML string to be played.
Pass this variable to the MML Decoder to convert it into sequence data.

---

asyncRenderIsDone

```
public bool asyncRenderIsDone { get; private set; } = false;
```

```
```

v0.9.6beta

Asynchronous rendering completes and returns `True`.
Use after calling RenderSeqToClipDataAsync().

---

asyncRenderProgress

```
public float asyncRenderProgress { get; private set; } = 0f;
```

v0.9.6beta

The progress rate for asynchronous rendering increases from 0 to 1.
Use after calling RenderSeqToClipDataAsync().

---

renderedDatas

```
public float[] renderedDatas { get; private set; }
```

v0.9.6beta

This contains clip data generated by RenderSequenceToClipData() or RenderSeqToClipDataAsync().

Public Functions

Play()

```
public void Play();
```

- Parameter : None

Converts the MML string in the mmlString variable into sequence data and begins playback.

Play(string _mmlString)

```
public void Play(string _mmlString);
```

- Parameter : **_mmlString** MML string

Pass the parameter arguments to the mmlString variable, convert them to sequence data, and start playback.

DecodeMML()

```
public bool DecodeMML();
```

- Parameter : None
- Return value : `True` if decoding succeeds

Pass the MML string from the mmlString variable to the MML Decoder to convert it into sequence data.

PlayDecoded()

```
public void PlayDecoded();
```

- Parameter : None

Plays back decoded sequence data.
Since no conversion processing is performed, reduced CPU load is expected.

PlaySequence()

```
public void PlaySequence();
```

- Parameter : None

Plays back decoded sequence data.
Same as PlayDecoded().

Stop()

```
public void Stop();
```

- Parameter : None

Stop the currently playing audio.

IsPlaying()

```
public bool IsPlaying();
```

- Parameter : None
- Return value : `True` if playing

Returns the playback status of AudioSource.

Mute(bool isOn)

```
public void Mute(bool isOn);
```

- Parameter: **isOn** Set `True` to Mute on

When muted, the AudioSource is silenced and the generated sample is set to 0 (silence).
When unmuted with `False`, the AudioSource is immediately unmuted, but the sample remains silent until the next note event occurs.
However, if unmuted before the buffered sample plays out, the already generated sample will be played.

ExportSeqJson(bool _prettyPrint)

```
public string ExportSeqJson(bool _prettyPrint)
```

`v0.9.3beta`

- Parameter: **_prettyPrint** `True` enables line breaks and indentation (`False` by default)
- Return value: **JSON string**

The decoded MML sequence data is converted to JSON and output as a string.
If there is no sequence data (DecodeMML() or Play() has not been called), Null is returned.
The JSON content combines the tickPerNote value with the seqList.

DecodeAndExportSeqJson(bool _prettyPrint)

```
public string DecodeAndExportSeqJson(bool _prettyPrint)
```

`v0.9.3beta`

- Parameter: **_prettyPrint** `True` enables line breaks and indentation (`False` by default)
- Return value: **JSON string**

After decoding MML, serializes the sequence data into JSON and outputs it.

GetSeqJson()

```
public SeqJson GetSeqJson()
```

v0.9.3beta

- Parameter: None
- Return value: **SeqJson class object**

Outputs the decoded MML sequence data as a SeqJson class object.
This is the data before being converted to JSON by ExportSeqJson().
It is primarily used when exporting multi-channel JSON with the MML Splitter.

ImportSeqJson(string _jsonString)

```
public bool ImportSeqJson(string _jsonString)
```

v0.9.3beta

- Parameter: **_jsonString** JSON string
- Return value: `True` if import succeeds

Import JSON-formatted strings as sequence data.
At this time, the value of tickPerNote is also loaded.

SetSeqJson(SeqJson _seqJson)

```
public bool SetSeqJson(SeqJson _seqJson)
```

v0.9.3beta

- Parameter: **_jsonString** JSON string
- Return value: `True` if import succeeds

Directly reads the tickPerNote value and sequence data from a SeqJson class object.
Primarily used when importing multi-channel JSON with MML Splitter.

RenderSequenceTodClipData()

```
public float[] RenderSequenceTodClipData()
```

v0.9.4beta

- Parameter: None
- Return value: **Sample data consisting of a float array**

Generates the waveform data for the entire sequence.
Calculates each sample at the sample rate specified by sampleRate, then returns each sample as a float array.
The higher the sample rate and the longer the sequence, rendering will take longer time.
Additionally, the sequence loop command (MML Loop "L") will be disabled during rendering.

ExportRenderedAudioClip(bool isAsyncRendered)

```
public AudioClip ExportRenderedAudioClip(bool isAsyncRendered)
```

v0.9.4beta

- Parameter: None
- Return value: **Rendered AudioClip**

Generate waveform data for the entire sequence and export it as an AudioClip.
The sample rate of the AudioClip is set to the value specified by samplRate.
When prioritizing responsiveness for sound effects, you can use this function to prepare the AudioClip in advance.
Conversely, for long sequences such as background music, be aware to rendering will takes long time.

v0.9.6beta

- Parameter: **isAsyncRendered** Use renderedDatas when `True` (default `False`)
- Return value: **Rendered AudioClip**

Export the AudioClip using renderedDatas without performing rendering.

RenderSeqToClipDataAsync(int interruptSample)

```
public bool RenderSeqToClipDataAsync(int interruptSample)
```

v0.9.6beta

- Parameter: **interruptSample** Number of samples where processing is interrupted
- Return value: `True` when rendering starts successfully

Begin asynchronous rendering of the entire sequence to waveform data.
Monitor rendering within the main thread's Update() method or similar. The rendering progress increases from 0 to 1 using asyncRenderProgress.
When rendering completes, asyncRenderIsDone becomes `True`.
The generated clip data is stored in renderedDatas.
ExportRenderedAudioClip(true) converts renderedDatas into an AudioClip.
Note that the sequence loop command (MML Loop "L") is disabled during rendering.

CalcSeqSample()

```
public int CalcSeqSample()
```

`v0.9.6beta`

- Parameter: None
- Return value: Number of samples after rendering

Calculates the size of the clip data generated when rendering.

PlayRenderedClipData(bool isLoop)

```
public bool PlayRenderedClipData(bool isLoop)
```

`v0.9.6beta`

- Parameter: **isLoop** `True` for loop playback
- Return value: `True` if playback is successful

Use renderedDatas to generate an AudioClip and play it using the AudioSource specified for the PSG Player.

MML Splitter script refference

[MML Splitter] List of Variables & Public Functions

- Variables
  - psgPlayers
  - multiChMMLString
  - asyncMultiRenderIsDone
  - asyncMultiRenderProgress
- Public Functions
  - SplitMML()
  - SplitMML(string _multiChMMLString)
  - SetAllChannelsSampleRate(int _rate)

- SetAllChannelClipSize(int _msec)
- PlayAllChannels()
- PlayAllChannelsDecoded()
- PlayAllChannelsSequence()
- DecodeAllChannels()
- StopAllChannels()
- IsAnyChannelPlaying()
- MuteChannel(int channel, bool isMute)
- ExportMultiSeqJson(bool _prettyPrint)
- DecodeAndExportMultiSeqJson(bool _prettyPrint)
- ImportMultiSeqJson(string _jsonString)
- ExportMixedAudioClip(int _sampleRate, bool isAsyncRendered)
- RenderMultiSeqToClipDataAsync(int _sampleRate, int interruptSample)
- PlayAllChannelsRenderedClipData(bool isLoop)

[MML Splitter] Variables

---

psgPlayers

```
[SerializeField] private PSGPlayer[] psgPlayers;
```

Register a PSG Player component for each channel to send the MML in segments.

---

multiChMMLString

```
public string multiChMMLString;
```

Register the original MML string to be sent in segments.

---

asyncMultiRenderIsDone

```
public bool asyncMultiRenderIsDone { get; private set; } = false;
```

`v0.9.6beta`

Asynchronous rendering completes on all channels returns `True`.
Use after calling RenderMultiSeqToClipDataAsync().

---

asyncMultiRenderProgress

```
public float asyncMultiRenderProgress { get; private set; } = 0f;
```

v0.9.6beta

The progress rate for asynchronous rendering across all channels increases from 0 to 1.
Use after calling RenderMultiSeqToClipDataAsync().

[MML Splitter] Public Functions

SplitMML()

```
public void SplitMML();
```

- Parameter : None

Splits the MML string from multiChMMLString and sends it to the PSG Player registered in psgPlayers.
For channel assignment details, refer to "MML Reference".

SplitMML(string _multiChMMLString)

```
public void SplitMML(string _multiChMMLString);
```

- Parameter : **_multiChMMLString** Multi-channel MML string

Pass the parameter arguments to the multiChMMLString variable to send the MML to PSG Player in segments.

SetAllChannelsSampleRate(int _rate)

```
public void SetAllChannelsSampleRate(int _rate);
```

- Parameter : **_rate** Sample rate

Set the sample rate for all PSG players (in Hz).

SetAllChannelClipSize(int _msec)

```
public void SetAllChannelClipSize(int _msec);
```

- Parameter : **_msec** AudioClip length

Sets the AudioClip length for all PSG Players (in milliseconds).

PlayAllChannels()

```
public void PlayAllChannels();
```

- Parameter : None

All PSG Players decode MML and play simultaneously.

PlayAllChannelsDecoded()

```
public void PlayAllChannelsDecoded();
```

- Parameter : None

Play back the decoded sequence data simultaneously on all PSG Players.

PlayAllChannelsSequence()

```
public void PlayAllChannelsSequence()
```

- Parameter : None

Plays sequence data simultaneously on all PSG players.
Same as PlayAllChannelsDecoded().

DecodeAllChannels()

```
public void DecodeAllChannels()
```

- Parameter : None

All PSG Players decode MML into sequence data.
Since multi-channel MML is not transmitted in split segments, please execute SplitMML() beforehand.

StopAllChannels()

```
public void StopAllChannels();
```

- Parameter : None

Stop playing all PSG players.

IsAnyChannelPlaying()

```
public bool IsAnyChannelPlaying();
```

- Parameter : None
- Return value : `True` if any PSG Player is currently playing

Returns `True` if any of the AudioSources for each PSG Player is currently playing.

MuteChannel(int channel, bool isMute)

```
public void MuteChannel(int channel, bool isMute);
```

- Parameter: **channel** Target channel
  isMute `True` for mute on

Mutes the specified channel.

ExportMultiSeqJson(bool _prettyPrint)

```
public string ExportMultiSeqJson(bool _prettyPrint)
```

`v0.9.3beta`

- Parameter: **_prettyPrint** Enables line breaks and indentation when set to `True` (default `False`)
- Return value: **JSON string**

The decoded MML sequence data for each PSG Player is consolidated into JSON and output as a string.
The JSON contents consist of a list of SeqJson class objects output by each PSG Player.

DecodeAndExportMultiSeqJson(bool _prettyPrint)

```
public string DecodeAndExportMultiSeqJson(bool _prettyPrint)
```

v0.9.3beta

- Parameter: **_prettyPrint** Enables line breaks and indentation when set to `True` (default `False`)
- Return value: **JSON string**

Each PSG Player's MML is decoded, then the sequence data is converted to JSON and output.
Since multi-channel MML is not transmitted in split segments, please execute SplitMML() beforehand.

ImportMultiSeqJson(string _jsonString)

```
public void ImportMultiSeqJson(string _jsonString)
```

v0.9.3beta

- Parametr: **_jsonString** JSON string

Import JSON-formatted strings as multi-channel sequence data.

ExportMixedAudioClip(int _sampleRate, bool isAsyncRendered)

```
public AudioClip ExportMixedAudioClip(int _sampleRate, bool isAsyncRendered)
```

v0.9.4beta

- Parameter: **_sampleRate** AudioClip sample rate
- Return value: **AudioClip**

Mix the waveform data rendered by each PSG Player and export as an AudioClip.
The combined waveform data is summed by dividing by the number of channels to ensure it does not exceed the sample value range.
Therefore, the maximum volume of a single tone decreases depending on the number of channels.
Additionally, since the sample rate must be consistent across all channels, after using this function, the sampleRate of each PSG Player will be set to the value specified in the argument.
Please note that rendering all PSG players before mixing can take a significant amount of time, especially for longer sequences.

v0.9.6beta

- Parameter: **_sampleRate** AudioClip sample rate
- Parameter: **isAsyncRendered**　Use rendered waveform data (default `False`)
- Return value: **AudioClip**

Setting `isAsyncRendered` to `True` will mix the renderedDatas from each channel and export an AudioClip.
In this case, rendering is not performed, so after calling RenderMultiSeqToClipDataAsync(), use it only after asyncMultiRenderIsDone returns `True`.

RenderMultiSeqToClipDataAsync(int _sampleRate, int interruptSample)

```
public bool RenderMultiSeqToClipDataAsync(int _sampleRate, int interruptSample)
```

`v0.9.6beta`

- Parameter: **_sampleRate** AudioClip sample rate
- Parameter: **interruptSample** Number of samples where processing is interrupted
- Return value: Once rendering begins on any channel, return `True`

Begin asynchronous rendering on all channels.
Monitor rendering within the main thread's Update() method or similar.
The rendering progress increases from 0 to 1 using asyncMultiRenderProgress.
When rendering completes, asyncMultiRenderIsDone becomes `True`.
The generated clip data is stored in the renderedDatas of each PSG Player.
You can use ExportMixedAudioClip() to mix renderedDatas and convert them into an AudioClip.
Note that the sequence loop command (MML Loop “L”) is disabled during rendering.

PlayAllChannelsRenderedClipData(bool isLoop)

```
public void PlayAllChannelsRenderedClipData(bool isLoop)
```

`v0.9.6beta`

- Parameter: **isLoop** If `True`, loop playback
- Return value: None

Use renderedDatas to generate AudioClips on each channel, then play them using the AudioSource specified for each PSG Player. Since the audio is played unmixed, you can adjust the volume or mute each channel individually.
