Midity

A simple MIDI-centered rhythm game backend for Unity

Created by **Jacob Serfaty** in December 2023

- MIDI parser based on the `SmfLite.cs` MIDI Parser by Keijiro Takahashi
- If you have any questions or suggestions or if you find any bugs, please reach out to me at jserf02@gmail.com

Features- Midity

- A **MIDI parser** for note on/off statuses, velocities, and tempo changes
- A **rhythm game hit-detection script** that reads user inputs and determines whether they should hit notes in a MIDI track
    - Some supported features include:
        - Togglable press and release detection for notes
        - Occasional held notes
        - Accounting for audio latency
- A **MIDI player** that plays back a MIDI file (or MIDI track) by pitch shifting an audio clip

This package supports both the *Old Input System* and the *New Input System*, though the *New Input System* is recommended!

- The "Hit Detection Demo" scene only supports the *New Input System*.
- If you are unsure what the *New Input System* is, check out this guide: https://gamedevbeginner.com/input-in-unity-made-easy-complete-guide-to-the-new-system/

Table of Contents

Quick Start Guide

Genearlly, to use these scripts, drag them onto objects and assign the appropriate values to the component in the inspector.

Every value in the inspector is configured with a *Tooltip*, so hovering over anything will give you an explanation of what it does and how to fill it in.

More complete explanations for everything are below.

Parsing MIDI Files - `Midi.cs`

Overview

The `Midi.cs` script takes a MIDI file as input, parses it to determine tempo data, and creates a `MidiTrackReader` for each track in the MIDI file that allows continuous reading of a stream of MIDI events.

These `MidiTrackReader`s are used in the backend to handle hit detection and MIDI playback. If you are only using these scripts and do not intend to directly access the individual notes in the MIDI file, don't worry about these at all!

Setup

1. Attach a `Midi.cs` component to an object in your scene
2. Change the file extension of your MIDI file from `.mid` to `.bytes`
    - Ex: `MySong.mid` -> `MySong.bytes` or `MySong.mid.bytes` if you prefer
3. In the inspector, drag your MIDI file to the *Midi File* area
4. If you want, change the speed of the song by adjusting the value in the *Tempo Multiplier* area
    - Note: This value does not always automatically update the internal time changes when set while the game is running. It only does so whenever a track reader resets. To manually change the internal time changes to reflect this new multiplier, use the `SetTempoMultiplier()` function.
        - Setting this value before the game starts will still work as expected.

Usage

You can control this component through scripting. To call any of these functions, get a reference to this object (Ex: Add `public Midi midi;` to your code and drag the component to this new option in the inspector) and then call the functions using that reference (Ex: `midi.ChangeTempoMultiplier(2)`).

- Make sure you include `using Midity;` at the top of any file that references this component!

To parse the header and setup the tempo changes and track reader, call the `Setup()` function.

To access readers for a given MIDI track, use the `GetTrack()` function. You can also check if a track with a specific index exists using the `ContainsTrack()` function.

To change the speed of the song, use the `ChangeTempoMultiplier()` function. To set the tempo multiplier from the inspector as the internal tempo multiplier, call the `SetTempoMultiplier()` function.

To find out the tempo of the song at a given time, use the `TempoAtTime()` function.

Hit Detection - `HitDetection.cs`

Overview

The `HitDetection.cs` script plays through a track of a MIDI song and lines up user inputs with the notes of the track.

To use this script in your game, call all of your game's functions in the *Unity Events* listed in the inspector. Then, your game's function will automatically get called at the correct times by this script!

This script will call *UnityEvents* whenever:

- A note is spawned in the background
- A note is hit
- A note is missed (the player did not hit the note in time)
- The player presses or releases a button without hitting a note
- The song finishes playing

Setup

1. Attach a `HitDetection.cs` script to an object in your scene
2. If you are using the *New Input System*:
    - Go to the `PlayerInput` component in the inspector
    - Make sure *Behavior* is set to *Invoke Unity Events*
    - Click on the *Events* toggle and then sort through the remaining toggles until you find the event for the input action you want to use to hit notes
    - Click *+* to create a new event, drag the object with the `HitDetection.cs` script into the object section, and in the right side dropdown, select *HitDetection>HandleButton*
        - The *HandleButton* option should be at the top of the dropdown after selecting *HitDetection*
3. Fill out all of the parameters in the inspector as detailed below

- Note: All of this information is revealed as a *Tooltip* in the inspector when you hover over any of these parameters, so reading all of this here is not strictly necessary
- Song Settings:
  - *Midi*: The `Midi.cs` script that is setup with the MIDI file you want to play
  - *Track Idx*: The index of the MIDI Track that hit detection is based on
    - These indexes begin counting at 0 and ignore any tracks without MIDI notes
- Time Measurement Setting:
  - *Time Measurement Mode*: The unit of time used for many of the other settings
    - Choose *Seconds* if you want the amount of time for each note to be **constant regardless of the speed of the song**
    - Choose *Beats* if you want the amount of time for each note to **change with the speed of the song**
      - *Beats* means you are representing times as a number of quarter-note beats
- Detection Interval Settings:
  - *Interval Size*: The amount of time before and after a note that the player has to hit the note
    - Ex: If the Time Measurement Mode* is *Seconds* and this is set to 0.5, then the player will have 0.5 seconds before the note's actual time and 0.5 seconds after the note's actual time to hit the note, thus giving a total of 1 second for the note
  - *Preemption Time*: The amount of time before a note's start time that the note should be spawned internally
    - Useful for letting external objects know in advance that a note is coming
      - If you do not care about this, then set this parameter to 0
- Input Detection Settings:
  - *Input Mode*: The types of inputs detected by the program
    - *Press And Long Release*: Detect all button presses and only detect releases if the note is sufficiently long
      - Useful for creating games with occasional held notes
    - All other modes are self-explanatory
  - *Min Release Time*: The minimum length of a held note in the *Press And Long Release* input mode
    - This setting is ignored when the input mode is not *Press And Long Release
  - *Safety Time*: The amount of time at the start of the song where presses and releases won't call misclick events
- Events:
  - *On Spawn*: Called whenever a note of the specified type is spawned internally
  - *On Hit*: Called whenever a note of the specified type is successfully hit
  - *On Miss*: Called whenever a note of the specified type is not hit in time
  - *On Extra*: Called whenever the player makes an input of the specified type that does not hit any notes
  - *On Song Completion*: Called when the song finishes and all notes have either been hit or missed
- Other Settings:
  - *Audio Latency*: The amount of time, in seconds, after a note plays in the song before the detection period of the note should start
  - *Setup On Start*: Whether the internal setup function should be called in the Unity `Start()` function
    - Enable this if only interacting with this script through the inspector
    - Disable this if manually controlling this script through scripting
- Old Input System (These settings do not appear if the *Old Input System* is not enabled in *Player Settings*)
  - *Using Old Input System*: Check this box if you are using the *Old Input System*
    - Necessary in case users have both input systems enabled and only intend to use the *New Input System*
  - *Input Name*: The name of the input action to read button inputs from when using the Old Input System

Usage

You can control this component through scripting. To call any of these functions, get a reference to this object (Ex: Add `public HitDetection hitDetection;` to your code and drag the component to this new option in the inspector) and then call the functions using that reference (Ex: `hitDetection.Play()`).

- Make sure you include `using Midity;` at the top of any file that references this component!

To get this script ready, call the `Setup()` function or the `LoadSong()` function if you want to change the song.

You can start and stop hit detection via scripting with the `Play()` and `Stop()` functions. You can also temporarily stop and resume hit detection with the `Pause()` and `Resume()` functions.

To reset hit detection once it is complete, use the `Restart()` function.

Playing a single MIDI Track - `MidiTrackPlayer.cs`

Overview

The `MidiTrackPlayer.cs` script takes in an audio clip and repeatedly plays, pitch shifts, and stops the clip to play back a MIDI track.

This script only works for MIDI tracks that do not have polyphony (multiple notes playing at the same time), and it will throw an error if there is polyphony detected in the track.

Playing back a MIDI track in this manner is functional, though it is not recommended as the primary means of playing songs for your game. This feature is present primarily as a way for testing the *Hit Detection* system with arbitrary MIDI files.

Setup

1. Attach a `MidiTrackPlayer.cs` script to an object in your scene
2. Fill out all of the parameters in the inspector as detailed below
   - Note: All of this information is revealed as a *Tooltip* in the inspector when you hover over any of these parameters, so reading all of this here is not strictly necessary
   - Track Selection Settings:
     - *Midi*: The `Midi.cs` script that is setup with the MIDI file you want to play
     - *Track Idx*: The index of the MIDI Track that hit detection is based on
       - These indexes begin counting at 0 and ignore any tracks without MIDI notes
   - Audio Clip Settings (click the dropdown next to *Clip*)
     - *Audio Clip*: The `AudioClip` component to play and pitch shift
       - For this script to work, the `AudioClip`'s length must be longer than the maximum duration of a note in the song
       - This component does pitch shifting naively, so pitch shifting affects the speed of the song. Make sure to **pick an audio clip that is**

mostly uniform so it does not sound too weird and distorted when sped up or slowed down!
        - *Base Note*: The music note played in the audio clip, formatted as the note letter, the note octave, then whether the note is sharp or flat
            - Ex: C4#
        - *Start Time*: The timestamp to start playing the audio clip
            - Useful when the beginning of the audio clip sounds distorted when sped up but the rest of the clip is fine
    - Events
        - *Note On Event*: Called whenever a note starts playing
        - *Note Off Event*: Called whenever a note stops playing
    - Other Settings:
        - *Setup On Start*: Whether the internal setup function should be called in the Unity `Start()` function
        - Enable this if only interacting with this script through the inspector
        - Disable this if manually controlling this script through scripting

Usage

You can control this component through scripting. To call any of these functions, get a reference to this object (Ex: Add `public MidiTrackPlayer trackPlayer;` to your code and drag the component to this new option in the inspector) and then call the functions using that reference (Ex: `trackPlayer.Play()`).

- Make sure you include `using Midity;` at the top of any file that references this component!

To setup playback, use the `Setup()` function or the `Configure()` function if you want to change the inspector settings through scripting.

You can start and stop playback with the `Play()` and `Stop()` functions. You can also temporarily stop and resume hit detection with the `Pause()` and `Resume()` functions.

Playing an entire MIDI Song - `MidiPlayer.cs`

Overview

The `MidiPlayer.cs` script takes a MIDI and multiple configured audio clips as inputs and creates and manages MIDI Track Players for each track in the MIDI song in order to play back the song.

Playing back a MIDI song in this manner is functional, though it is not recommended as the primary means of playing songs for your game. This feature is present primarily as a way for testing the *Hit Detection* system with arbitrary MIDI files.

Setup

1. Attach a `MidiPlayer.cs` script to an object in your scene
2. Fill out all of the parameters in the inspector as detailed below
    - Note: All of this information is revealed as a *Tooltip* in the inspector when you hover over any of these parameters, so reading all of this here is not strictly necessary
    - *Midi*: The `Midi.cs` script that is setup with the MIDI file you want to play
    - *Tracks Settings*: A list of audio clips and events to give to each MIDI Track Player
        - The order of these items must be the same as the order of MIDI tracks with notes in the MIDI file
        - All settings for audio clips and events function the same as described in the `MidiTrackPlayer.cs` section
    - Other Settings:
        - *Tracks Parent*: The parent object for the objects with `MidiTrackPlayer.cs` components that this component will spawn
            - In most circumstances, you can set this to the transform of the object this component is attached to (drag this object into the field)
        - *Setup On Start*: Whether the internal setup function should be called in the Unity `Start()` function
        - Enable this if only interacting with this script through the inspector
        - Disable this if manually controlling this script through scripting

Usage

You can control this component through scripting. To call any of these functions, get a reference to this object (Ex: Add `public MidiPlayer player;` to your code and drag the component to this new option in the inspector) and then call the functions using that reference (Ex: `player.Play()`).

- Make sure you include `using Midity;` at the top of any file that references this component!

To setup playback, use the `Setup()` function or the `Configure()` function is you want to change the inspector settings through scripting.

You can start and stop playback with the `Play()` and `Stop()` functions. You can also temporarily stop and resume hit detection with the `Pause()` and `Resume()` functions.

Scripting Overview

To use any of the classes, structs, and enums defined in these scripts, add `using Midity` to the top of your script to open the `Midity` namespace.

Descriptions of all of these variables, properties, and functions are contained in comments in their associated files.

`Midi.cs`

`struct MidiEvent`

A single MIDI Note-On or Note-Off Event

Public Data

- `MidiEventType eventType` : The type of note event, which is either `NoteOn` or `NoteOff`
- `float time` : The time of the note in the song, in seconds
- `int note` : The MIDI note value of the note (60 is Middle C)
- `float velocity` : The MIDI velocity of the note (from 0 - 100 typically)

`struct TempoMarker`

Data associated with a tempo change

Public Data

- `int pulse` : The MIDI pulse of the song where this tempo begins
- `float time` : The time, in seconds, of the song where this tempo begins
- `float secondsPerBeat` : The new speed of the song, measured in [seconds/beat]

`struct ByteStreamReader`

A modified version of the `MidiDataStreamReader` class from `SmfLite.cs`.

Provides a stream-like interface for reading through an array of `byte`s and interpreting a collection of `bytes` as other data types in the manner dictated by the MIDI standard.

Public Methods:

- `ByteStreamReader Clone()`
- `void Advance(int length = 1)`
- `byte PeekByte()`
- `byte ReadByte()`
- `char[] ReadChars()`
- `int ReadInt32()`
- `int ReadInt16()`
- `int ReadMultiByteValue()`
- `int PeekMultiByteValue()`
- `bool AdvancePastByte(byte target)`
- `bool AdvancePastByteSequence(byte[] bytes)`

`class MidiTrackReader`

A class that manages and provides a stream-like interface for reading through a MIDI track

Public Data

- `ByteStreamReader baseReader` : A `ByteStreamReader` that points to the start of the track

Public Properties

- `bool SongComplete`
- `float CurTempo`

Public Methods:

- `MidiTrackReader Clone()`
- `List<MidiEvent> Advance(float time)`
- `void Restart()`
- `bool ContainsNotes()`
- `bool ContainsPolyphony()`

`class Midi`

Parses header information of a MIDI file, stores all tempo changes from the MIDI file, and manages `MidiTrackReader`s for each track in the MIDI file

Public Data

- `List<TempoMarker> timeChanges` : All of the tempo changes in the song
- `int pulsesPerBeat` : The number of MIDI pulses, the unit of time in MIDI files, that make up a single quarter note beat

Public Properties

- `float SlowestTempo`

Public Methods

- `bool ContainsTrack(int trackIdx)`
- `MidiTrackReader GetTrack(int trackIdx)`
- `MidiTrackReader GetTrackWithNotes(int trackIdx)`
- `void SetTempoMultiplier()`
- `void ChangeTempoMultiplier(float newMultiplier)`
- `void Setup()`
- `float TempoAtTime(float time)`

`HitDetection.cs`

`class HitDetection`

Manages a `Queue` of `MidiEvent`s from a `MidiTrackReader` with associated time intervals in order to determine when notes are hit or missed and whether the player's input is successful or a misclick.

- Triggers `UnityEvent`s whenever anything of note occurs

Public Data

The only public data is the `UnityEvent`s listed below. All of these are explained thoroughly above and in the code comments.

- `NoteSpawnEvent onPressSpawn`
- `NoteSpawnEvent onReleaseSpawn`
- `HitEvent onPressHit`
- `HitEvent onReleaseHit`
- `MissEvent onPressMiss`
- `MissEvent onReleaseMiss`
- `MisinputEvent onExtraPress`
- `MisinputEvent onExtraRelease`
- `SongCompletionEvent onSongCompletion`

Public Properties

- `float PreemptionTime`
- `float SafetyTime`
- `float AudioLatency`
- `bool IsPlaying`
- `bool SongComplete`

Public Methods

- `float IntervalSize(float secondsPerBeat)`
- `float IntervalSize(MidiEvent note)`
- `bool InNoteInterval(float time, MidiEvent note)`
- `void LoadSong(Midi newMidi, int newTrackIdx)`
  - Sets internal values that are typically setup in the inspector in order to ready this script to play a new song
  - This is the primary way of controlling this script from other scripts
  - Make sure to disable the `setupOnStart` field in the inspector if calling this function manually!
- `void Setup()`
  - Make sure to disable the `setupOnStart` field in the inspector if calling this function manually!
- `void Restart()`
- `void Play()`
- `void Stop()`
- `void Pause()`
- `void Resume()`
- `HandleButton()` if using the *Old Input System*
- `HandleButton(InputAction.CallbackContext context)` if using the *New Input System*

`MidiTrackPlayer.cs`

`struct MidiPlayingClip`

Stores information necessary for playing and pitch shifting an `AudioClip` for MIDI playing

Public Data

These variables are described in greater detail above and in the comments

- `AudioClip audioClip` : The `AudioClip` that is being described
- `string baseNote` : The music note that the audio clip plays
- `float startTime` : The starting time, in seconds, of the audio clip

`class MidiTrackPlayer`

Plays a single MIDI track by `Play`ing, `Stop`ing, changing the `volume`, and changing the `pitch` of an `AudioSource`.

Note: In order to prevent audio popping (thanks Unity...), playing a note is delayed by 2 frames and releasing a note takes 1 extra frame

- The delay for playing a note does not start until after the release timer is no longer active

Be careful playing MIDI files with this class, as pitch shifting changes the speed of the audio clip which can cause the audio to sound distorted.

Public Data

The only public data is the `UnityEvent`s listed below. All of these are explained thoroughly above and in the code comments.

- `MidiPlayerEvent noteOnEvent`
- `MidiPlayerEvent noteOffEvent`

Public Methods

- `void Configure(Midi midi, int trackIdx, MidiPlayingClip clip, MidiPlayerEvent noteOnEvent = null, MidiPlayerEvent noteOffEvent = null)`
  - Sets internal values that are typically setup in the inspector in order to ready this script to play a new song
  - This is the primary way of controlling this script from other scripts
  - Make sure to disable the `setupOnStart` field in the inspector if calling this function manually!
- `void Setup()`
  - Make sure to disable the `setupOnStart` field in the inspector if calling this function manually!
- `void Play()`
- `void Stop()`
- `void Pause()`
- `void Resume()`

`MidiPlayer.cs`

`struct MidiTrackSettings`

All of the settings other than the song and track data necessary for `Configure`ing a `MidiTrackPlayer`.

Public Data

- `MidiPlayingClip clip` : The AudioClip to play and its associated parameters
- `MidiPlayerEvent noteOnEvent` : An event that is triggered whenever a note starts playing
- `MidiPlayerEvent noteOffEvent` : An event that is triggered whenever a note stops playing

`class MidiPlayer`

Instantiates and manages `GameObjects` containing `MidiTrackPlayer` components and uses those components to play all the tracks of a MIDI file at the same time.

Note that as described in the `MidiTrackPlayer` section, all note starts are delayed by 2 frames and all note stops will take one extra frame in order to prevent audio popping.

Public Methods

- `void Configure(Midi newMidi, List<MidiTrackSettings> newTracksSettings, Transform newTracksParent = null)`
  - Sets internal values that are typically setup in the inspector in order to ready this script to play a new song
  - This is the primary way of controlling this script from other scripts
  - Make sure to disable the `setupOnStart` field in the inspector if calling this function manually!
- `void Setup()`
  - Make sure to disable the `setupOnStart` field in the inspector if calling this function manually!
- `void Play()`
- `void Stop()`
- `void Pause()`
- `void Resume()`