

Rhythm Timeline

Last Updated: 18/4/2021

Version: 1.1



dypsl**oo**m

Table of Contents

- [Introduction](#)
- [Contents](#)
- [Getting started](#)
- [Rhythm Timeline Asset](#)
- [Notes and Note Definitions](#)
- [Rhythm Director](#)
- [Rhythm Processor](#)
- [Track Object](#)
- [Input](#)
- [Score Manager](#)
- [Save Manager](#)
- [Event Receivers](#)
- [Utility Scripts](#)

Introduction

Rhythm Timeline is a system built on top of the Unity Timeline and allows you to create rhythm tracks very easily in the Unity Editor. In addition all Unity Timeline features work too, so you may create cinematic views animated by the timeline while your rhythm tracks are playing. It will be synced automatically and everything can be previewed in the Editor, no need to enter play mode to see the result. This makes it completely different from other existing solutions.

The Notes can easily be edited without any code, both in game and in the editor. Those who wish to create custom notes with special functionality can easily do so with a bit of code as the Rhythm Timeline source code was refactored many times to be as simple as possible. All the source code is included!



Contents

Main Features

- Rhythm track Editor using Timeline
- Preview note positions from the Game View in Editor mode
- Drag & Drop Note Definitions on the Rhythm Tracks
- Bpm & Non-Bpm constrained Note duration and position supported
- Fully customizable Note Prefabs
- Customizable Editor Clips
- Event Senders and Receivers for each Input & Note state

- Easy to use Pool, Toolbox, and Scheduler utility scripts
- Accuracy & Score system

The asset contains the following:

- Custom Playable scripts for the Rhythm Track, Timeline Asset, Clip and more
- 11 timelines with 8 music of different genres.
- 4 Note scripts, 7 Note prefabs: Tap, Hold, Counter, Swipe (Left, Right, Up, Down)
- Utility scripts: Scheduler, Object Pool, Toolbox
- Accuracy and scoring system with saving to disk
- Demo scene with a song chooser, works both on PC and Mobile devices

The asset requires:

- Unity 2019.4 or higher
- TextMeshPro v2 or higher
- Timeline v1.2 or higher

Getting started

Before importing the asset in your project make sure to download the required packages from the package manager. Go to Window -> Package Manager.

Import the following packages:

- Timeline v1.2 or higher

If you do not import those packages prior this asset errors will pop up in the console.

The best way to get started is by trying out the demo scene. By checking the game objects in the scene and the components attached to them you'll get an idea of how components interact with each other.

The main game objects to look at are:

- Game Managers
 - Pool Manager : Automatically pools objects to improve performance
 - Toolbox : Register and get any object from anywhere using the Toolbox
 - Dsp Time : Get the Dsp Time or the adaptive Dsp Time from anywhere
 - Rhythm Game Manager : The manager used in the demo to control the UI and the gameplay loop
 - Score Manager : The manager that listens to the notes being triggered through events and manages the score

- Rhythm Director
 - Rhythm Director : This takes care of setting all the parameters correctly when a timeline asset start playing, it also maps Track Objects to Timeline Rhythm Tracks.
 - Playable Director : The default Timeline component used by the Rhythm Director (Make sure to set it as **Update Method DPS Clock**)
 - Rhythm Processor : Takes care of managing Notes and receiving input event
 - Rhythm Standard Input : Get input the standard way and tells the processor what was input was executed
- Game World -> Default -> TrackObjects
 - Track Object : Define the start and end points for that track

When starting fresh it is recommended to duplicate the demo scene. and build on top.

Creating a new Scene

If you wish to create a scene follow these steps:

1. Create a new empty scene.
2. Drag and drop the "Rhythm Director" prefab and "Managers" prefab in the new Scene.
3. Drag and drop as many "Track Object" prefabs as you want (The demo songs use 4). Adjust their positions such that the target end position can be seen in the game view.
4. Make sure to reference those track objects in the director, under the Track Objects field.
5. (Optionally) Make sure to toggle the Play On Start option on the Rhythm Director. This will allow you to start playing your songs without having to create a game manager for selecting songs. The song timeline can be set in Playable field the Playable Director component next to the Rhythm Director
6. (Optionally) Add UI to display the score by assigning Text Mesh Pro components in the Score Manager (in Managers).
7. You may create a Rhythm Timeline Asset (learn how below) and set it in the Playable Director. Make sure the amount of tracks in the timeline matches the amount of track objects in the scene.

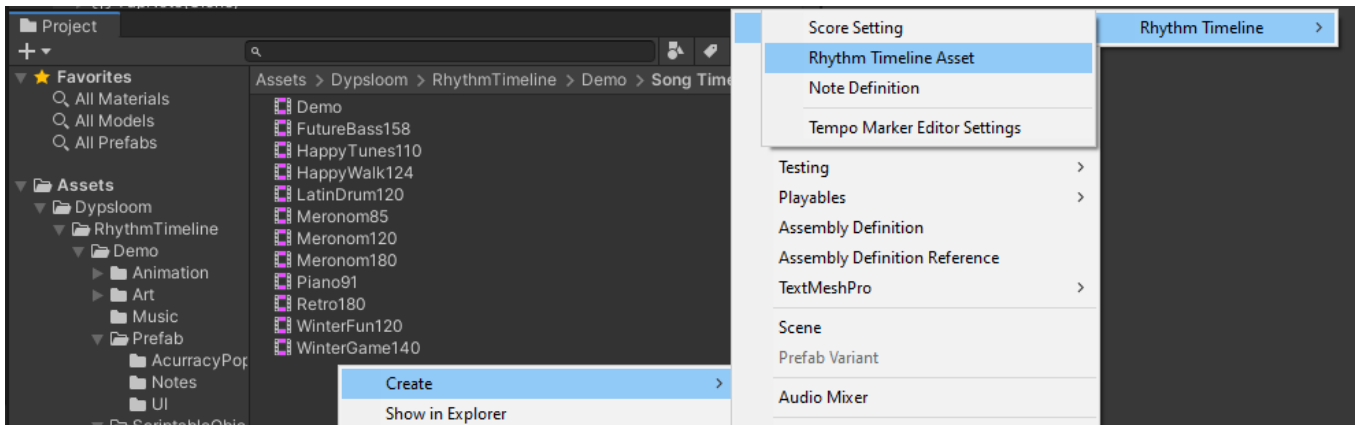
After following those steps you should be able to play your songs in the new scene. It is recommended you start creating your own custom Rhythm Game Manager from scratch to create your own gameplay loop. You are free to use the song chooser provided in the demo scene, but you may wish to create your own.

The rest of this documentation will teach you how to take advantage of each existing component and even extend them to fit your exact needs.

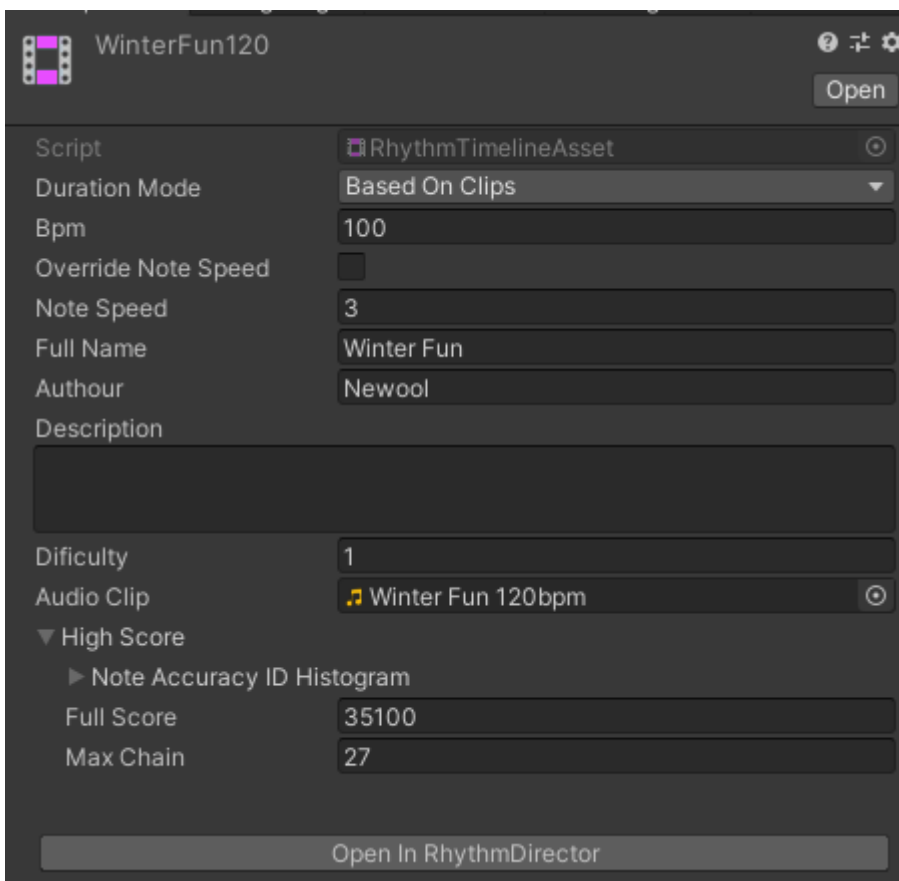
Rhythm Timeline Asset

The Rhythm Timeline asset is a custom Timeline Asset which has information about your song, such as the BPM, the Author, a description, etc...

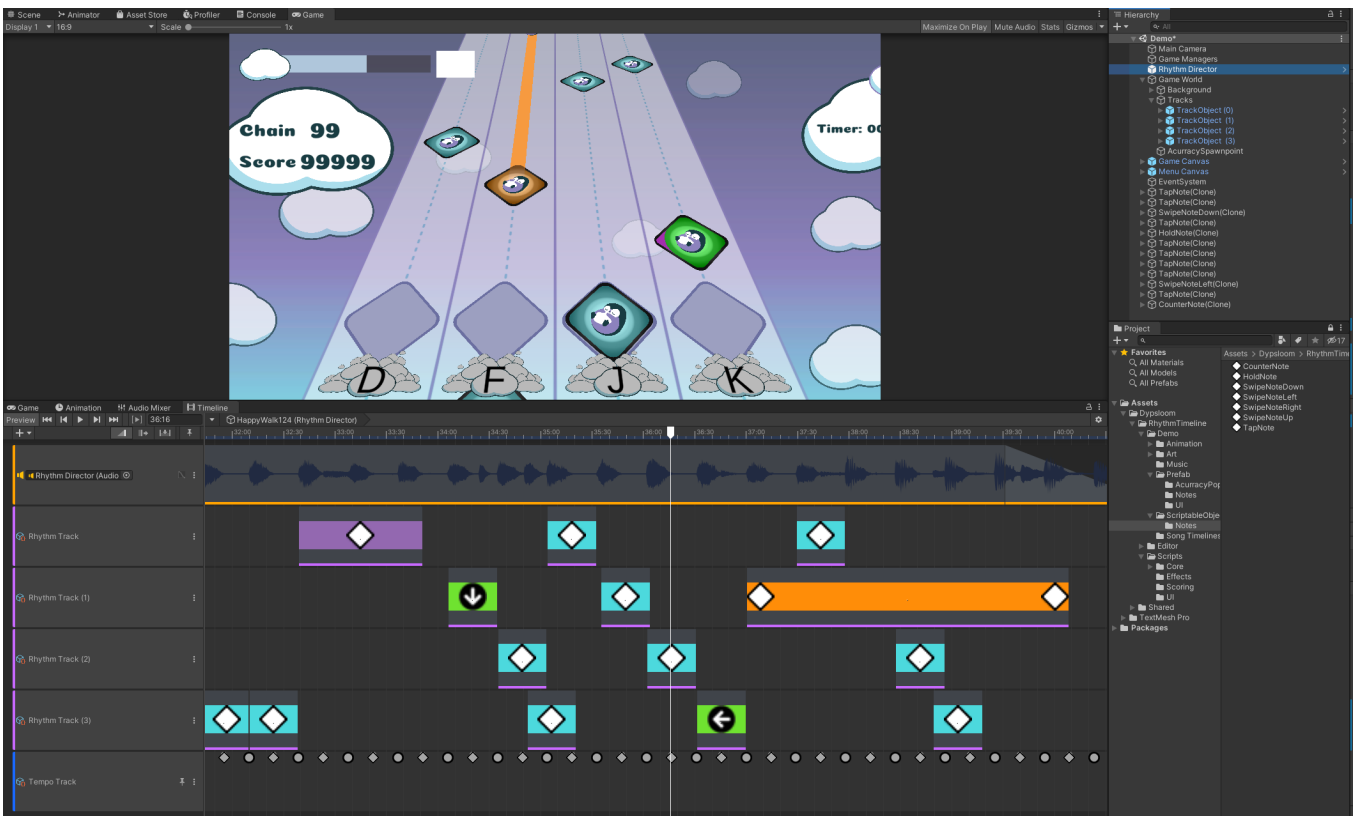
You can create a new Rhythm Timeline Asset using right-click in the project view and pressing Create -> Dypsloom -> RhythmTimeline -> Rhythm Timeline Asset.



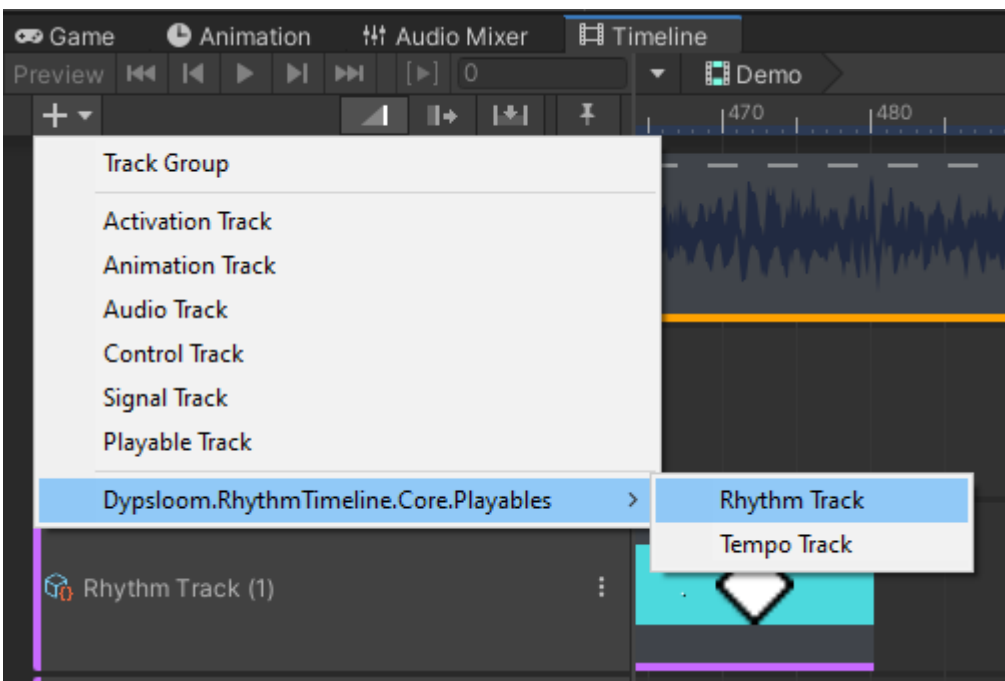
Once created and selected in the inspector you will see an "Open in Rhythm Director" button. Make sure that a Rhythm Director is setup correctly in the scene. Pressing the button will setup the RhythmTimeline as the current Playable such that it can be edited while viewing the preview.



It is recommended to open a Timeline window and the Game window adjacent to one another when editing the tracks.



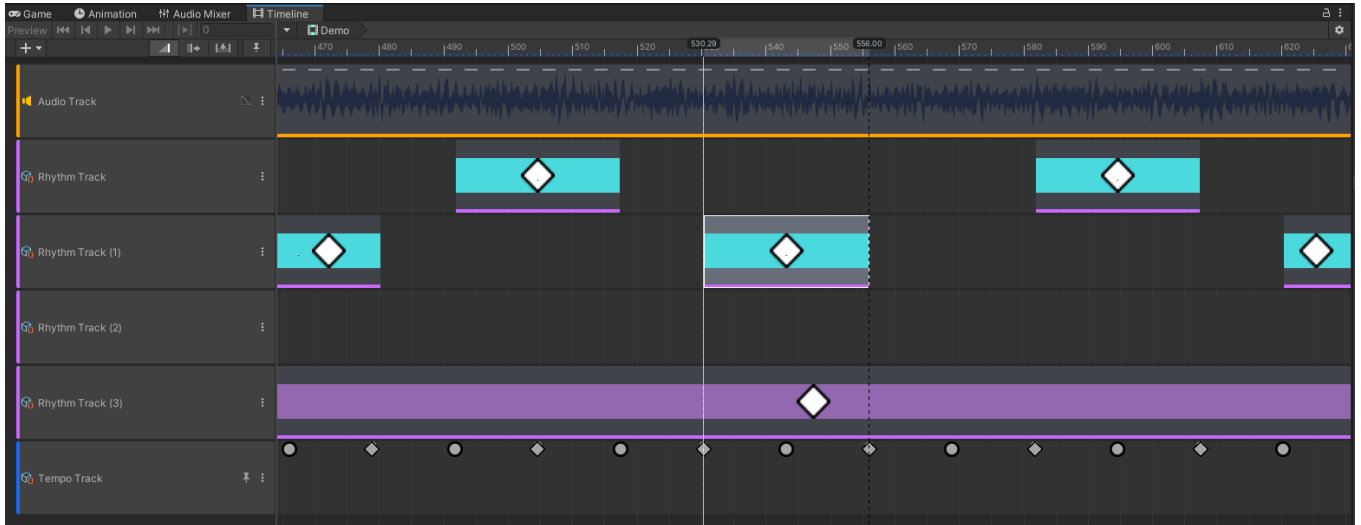
It is recommended to add at least an AudioTrack, as many RhythmTracks as wanted (it must match the amount of Track Objects referenced by the Rhythm Director in the scene) and a Tempo Track.



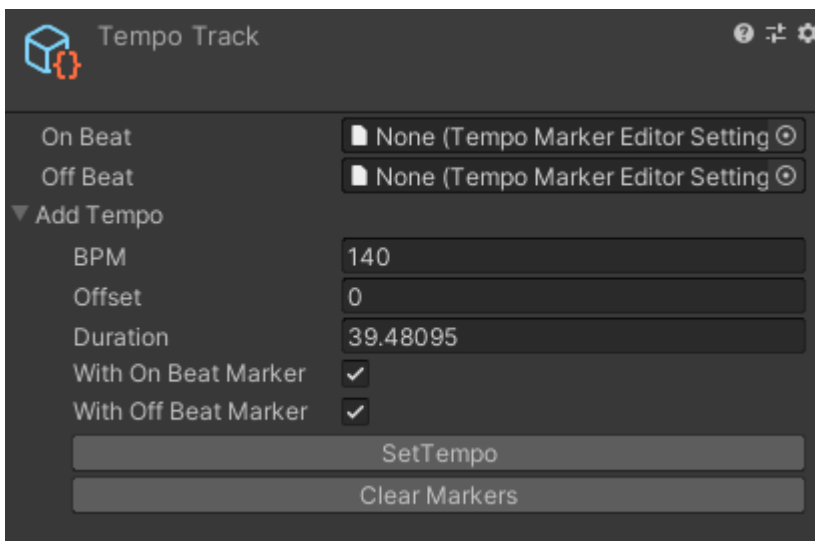
Rhythm Tracks must have IDs assigned to them

The Audio Track will play the music, you may even combine multiple songs together, ease in and out, etc... The editor even shows the Audio waveform, which helps setup the notes in the correct place. Note that the sound happen a little bit after it is shown in the waveform due to the timeline syncing with the Digital Signal Processing (DSP) clock.

The Tempo Track is highly recommend but not required. It helps the Rhythm Clips to snap in the correct place if you wish to have a somewhat consistent BPM.

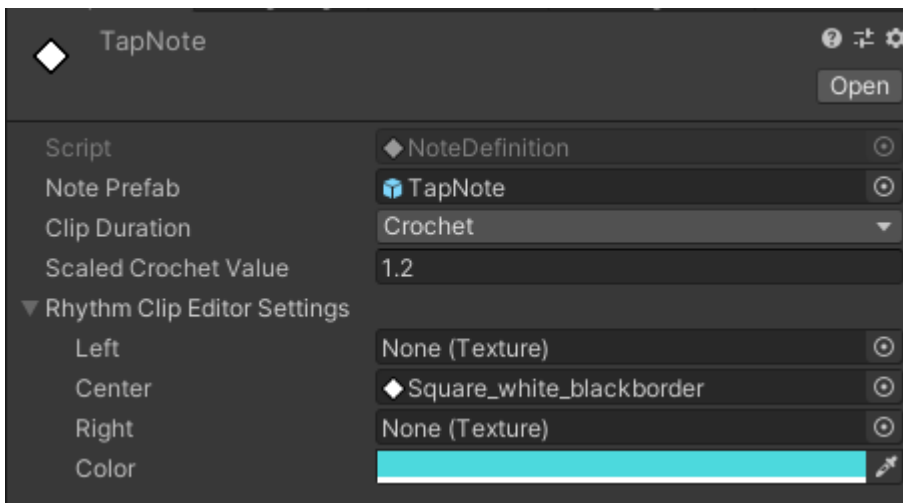


The Tempo can be set in the Track inspector. By default it is automatically set to draw the tempo using the BPM of the Rhythm Timeline Asset.



Finally you may start adding Rhythm Clips to the Rhythm Tracks. This can be done either by right-click on the Track and pressing "Add From Note Definition" or simply by drag & dropping the Note Definition scriptable objects on Track. The Note Definitions from the Demo can be found in Assets/Dypsloom/RhythmTimeline/Demo/ScriptableObject/Notes

Once Rhythm Clips exist on any track you may copy paste them and move them around to your pleasing. Clips can be easily customized visually using the Note Definition inspector.



If everything is setup correctly you will see the notes previewed in the Game view when using the timeline play button or while moving around the Timeline playhead.

To learn more about Timeline check the Unity documentation:

<https://docs.unity3d.com/Packages/com.unity.timeline@1.3/manual/index.html>

Especially read the playback controls which have some great tips to work on your timelines:

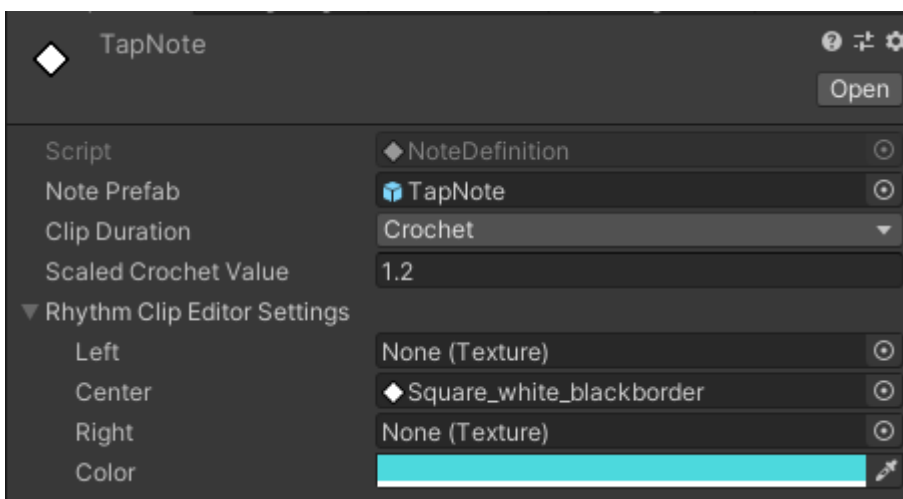
https://docs.unity3d.com/Packages/com.unity.timeline@1.3/manual/tl_play_cntrl.html

Notes and Note Definitions

The Note Definition is a scriptable object which references a Note prefab. It is used on Rhythm Clips to know what the clip should look like in the Editor and what note should be spawned.

The Note Definition also has a way of limiting the clip duration to match the bpm or a fraction of it.

The Note Definition can be dragged and dropped in the timeline editor to place notes. It also has fields to customize the look of the clip within the Editor.



Notes are components which update with the timeline and sync to the DSP. By default our system comes with 4 different Note types:

- Tap Note
- Hold Note
- Swipe Note
- Counter Notes

All notes inherit from the same base class Note. The Note class deals with events that initialize it and updates over time, whether it is from the timeline or Monobehavior Update event.

Notes are Spawned by the Rhythm Processor and they are pooled. Pooling object means they are reused once deactivated, they are never destroyed completely, just hidden until they are needed again.

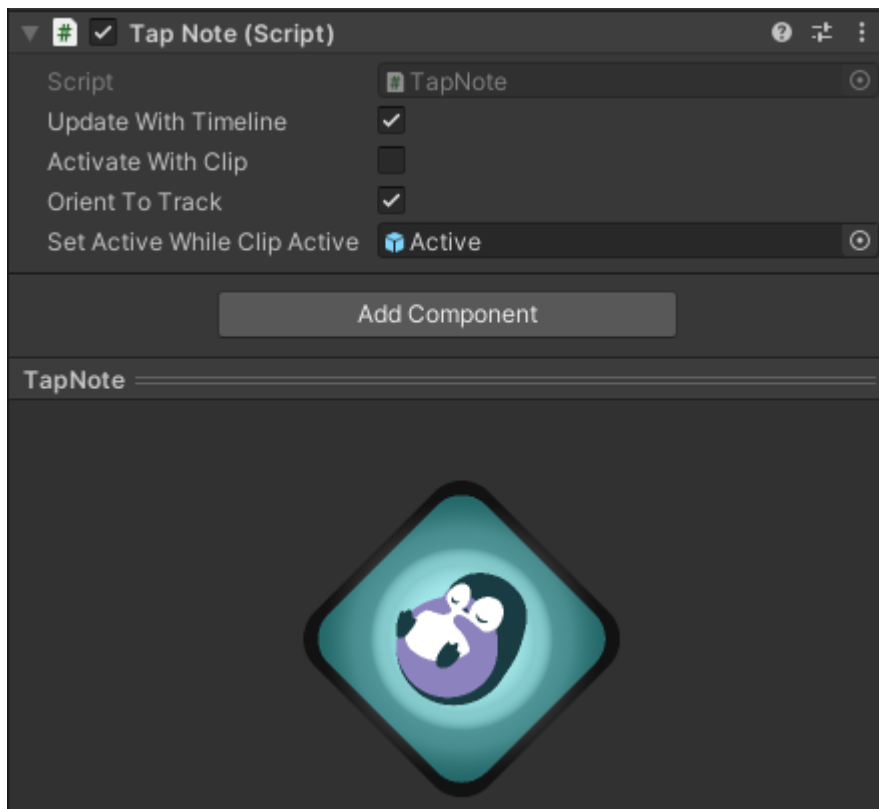
Therefore Notes have multiple states:

- Disabled : They are not used
- Pre-Active : The note is moving but is not yet interactable
- Active : The note can be interacted with
- Post-Active : The note used to be interactable but no longer is.

The movement of each Note is usually defined to start at the Track Object start point and be on the perfect timing at the end point. This is easily customizable by creating a custom note.

Tap Note

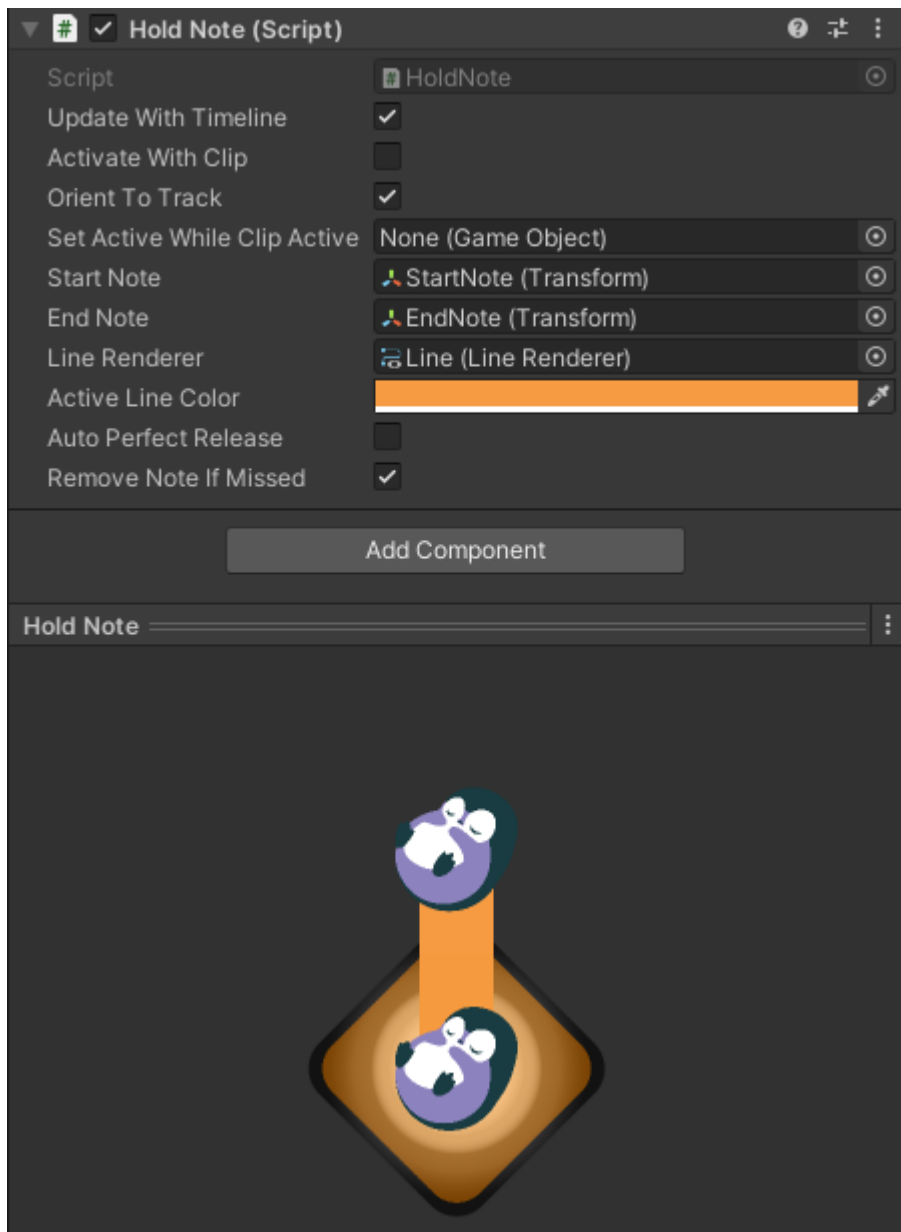
The tap note detects a single press input. The perfect timing is to press in center of the clip. Therefore it is recommended to combine it with a Note Definition that limits the clip size to a Crochet.



Hold Note

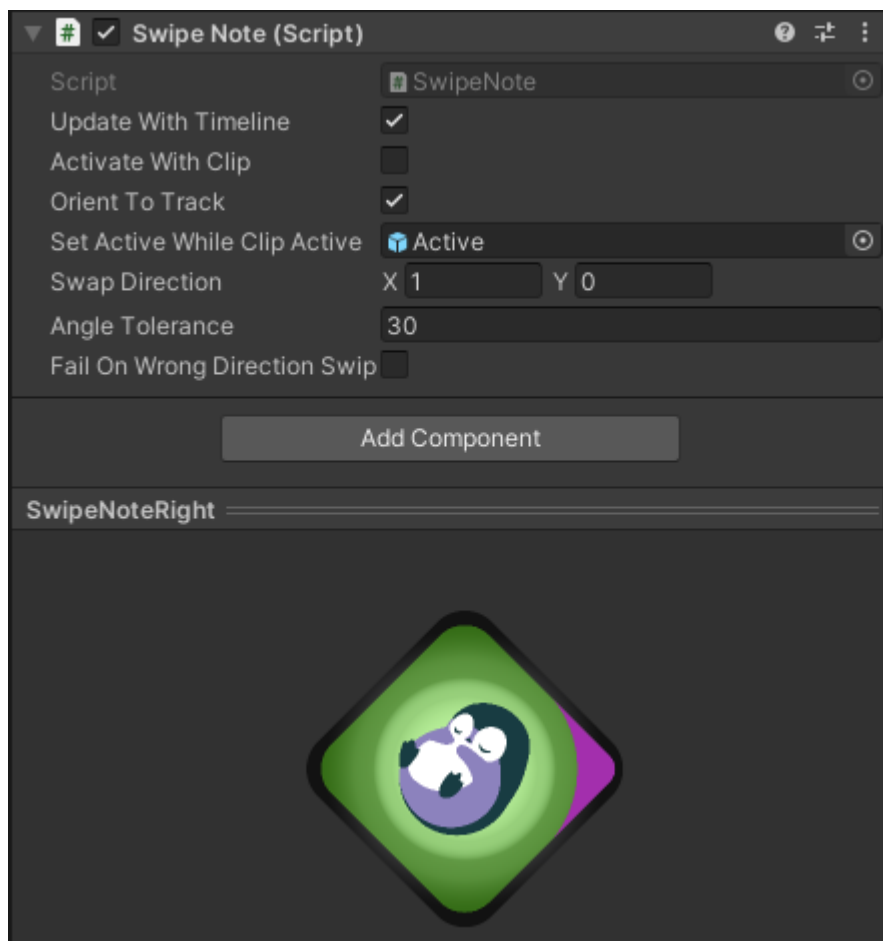
The hold note detects two input, press and release. Both need to match exactly the right timings. The perfect timing is to press in the clip half a crochet after the clip start and half a crochet before the clip end. Therefore it is recommended to combine it with a Note Definition that does not limit the clip size.

There are options to allow the hold note to be triggered automatically at the end without needing to release it manually at the perfect time.



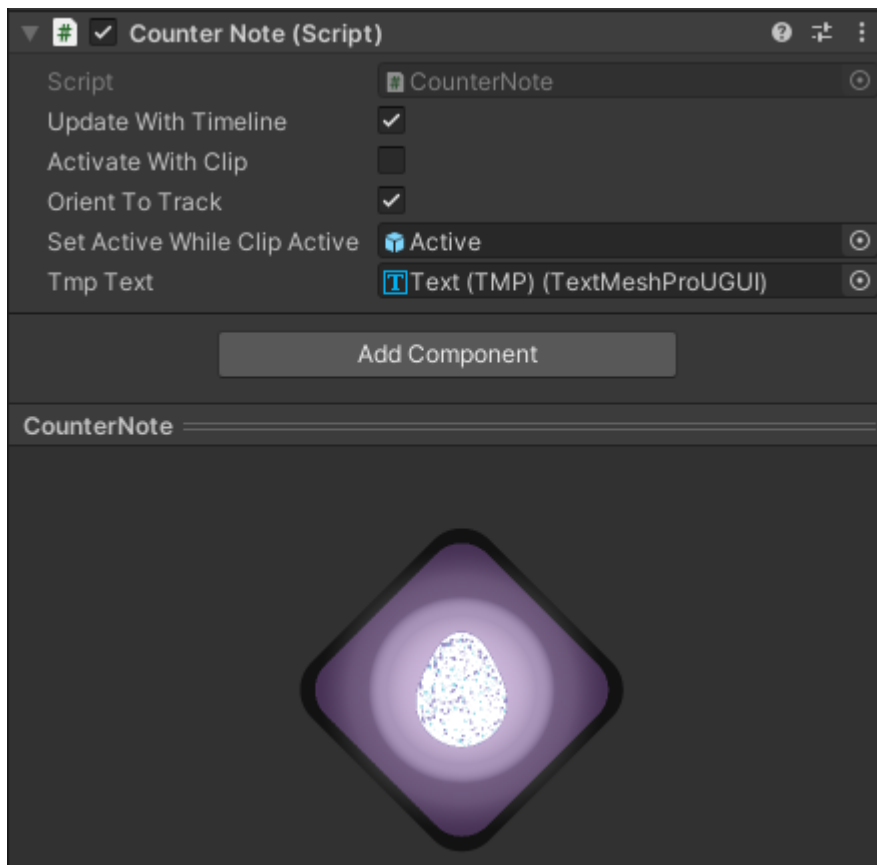
Swipe Note

The Swipe Note detects a swipe input with a certain direction. The perfect timing is to press in center of the clip. Therefore it is recommended to combine it with a Note Definition that limits the clip size to a Crochet.

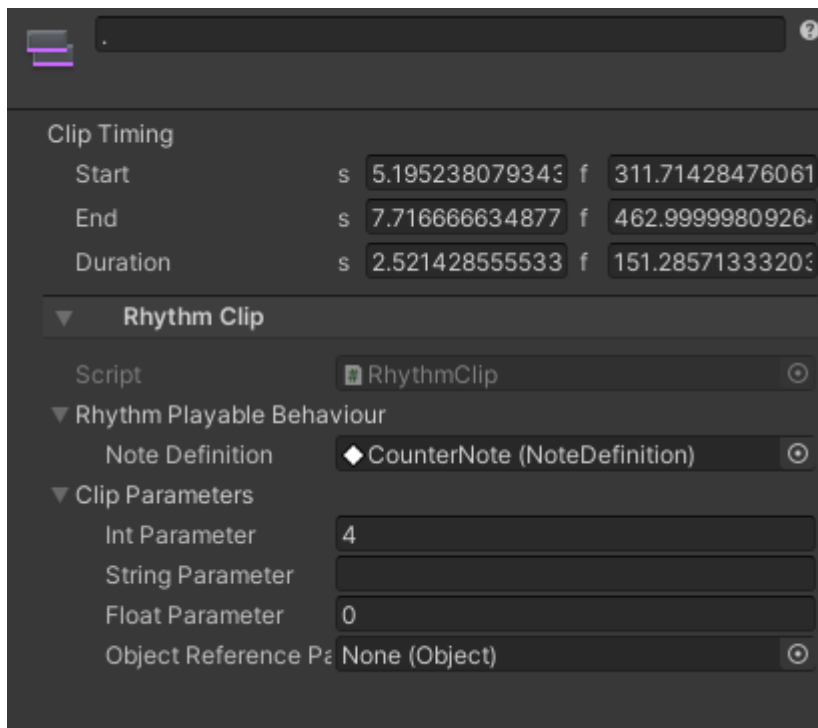


Counter Note

The counter Note needs to be The perfect timing is to press in the clip half a crochet after the clip start and half a crochet before the clip end. Therefore it is recommended to combine it with a Note Definition that does not limit the clip size.



The counter number can be customized per clip instead of per prefab. This allows for much easier iteration. Simply edit the Integer parameter of the Rhythm Clip



These values can be retrieved easily within the custom Note scripts via the Rhythm Clip Data property.

Custom Notes

Creating custom notes is very easy. Simply inherit from the Note class and override the functions you are interested in.

The code below shows an example of a Tap Note. The comments on the functions explain in detail when they are called and why.

```
/// <summary>
/// The Tap Note detects a single press input.
/// </summary>
public class TapNote : Note
{
    /// <summary>
    /// The note is initialized when it is added to the top of a track.
    /// </summary>
    /// <param name="rhythmClipData">The rhythm clip data.</param>
    public override void Initialize(RhythmClipData rhythmClipData)
    {
        base.Initialize(rhythmClipData);
    }

    /// <summary>
    /// Reset when the note is returned to the pool.
    /// </summary>
    public override void Reset()
    {
        base.Reset();
    }

    /// <summary>
    /// The note needs to be activated as it is within range of being triggered.
    /// This usually happens when the clip starts.
    /// </summary>
    protected override void ActivateNote()
    {
        base.ActivateNote();
    }

    /// <summary>
    /// The note needs to be deactivated when it is out of range from being triggered.
    /// This usually happens when the clip ends.
    /// </summary>
    protected override void DeactivateNote()
    {
        base.DeactivateNote();

        //Only send the trigger miss event during play mode.
        if(Application.isPlaying == false){return;}

        if (m_IsTriggered == false) {
            InvokeNoteTriggerEventMiss();
        }
    }
}
```

```

/// <summary>
/// An input was triggered on this note.
/// The input event data has the information about what type of input was triggered.
/// </summary>
/// <param name="inputEventData">The input event data.</param>
public override void OnTriggerInput(InputEventData inputEventData)
{
    //Since this is a tap note, only deal with tap inputs.
    if (!inputEventData.Tap) { return; }

    //The game object can be set to active false. It is returned to the pool automat
    gameObject.SetActive(false);
    m_IsTriggered = true;

    //You may compute the perfect time anyway you want.
    //In this case the perfect time is half of the clip.
    var perfectTime = m_RhythmClipData.RealDuration / 2f;
    var timeDifference = TimeFromActivate - perfectTime;
    var timeDifferencePercentage = Mathf.Abs((float)(100f*timeDifference)) / perfec

    //Send a trigger event such that the score system can listen to it.
    InvokeNoteTriggerEvent(inputEventData, timeDifference, (float) timeDifferencePer
    RhythmClipData.TrackObject.RemoveActiveNote(this);
}
/// <summary>
/// Hybrid Update is updated both in play mode, by update or timeline, and edit mode
/// </summary>
/// <param name="timeFromStart">The time from reaching the start of the clip.</param>
/// <param name="timeFromEnd">The time from reaching the end of the clip.</param>
protected override void HybridUpdate(double timeFromStart, double timeFromEnd)
{
    //Compute the perfect timing.
    var perfectTime = m_RhythmClipData.RealDuration / 2f;
    var deltaT = (float)(timeFromStart - perfectTime);

    //Compute the position of the note using the delta T from the perfect timing.
    //Here we use the direction of the track given at delta T.
    //You can easily curve all your notes to any trajectory, not just straight lines
    //Here the target position is found using the track object end position.
    var direction = RhythmClipData.TrackObject.GetNoteDirection(deltaT);
    var distance = deltaT * m_RhythmClipData.RhythmDirector.NoteSpeed;
    var targetPosition = m_RhythmClipData.TrackObject.EndPoint.position;

    //Using those parameters we can easily compute the new position of the note at a
    var newPosition = targetPosition + (direction * distance);
    transform.position = newPosition;
}
}

```

The "RhythmClipData" property has a lot of very useful information about the clip that is bound to the note. It is extremely useful if you plan to create your own notes.

You may easily customize Note to function exactly as you wish by overriding those functions.

Here are some ideas of custom notes which could be created by you

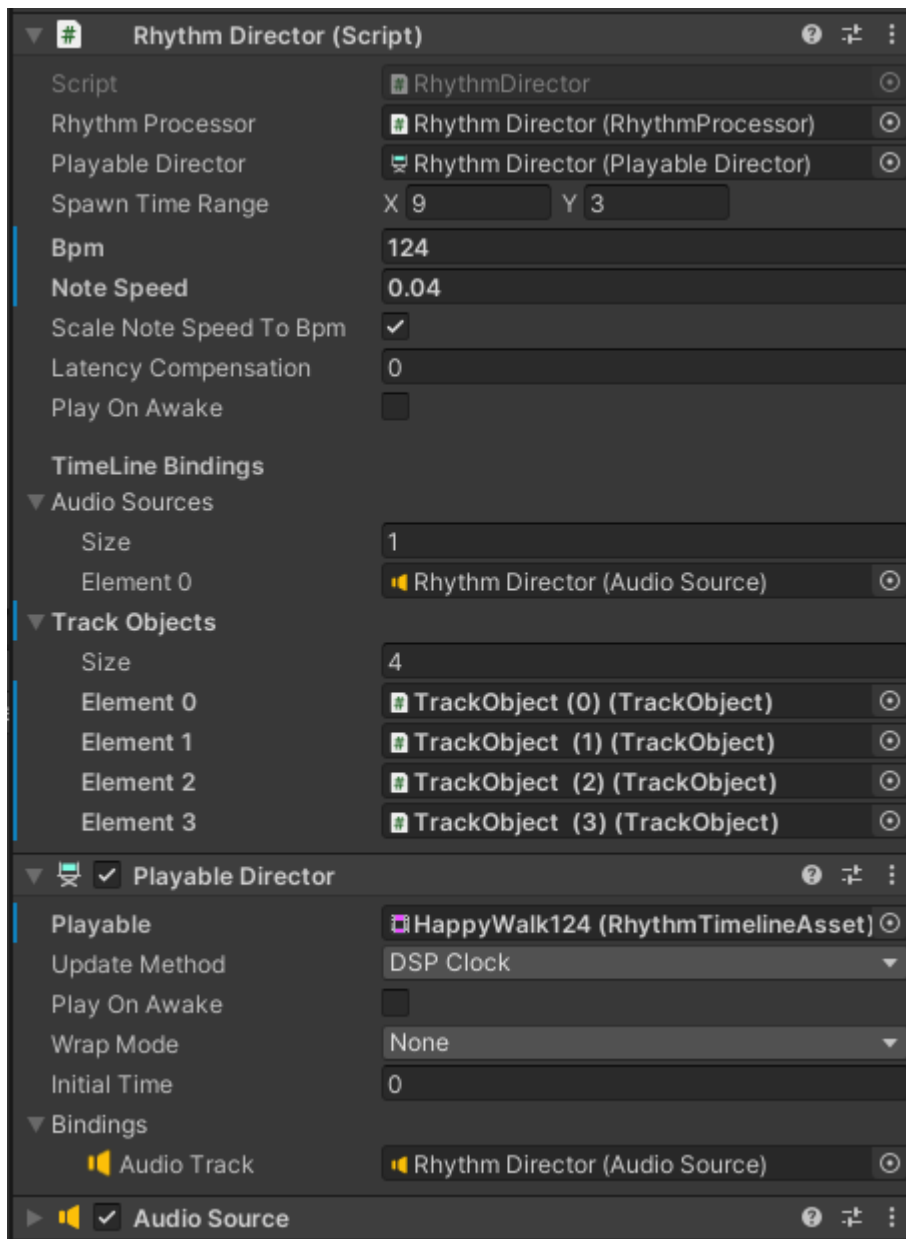
- Notes which rotate while going down the track
- Notes which gives you extra points when pressed
- Note that combines hold with swipe

These are just examples, with a bit of creativity any type of notes are possible.

Apart from customizing notes directly you may also customize the Track Object code to tell the notes trajectory at any given time.

Rhythm Director

The Rhythm Director component is the most important component in the system. It controls the Playable Director to binding the Rhythm Tracks to Track Objects and makes sure the Rhythm Timeline Asset can be previewed in Edit mode. Rhythm Timeline Assets must be opened in the Rhythm Director Playable Director to be previewed in edit more.



The Track Objects must be referenced in the Rhythm Director. The number must match the number of Rhythm Tracks within the Rhythm Timeline Asset.

The Spawn Time Range field determines how many seconds the notes must be spawned in the scene before the clips starts and how many seconds it should wait after the end of clip to remove the note.

The Note speed may be defined on the Rhythm Director or directly on the Rhythm Timeline Asset.

The Latency compensation delays the audio compared to the notes. This is useful in case the different devices used have a slight difference of latency between the image and the sound.

API

```
//Get the Rhythm Director from anywhere using the Toolbox.
m_RhythmDirector = Toolbox.Get<RhythmDirector>();

//Play a song from a Rhythm Timeline Asset
m_RhythmDirector.PlaySong(rhythmTimelineAsset);

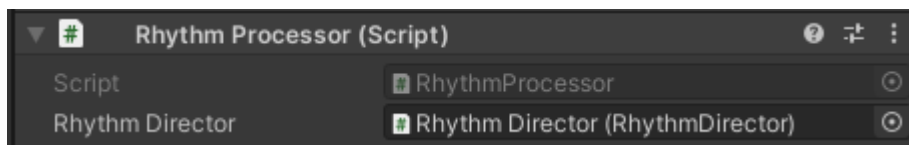
//Pause the song
m_RhythmDirector.Pause()

//Unpause the song
m_RhythmDirector.UnPause()

//End the song.
m_RhythmDirector.EndSong()
```

Rhythm Processor

The Rhythm Processor is used to create the Notes and return them to the pool when done. It also processes all the Note and Input events and broadcast them everywhere else.



API

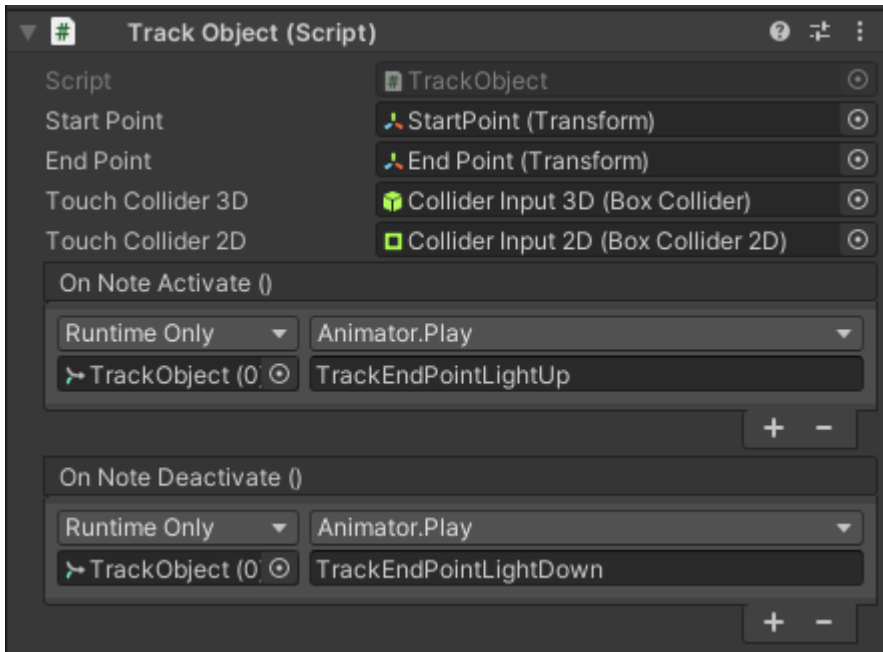
```
//Get the Rhythm Director from anywhere using the Toolbox.
m_RhythmDirector = Toolbox.Get<RhythmDirector>();

//Get the Rhythm Processor from the Rhythm Director.
m_RhythmProcessor = m_RhythmDirector.RhythmProcessor;

//Trigger an input
m_RhythmProcessor.TriggerInput(inputEvent);
```

Track Object

The Track Object is used to define the start and end point where the notes should go through. It may also define the path it takes from one point to the other.



The touch collider are defined on the Track Object. You may use both 2D and 3D colliders.

It is very useful to add Event Receiver components on the track to detect input and note events.

API

```
//Get the Track Objects from the RhythmDirector.
var trackObject = m_RhythmDirector.TrackObjects[0];

//Set the active note, this adds the note to be detected by input in the Rhythm Processo
endPoint.SetActiveNote(note);

//Remove the note from being active.
endPoint.RemoveActiveNote(note);

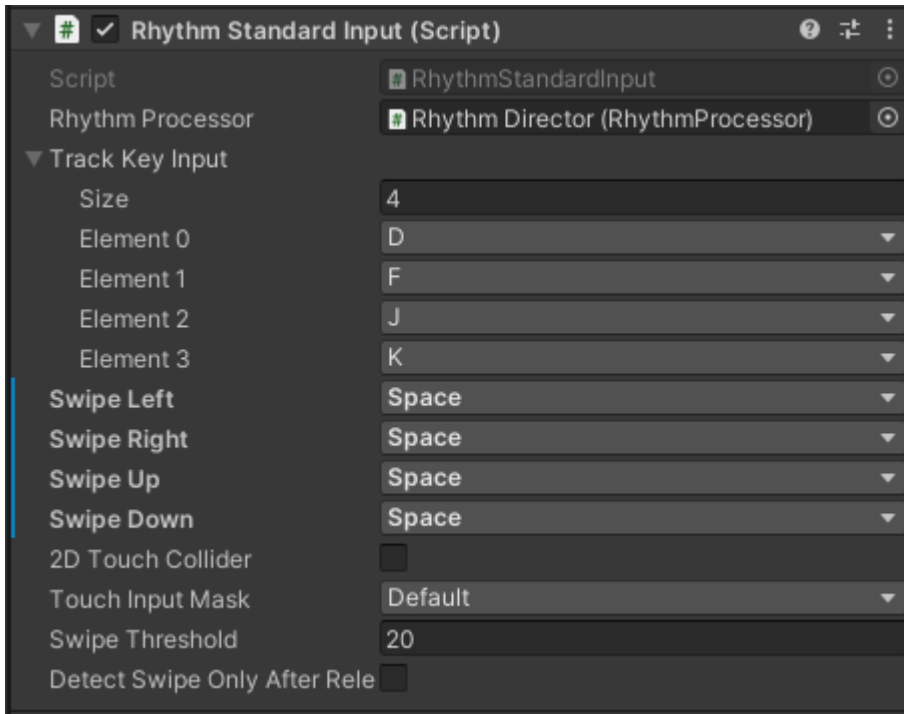
//Get the start point.
var start = trackObject.StartPoint;

//Get the end point.
var endPoint = trackObject.EndPoint;

//Get the not direction a delta T
var noteDirection = endPoint.GetNoteDirection(deltaT);
```

Input

The system comes with a simple Rhythm Standard Input component. This uses the default Unity input system.



You may define keyboard and mouse inputs.

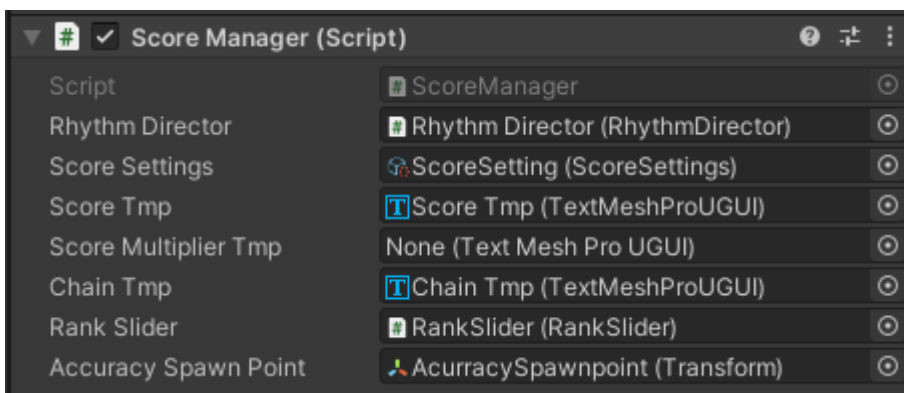
It is recommended to use 3D Touch Collider as it works both in perspective and orthogonal camera modes.

You may replace this component for another one that uses the Input system you like. Simply call the Trigger Input function on the Rhythm Processor to process your input.

Score Manager


The Score Manager listens to events from the Rhythm Processor to know when notes have been triggered. It uses the Score Setting scriptable object to find how accurate the press was and attribute the correct score as well as spawn a pop up.

Ranks (S, A, B, etc...) are also defined on the Score Setting.



The Score Manager component has many fields to set Text fields and sliders to display the score while playing a song.

To create a new Score Setting right-click in the project view and press Dypsloom -> RhythmTimeline -> Score Setting.

 ScoreSetting

Open

Script

ScoreSettings

▼ Accuracy Table

Size

5

▼ Perfect

Name

Perfect

Miss

☐

Break Chain

☐

Percentage Theshold

20

Score

800

Icon

Perfect

Pop Prefab

PerfectAccuracyPopUp

► Great

▼ Good

Name

Good

Miss

☐

Break Chain

☐

Percentage Theshold

60

Score

200

Icon

Good

Pop Prefab

GoodAccuracyPopUp Variant

► Bad

▼ Bad

Name

Bad

Miss

☐

Break Chain

☒

Percentage Theshold

100

Score

-100

Icon

Bad

Pop Prefab

BadAccuracyPopUp Variant

► Miss

▼ Rank Table

Size

7

▼ D

Name

D

Icon

D

Percentage Theshold

0

► C

► B

▼ A

Name

A

Icon

A

Percentage Theshold

70

► S

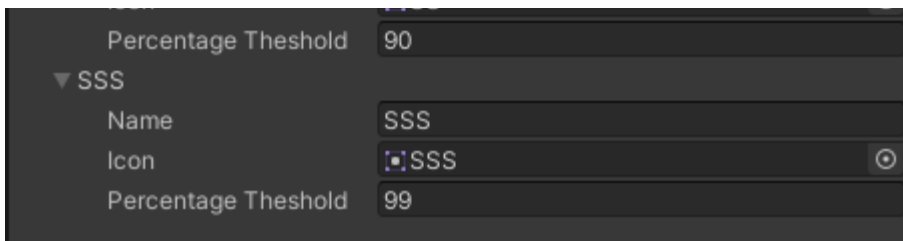
▼ SS

Name

SS

Icon

SS



The Score Manager and Score Settings are quite flexible and allow many different set ups that should accommodate most users. As the Score Settings is separate from the main core systems it can easily be replaced by your own custom score manager if required.

API

```
//Get the Score Manager from anywhere using the Toolbox.
m_ScoreManager = Toolbox.Get<ScoreManager>();

//Get the score data of the current song.
var scoreData = m_RhythmProcessor.GetScoreData();

//Get the note accuracy using the offset percentage and a bool for whether or not the no
var noteAccuracy = m_RhythmProcessor.GetAccuracy(offsetPercentage, miss);

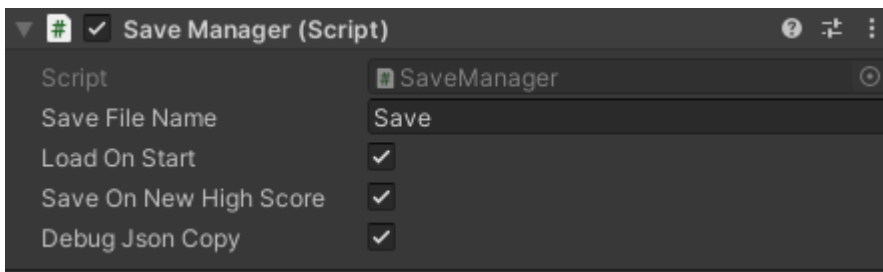
//Add score using the note and note accuracy
m_RhythmProcessor.AddNoteAccuracyScore(note, noteAccuracy);

//Add score unrelated to a specific note
m_RhythmProcessor.AddNoteAccuracyScore(score);

//Get values for rank, score, chain as units or percentages
var chain = m_RhythmProcessor.GetChain();
var chainPercentage = m_RhythmProcessor.GetChainPercentage();
var maxChain = m_RhythmProcessor.GetMaxChain();
var maxChainPercentage = m_RhythmProcessor.GetMaxChainPercentage();
var score = m_RhythmProcessor.GetScore();
var scorePercentage = m_RhythmProcessor.GetScorePercentage();
var rank = m_RhythmProcessor.GetRank();
```

Save Manager

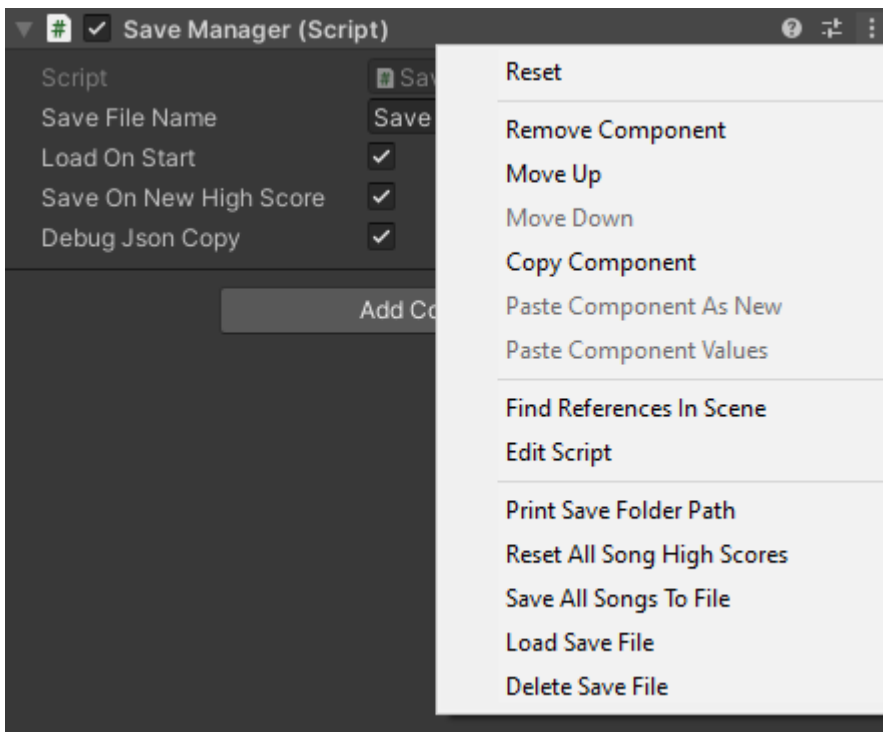
The Save Manager allows you to save the songs high score to disk. The high scores are saved on the Rhythm Timeline Asset scriptable object. scriptable objects values are stored during development even when changed at runtime. But values do not persist once the game is built. Therefore the values must be stored on disk such that they may be saved while playing on a release build on a device. The Save Manager converts the high score data of all songs into Json and then into binary. The binary data is then saved on disk as a save file. When loading, the save manager does the same in reverse.



The Save Manager listens to an event on the Score Manager to know when a new high score was made to know when it should save to disk. You may also automatically load on start.

For Debugging purposes, there is an option to print a copy of the save file in a readable Json format.

Some context menu items are available when right-clicking the Save Manager Component (or when pressing the three dots on the top right of the component)



- *Print Save Folder Path*: This prints in the console, the path to the folder where the save file will be saved.
- *Reset All Song High Scores*: Resets all the high scores for the songs that are referenced in the Rhythm Game Manager.
- *Save ALL Songs To File*: Save all songs referenced by the Rhythm Game Manager to disk.
- *Load Save File*: Load the save file by updating the high scores of matching songs.
- *Delete Save File*: Delete the save file.

The Save Manager was designed to only save the high score of the songs. It is recommended to either replace it or built on top of it to allow saving other data specific to your game too.

It is important to know that converting Json to binary is not a secure encryption solution and therefore it is advised to only save data that is relevant to the game and that does not matter if hacked. You may edit or replace the Save Manager to add encryption to your save file.

API

```
//Get the Save Manager from anywhere using the Toolbox.  
m_SaveManager = Toolbox.Get<SaveManager>();  
  
//Return the save folder path.  
var saveFolderPath = m_SaveManager.GetSaveFolderPath();  
  
//ResetAllSongHighScores  
m_SaveManager.ResetAllSongHighScores();  
  
//Save all song high score to file.  
m_SaveManager.SaveAllSongsToFile();  
  
//Load Save File.  
m_RhythmProcessor.LoadSaveData();  
  
//Delete From Disk.  
m_RhythmProcessor.DeleteFromDisk();  
  
//Save a specific song and then save to disk.  
m_RhythmProcessor.SaveSong(song);
```

Event Receivers

The event receivers are very simply components that listen to events in the system and calls a Invoke Unity Events which can be setup in the inspector.

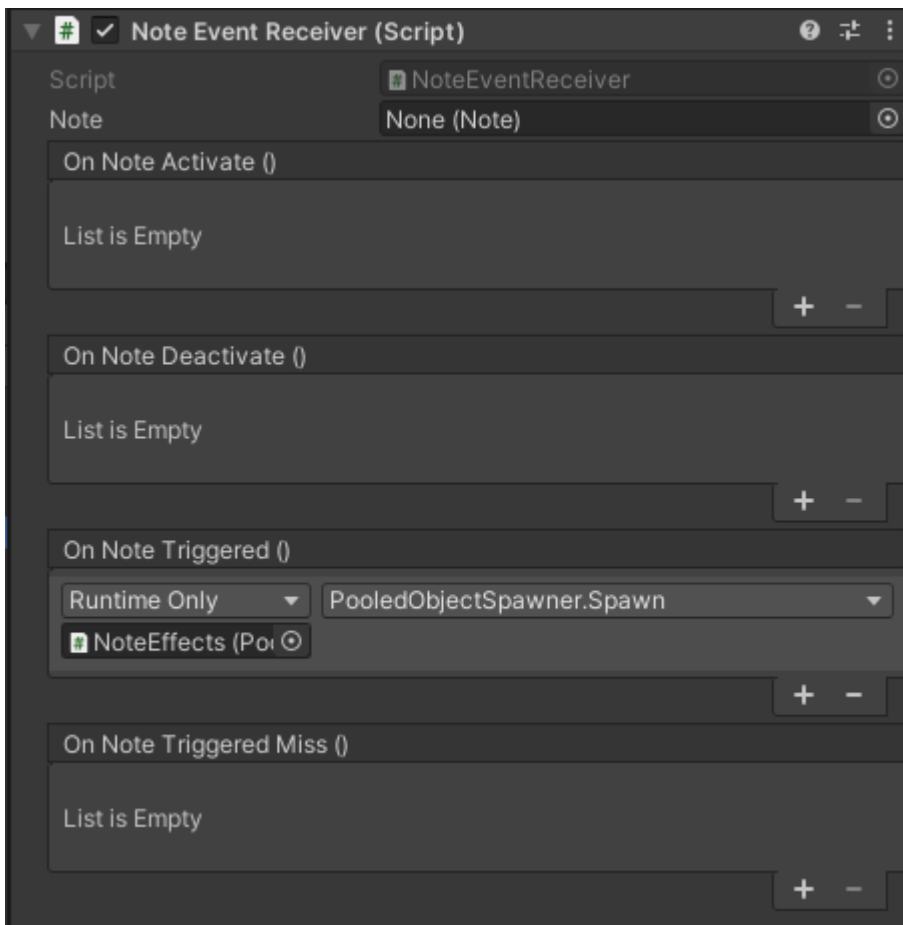
They are extremely useful to quickly customize your scene, make it more interactive a lively.

All those component can be found under the
Dypsloom/RhythmTimeline/Scripts/EventReceivers folder.

Some of the included are Event Receiver for:

- Note : detect events on a note
- Score : detect events when the score changes
- Song : detect events when the song plays or ends
- Track Input : detect inputs on tracks
- Track Note : detect notes on tracks

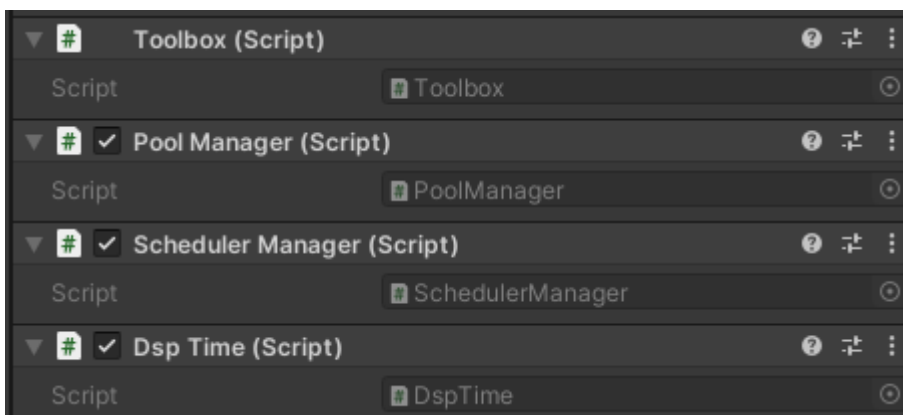
Here is an example of the Note Event Receiver



Utility Scripts

The system has a few very useful utility scripts

- Toolbox : Register and get any object from anywhere using the Toolbox
- Pool Manager : Automatically pools objects to improve performance
- Dsp Time : Get the Dsp Time or the adaptive Dsp Time from anywhere
- Scheduler : Invoke functions after a certain delay



Toolbox

The toolbox can be thought of a master singleton. instead of having many singletons in the system which can easily get out of hand. Having a single centralized singleton which manages all object is much easier to control.

Use this feature with care over relying on it will make your code hard to test outside of play mode.

```
// Register an object with an ID, by default that ID is 0
Toolbox.Set<MyObjectType>(myObject, objectID);

// Get that object anywhere using the Get function.
var myObject = Toolbox.Get<MyObjectType>(objectID);
```

Pool Manager

The pool manager automatically pools objects, it allows you to easily create pools of any game object prefab.

```
// Create a pooled instead of a prefab game object.
var pooledInstance = PoolManager.Instantiate(myPrefab);

// Return the pooled instance to the pool to be reused later.
PoolManager.Destroy(pooledInstance);
```

DSP Time

Digital Signal Processing time is not constant with the frame time. It can sometimes stay the same for multiple frames. The DSP Time component estimates an adaptive time by taking the last DSP time and adding the delta Time of all the frames where the Internal DSP time did not change. This gives a smoother interpolation when moving objects per frame in sync with the DPS time.

```
// Get the adaptive time.
var adaptiveDSPTime = DSPTime.AdaptiveTime;

// Get the rea DSP time.
var dspTime = DSPTime.Time;
```

Scheduler Manager

The scheduler is used to invoke functions delayed in seconds using coroutines.

```
// Delayed function call  
SchedulerManager.Schedule(MyFunction, delay);
```