Sound Reactor

Manual

Use for:
Sound Reactor Standard and Pro

TOC

Copyright

Overview

SUMMARY

Sound Reactor is a Unity Extension that makes it easier than ever to make things in Unity react to sound. And now with the Pro version, you can also make things in Unity react to **MIDI** events that can be played back from midi files. It is a versatile Unity Extension that is both modular and flexible. With it, you can create stunning visualizers, and drive any property to any Unity AudioSource or **MidiSource** with ease.

Features

Some features are only available in Sound Reactor Pro.

Standard

- Easy to use and powerful **Spectrum Builder**
- Property drivers that drive: **color**, **position**, **rotation**, **scale**, **particle emitters**, and **physics forces**
- **Event handler driver** that can be used to drive any public property
- **Expandable property driver class** so you can create your own property drivers
- Remembers changes made during play mode
- A modular design that makes it possible to integrate into any project

Pro

- Drive property values with MIDI note velocities.
- Handle MIDI events directly with a custom **MIDI Event Handler**. See **MIDI** for all the events and messages Sound Reactor Pro supports.

BASICS

The setup for Sound Reactor is fairly basic. A typical setup consists of some foundation scripts, some drivers to drive properties, and GameObjects that contain said scripts.

Foundation Scripts

These scripts are necessary for any project using Sound Reactor. They are:

- **SpectrumSource**
- **SpectrumFilter**
- **Level**
- One or more drivers that inherit from **PropertyDriver**

Peaks

A peaks are used to normalize audio frequencies. Read more about what that means here: **PeaksProfile**. Peaks profiles that are included with Sound Reactor are fine for general purposes use.

Drivers

Drivers are what cause property values to change. The drivers included with Sound Reactor are:

Standard

- PositionDriver
- RotationDriver
- ScaleDriver
- ColorDriver
- ForceDriver
- ParticleEmitterDriver
- EventDriver

Pro

- MidiTrackColorDriver

Supplemental

- EQ
- Spectrum Builder

MODULAR DESIGN

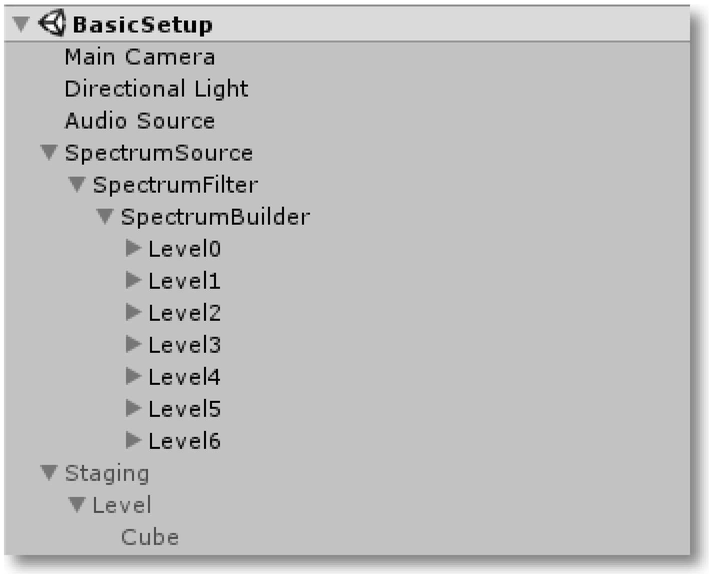
Sound Reactor is designed in such a way that connections between certain scripts can be done manually, or set up in a hierarchy so connections happen automatically. The hierarchy example in the figure below is set up so connections happen automatically.

Automatic Connections

The following scripts are arranged in the order they look for each other at runtime. i.e. **PropertyDriver** looks for **Level**, **Level** looks for **SpectrumFilter**, and so on. It does this by first looking for the script on the GameObject it's attached to, and if it doesn't find one, it travels up through each parent GameObject until it finds what it's looking for.

- PropertyDriver
- Level
- SpectrumFilter
- SpectrumSource
- AudioSource

HIERARCHY EXAMPLE



```
▼ BasicSetup
    Main Camera
    Directional Light
    Audio Source
  ▼ SpectrumSource
    ▼ SpectrumFilter
      ▼ SpectrumBuilder
        ▶ Level0
        ▶ Level1
        ▶ Level2
        ▶ Level3
        ▶ Level4
        ▶ Level5
        ▶ Level6
  ▼ Staging
    ▼ Level
        Cube
```

Basic Setup

SUMMARY

This is the most basic setup using: SpectrumSource, SpectrumFilter, Spectrum Builder, and ScaleDriver

Demos

See the *BasicSetup* scene in Demos included with Sound Reactor.

Setup

Setting up the hierarchy

1. In the hierarchy view, click: **Create->Audio->Audio Source**
2. Add an audio clip to the Audio Source. If you don't have one, you can search for the "Don't Make Me" song.
3. Right click in an empty place in the Hierarchy and select: **SoundReactor->SpectrumSource**
4. Drag Audio Source into the Audio Source property in *SpectrumSource* (leaving this property empty will cause all the sounds together to become one spectrum)
5. Right click the new *SpectrumSource* GameObject and select: **SoundReactor->SpectrumFilter**
6. Right click the new *SpectrumFilter* GameObject and select: **SoundReactor->Builder**

Staging a Level for the Builder

7. Create a GameObject called *Staging* at the root of the hierarchy
8. Right click *Staging* and select: **SoundReactor->Level**
9. Right click *Level* and select: **3D Object->Cube**
10. Click on the *Cube* and select: **Tools->SoundReactor->Driver->Scale**
11. Set the *Y* in *Travel* to 8
12. Disable the *Staging* GameObject since it's not used beyond using it to build up the spectrum.

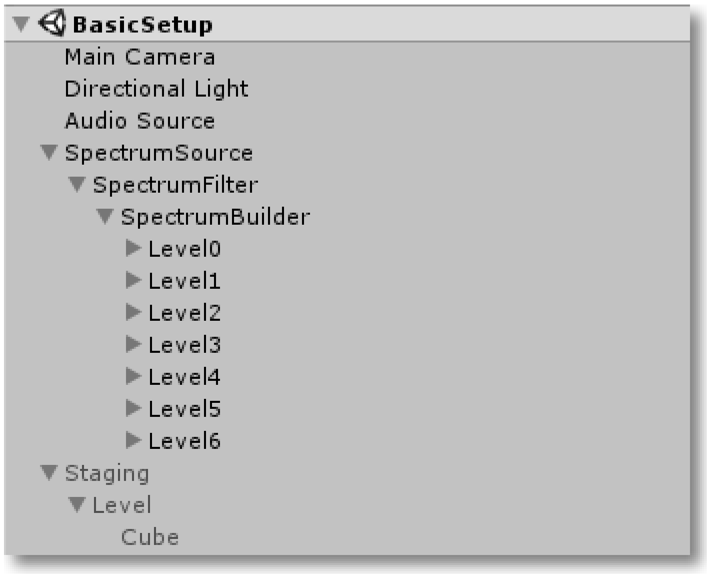
Building the spectrum visualizer

13. Click on the *SpectrumBuilder*
14. Click and drag the *Level* into the *Level* property in the *SpectrumBuilder*

Preview

15. Go into play mode and the cubes should be scaling with the audio.

RESULT



```
▼ BasicSetup
    Main Camera
    Directional Light
    Audio Source
  ▼ SpectrumSource
    ▼ SpectrumFilter
      ▼ SpectrumBuilder
        ▶ Level0
        ▶ Level1
        ▶ Level2
        ▶ Level3
        ▶ Level4
        ▶ Level5
        ▶ Level6
  ▼ Staging
    ▼ Level
        Cube
```

MIDI Setup

SUMMARY

Builds a custom piano that reacts to MIDI events.

This feature is only available in Sound Reactor Pro.

Demos

See the *MidiSetup* scene in Demos included with Sound Reactor Pro.

Setup

Add AudioMidiSync

1. Right click in an empty place in the Hierarchy and select: **SoundReactor->AudioMidiSync**.
2. With *AudioiMidiSync* still selected, add both an *AudioSource* and a *MidiSource* component.
3. Locate the K545 audio and MIDI file and attach them to the *AudioSource* and *MidiSource Source* properties.

Add SpectrumSource

4. Right click in an empty place in the Hierarchy and select: **SoundReactor->SpectrumSource**
5. Click on *SpectrumSource* then select: **Tools->SoundReactor->Component->SpectrumFilter**
6. Change the *Mode* to **Midi**
7. Drag *AudioMidiSync* into the *Source* property

Add PianoKeys

8. Create a new GameObject called *Staging* at the root and disable it.
9. Create a new child GameObject under *Staging* called *PianoKeys*.
10. Locate the PianoKeys asset folder. Drag all the piano keys into the *PianoKeys* GameObject.
11. Select all the piano keys under the *PianoKeys* GameObject and add a *MidiTrackColorDriver* and a *RotationDriver* script to them.
12. Select keys: AS, CS, DS, FS and GS, and set their resting color to black.
13. Set the *Travel* for the *RotationDriver* to: (-3.5, 0, 0), and the *Strength* to 2

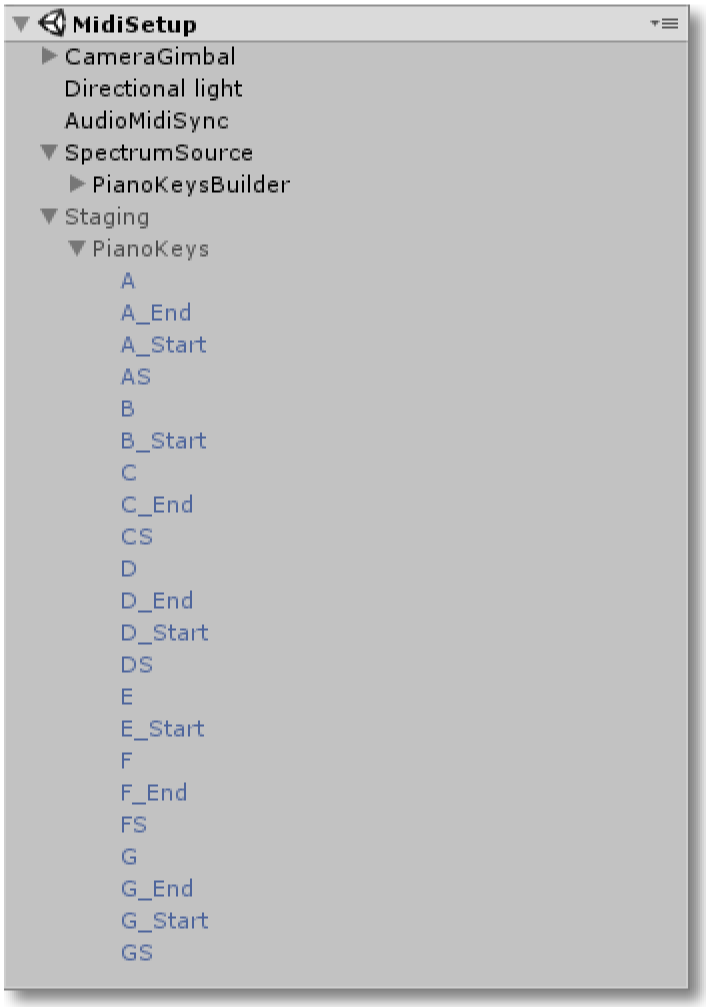
Add SpectrumBuilder

14. Right click on the *SpectrumSource* and select: **SoundReactor->Builder**.
15. Select the *SpectrumBuilder* and set it's mode to *Piano*.
16. Drag the *PianoKeys* GameObject that was created at step 9 into the SpectrumBuilder's *PianoKeys* property.
17. Set the *Frequency Range->Mode* to **Midi**
18. Set the *Frequency Range->Preset* to **88 Keys (C)**
19. Set *Layout->Levels* to **88** to match the preset.
20. Set the *Level->Start Note* to the note indicated by *Frequency->Preset*, which is **C** in this case.
21. Click **Assign**.
22. Click **Build**.

Preview

23. Go into play mode and the keys should animate and change color to the MIDI.

RESULT



```
▼ MidiSetup
  ▶ CameraGimbal
    Directional light
    AudioMidiSync
  ▼ SpectrumSource
    ▶ PianoKeysBuilder
  ▼ Staging
    ▼ PianoKeys
        A
        A_End
        A_Start
        AS
        B
        B_Start
        C
        C_End
        CS
        D
        D_End
        D_Start
        DS
        E
        E_Start
        F
        F_End
        FS
        G
        G_End
        G_Start
        GS
```

Event Driver Setup

SUMMARY

Setup a scene that drives properties.

Demos

See *EventDriverSetup* scene in Demos included with Sound Reactor.

Setup

Event Handler

```
using UnityEngine;
using LDG.SoundReactor;

public class EmissionHandler : MonoBehaviour
{
    public Color emissionColor;

    private Material material;
    private int emissionColorID;

    private void Start()
    {
        MeshRenderer meshRenderer;

        if ((meshRenderer = GetComponent<MeshRenderer>()))
        {
            if ((material = meshRenderer.material) != null)
            {
                emissionColorID = Shader.PropertyToID("_EmissionColor");
            }
        }
    }

    public void OnLevel(PropertyDriver driver)
    {
        if (material)
        {
            float level = driver.LevelScalar();

            material.SetColor(emissionColorID, emissionColor * level);
        }
    }
}
```

Create the class

1. Create the above EventHandler class and call it **EmissionHandler**

Basic Setup

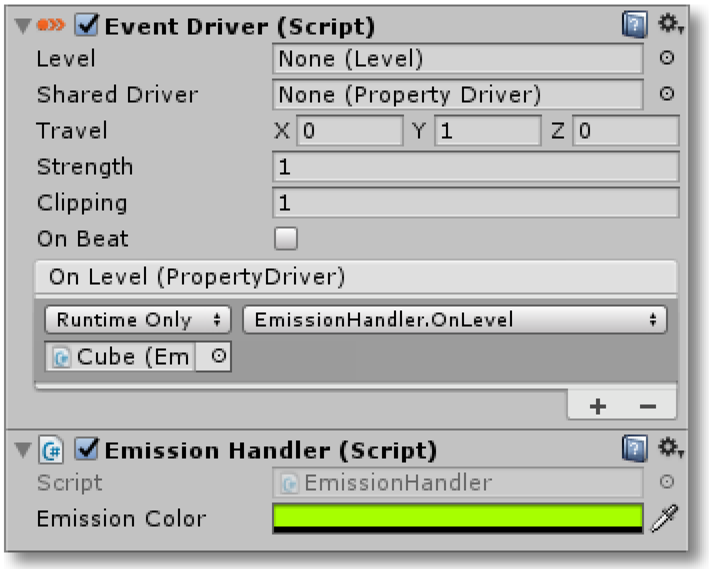
2. Follow the **Basic Setup**.

Attaching the Handler

3. Attach the **EmissionHandler** class to the *Cube*.
4. Create a new *OnLevel* handler by pressing the '+' button.
5. Attach the *Cube* to the GameObject property.

6. Select *EmissionHandler->OnLevel* from the function drop down menu. Make sure to choose the function located at the top of the list, which is the **dynamic** version.

RESULT



Event Driver (Script)
Level: None (Level)
Shared Driver: None (Property Driver)
Travel: X 0 Y 1 Z 0
Strength: 1
Clipping: 1
On Beat: ☐
On Level (PropertyDriver)
Runtime Only | EmissionHandler.OnLevel
Cube (Em

Emission Handler (Script)
Script: EmissionHandler
Emission Color: 🟩

MIDI Event Handler

SUMMARY

Code snippet to handle MIDI events manually.

This feature is only available in Sound Reactor Pro.

Demos

See the *MidiEventInfo* scene in Demos included with Sound Reactor Pro

Setup

MIDI Event Handler

```
public void OnMidiEvent(Sequencer sequencer, MidiEvent e)
{
    if (!this.enabled || !gameObject.activeSelf) return;

    this.sequencer = sequencer;

    switch(sequencer.PlayState)
    {
        case MidiPlayState.End:
            // reached the end of the MidiClip
            break;

        case MidiPlayState.Play:
            // the Play function has been called
            break;

        case MidiPlayState.Stop:
            // the Stop function has been called
            break;
    }

    // all possible channel voice messages
    if (e.IsChannelVoiceMessage)
    {
        switch(e.ChannelVoiceMessage)
        {
            case ChannelVoiceMessage.NoteOff:
                // handle message
                break;
            case ChannelVoiceMessage.NoteOn:
                // handle message
                break;
            case ChannelVoiceMessage.PolyphonicPressure:
                // handle message
                break;
            case ChannelVoiceMessage.ControlChange:
                // handle message
                break;
            case ChannelVoiceMessage.ProgramChange:
                // handle message
                break;
            case ChannelVoiceMessage.ChannelPressure:
                // handle message
                break;
            case ChannelVoiceMessage.PitchWheelChange:
                // handle message
                break;
        }
    }

    // all possible meta messages
    if (e.IsMetaMessage)
    {
        switch (e.MetaMessage.MetaType)
        {
            case MetaType.Text:
                // handle message
                //Debug.Log("text: " + e.MetaMessage.Text);
                break;
            case MetaType.CopyrightNotice:
                // handle message
                //Debug.Log("copyright notice: " + e.MetaMessage.CopyrightNotice);
                break;
            case MetaType.TrackName:
                // handle message
                //Debug.Log("track name: " + e.MetaMessage.TrackName);
                break;
            case MetaType.InstrumentName:
                // handle message
                //Debug.Log("instrument name: " + e.MetaMessage.InstrumentName);
                break;
            case MetaType.Lyrics:
                // handle message
                //Debug.Log("lyrics: " + e.MetaMessage.Lyrics);
                break;
            case MetaType.Marker:
                // handle message
                //Debug.Log("marker: " + e.MetaMessage.Marker);
                break;
            case MetaType.CuePoint:
                // handle message
                //Debug.Log("cue point: " + e.MetaMessage.CuePoint);
                break;
            case MetaType.ChannelPrefix:
                // handle message
                //Debug.Log("channel prefix: " + e.MetaMessage.ChannelPrefix);
                break;
            case MetaType.EndOfTrack:
                // handle message
                //Debug.Log("end of track");
                break;
            case MetaType.Tempo:
                // handle message
                //Debug.Log("tempo: " + e.MetaMessage.Tempo);
                break;

            case MetaType.SMPTEOffset: // handled, but not supported
                // handle message
                /*
                Debug.Log
            case MetaType.SequenceNumber:
                  // handle message
                  //Debug.Log("sequence number: " + e.MetaMessage.SeqeunceNumber);
                  break;
          }
      }
}
```

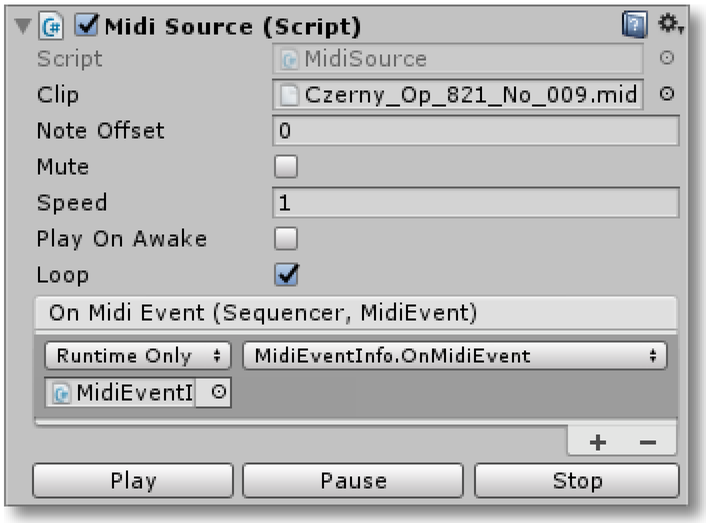
Basic Setup

1. Follow the MidiSetup.

Attaching the handler

2. Create or open a MonoBehaviour script and paste in the above method.
3. Select the GameObject that has a MidiSource component attached to it.
4. Press the '+' button to add a OnMidiEvent handler.
5. Drag the GameObject that has your OnMidiEvent into the Object property.
6. Select OnMidiEvent from the method drop down list.

RESULT



Midi Source (Script)

Script	MidiSource
Clip	Czerny_Op_821_No_009.mid
Note Offset	0
Mute	☐
Speed	1
Play On Awake	☐
Loop	✓

On Midi Event (Sequencer, MidiEvent)

Runtime Only | MidiEventInfo.OnMidiEvent

MidiEventI

MIDI File Setup

SUMMARY

Before a MIDI file can be attached to a **MidiSource** it must be converted to a *.asset* file first.

This feature is only available in Sound Reactor Pro.

Create a MIDI .asset file

Any of the following methods will create a *.asset* file from a valid MIDI file. See the **MIDI** specifications for which MIDI files are valid.

- Right click on a MIDI file and select: **Create->SoundReactor->Midi Clip** in the **Project** tab.
- Click on a MIDI file and select: **Create->Assets->SoundReactor->Midi Clip** in the **Project** tab.
- Click on a MIDI file and select:**Tools->SoundReactor->Midi Clip** in the tool bar.

RESULT

The result will be a new file with the extension *.midi.asset*, where *.midi* will show in the inspector, and *.asset* will be hidden from the inspector.

MIDI Load Resource

SUMMARY

Reads a MIDI *.asset* file that's inside a *Resources* folder and attaches it to a **MidiSource** at runtime. It also loads an **AudioClip** and attaches it to an AudioSource.

See **MIDI File Setup** for creating a MIDI *.asset* file.

This feature is only available in Sound Reactor Pro.

MIDI LOAD RESOURCE CLASS

```
using UnityEngine;
using LDG.MIDI;

public class MidiLoadResource : MonoBehaviour
{
    public string midiResource;
    public string audioResource;

    public MidiSource midiSource;
    public AudioSource audioSource;

    // Use this for initialization
    void Start()
    {
        MidiClip midiClip;
        AudioClip audioClip;

        midiSource = (midiSource) ? midiSource : gameObject.GetComponent<MidiSource>
();
        audioSource = (audioSource) ? audioSource : gameOb-
ject.GetComponent<AudioSource>();

        if (midiSource && audioSource)
        {
            midiClip = Resources.Load<MidiClip>(midiResource);
            midiSource.clip = midiClip;

            audioClip = Resources.Load<AudioClip>(audioResource);
            audioSource.clip = audioClip;

            audioSource.Play();
            midiSource.Play();
        }
    }
}
```

Demos

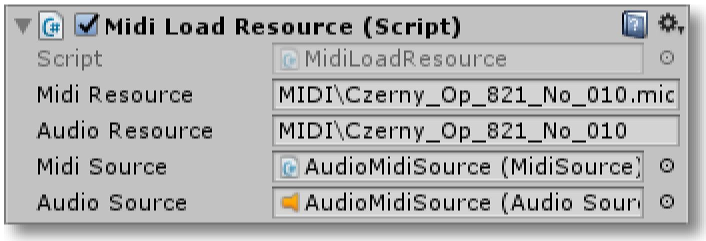
See the *MidiLoadResource* scene in Demos included with Sound Reactor Pro.

Setup

1. Create a "Resources" folder inside Unity's "Assets" folder.
2. Create a path inside the "Resources" folder where you'd like your MIDI files to be.

3. Create the above script in your project. *If the Demo was unpackaged then this script already exists in your project.*
4. Attach this script to a GameObject.
5. Provide the Resources path to your audio and midi files. *If you don't have a corresponding audio file, then edit the script to remove that dependency.*
6. Drag an *AudioSource* and **MidiSource** from your scene into the property fields. *If you don't have a corresponding audio file, then edit the script to remove that dependency.*

RESULT



Midi Load Resource (Script)

Script	MidiLoadResource
Midi Resource	MIDI\Czerny_Op_821_No_010.mic
Audio Resource	MIDI\Czerny_Op_821_No_010
Midi Source	AudioMidiSource (MidiSource)
Audio Source	AudioMidiSource (Audio Sour

Record Peaks

SUMMARY

Peaks are used to scale frequency magnitudes (**Levels**) to a range somewhere between 0 to 1. Without peaks, the values are to arbitrary to trigger events reliably. For this reason, **PeakProfiles** can be created and shared among any audio clip. See **Audio Peaks** for a more detailed explanation.

NOTE: If you're not looking to create a custom **PeakProfile**, Sound Reactor includes several that will work with any audio clip.

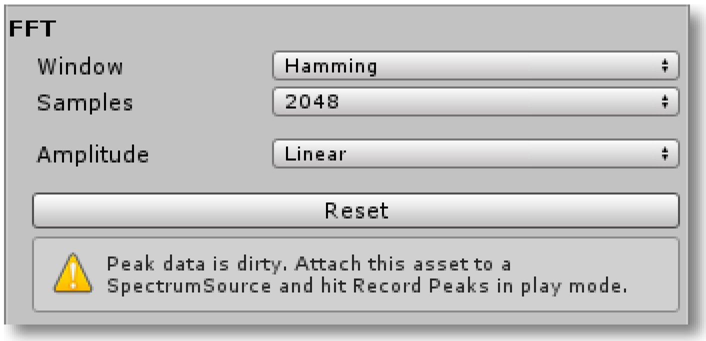
Creating a PeaksProfile

There are multiple ways to create a **PeaksProfile** file. Here are the most common ways:

- Right click on an audio file and select: **Create->SoundReactor->Peaks Profile**
- Select: **Create->Assets->SoundReactor->Peaks Profile**
- Select: **Tools->SoundReactor->Peaks Profile**

Recording a PeaksProfile

1. First the profile must be made dirty. To make it dirty, either change one of its settings, or press the Reset button in the inspector.



FFT

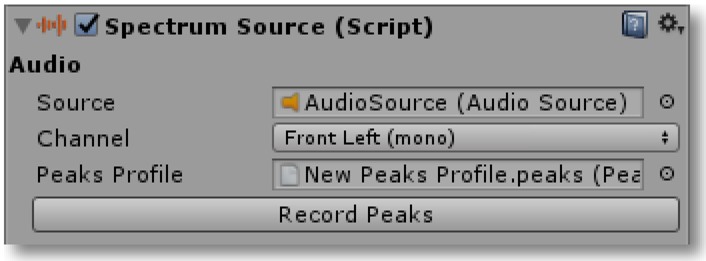
Window: Hamming

Samples: 2048

Amplitude: Linear

Peak data is dirty. Attach this asset to a SpectrumSource and hit Record Peaks in play mode.

2. Attach it to a **SpectrumSource**.
3. Attach an AudioSource that is pointing to an AudioClip you'd like to record peaks for. To generate generic peaks, record the peaks of a **Sweep Sound**.
4. Go into play mode and a *Record Peaks* button will appear just under where the profile was attached to in the **SpectrumSource**. Press the button.



Spectrum Source (Script)

Audio

Source: AudioSource (Audio Source)

Channel: Front Left (mono)

Peaks Profile: New Peaks Profile.peaks (Pea

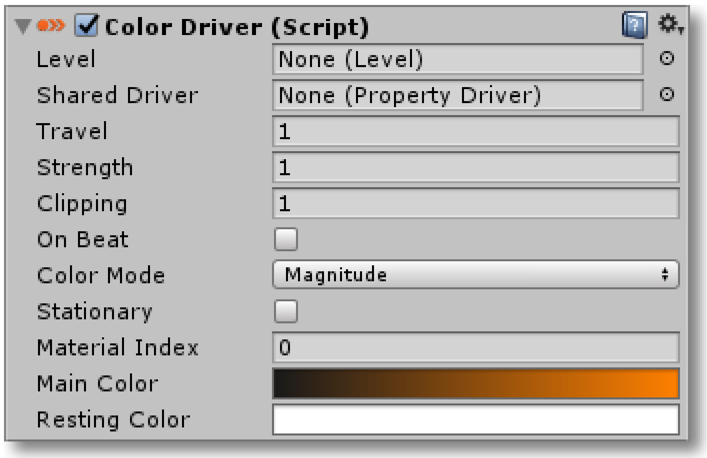
5. The peaks are done recording when the end of the AudioClip has been reached. Sound Reactor will confirm this by hiding the record button and posting a message in the console.

Color Driver

SUMMARY

This driver changes the color of: **materials**, **particles**, and **vectors**.

INSPECTOR



Color Driver (Script)

Level	None (Level)
Shared Driver	None (Property Driver)
Travel	1
Strength	1
Clipping	1
On Beat	☐
Color Mode	Magnitude
Stationary	☐
Material Index	0
Main Color	⬛🟧
Resting Color	⬜

PROPERTIES

Inherits From

[PropertyDriver]

Color Mode

The method the color gradient is applied.

- **Magnitude** Sets the gradient color using the level's magnitude.
- **Frequency** Sets the gradient color using the level's frequency.

Stationary

Check this if a Segmented Levels shape was built with the **SpectrumBuilder**. This option only displays when Color Mode is set to Magnitude.

Material Index

The index to a material for the attached mesh.

Main Color

The main color used to colorize the object. This changes the `_Color` property in the shader. If a `_Color` property doesn't exist, then the object will not be colorized.

Resting Color

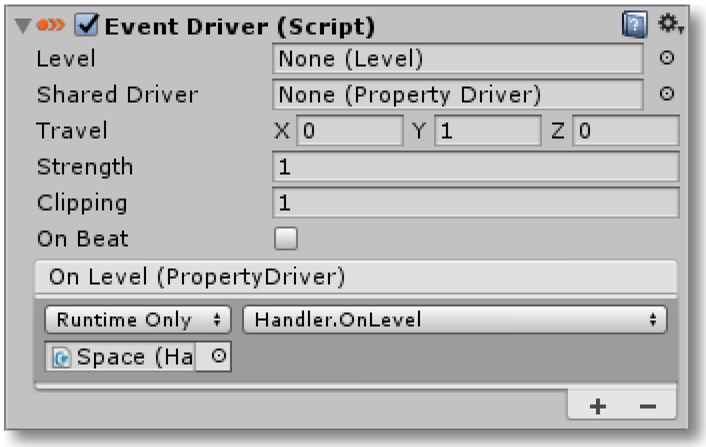
This color is applied to segmented levels, and vector shapes that are anchored.

Event Driver

SUMMARY

An easy way to animate properties in a class.
See **EventDriverSetup** for setup.

INSPECTOR



Event Driver (Script)

Level	None (Level)
Shared Driver	None (Property Driver)
Travel	X 0 Y 1 Z 0
Strength	1
Clipping	1
On Beat	☐

On Level (PropertyDriver)

Runtime Only | Handler.OnLevel

Space (Ha

PROPERTIES

Inherits From

PropertyDriver

OnLevel

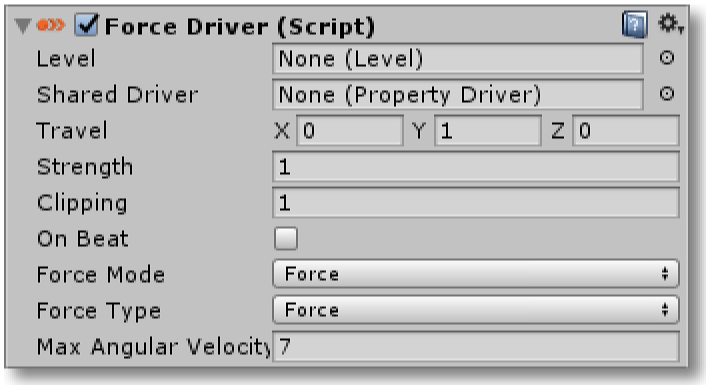
Takes an event handler that can process **PropertyDriver** values.

Force Driver

SUMMARY

Applies a force to a GameObject that has a RigidBody attached to it.

INSPECTOR



Force Driver (Script)

Level	None (Level)
Shared Driver	None (Property Driver)
Travel	X 0 Y 1 Z 0
Strength	1
Clipping	1
On Beat	☐
Force Mode	Force
Force Type	Force
Max Angular Velocity	7

PROPERTIES

Inherits From

PropertyDriver

Force Mode

How the force should be applied to the RigidBody.

- Force
- Impulse
- Velocity Change
- Acceleration

Force Type

What type of force to apply

- Force
- Torque
- Relative Force
- Relative Torque

Max Angular Velocity

Limits the angular velocity

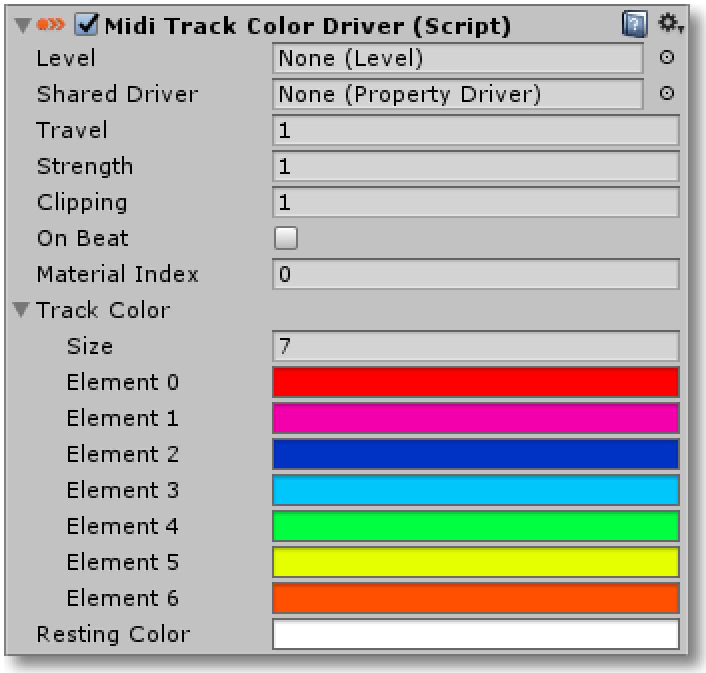
Midi Track Color Driver

SUMMARY

This driver changes the color of the notes associated with a particular MIDI track.

This feature is only available in Sound Reactor Pro.

INSPECTOR



Midi Track Color Driver (Script)

Level	None (Level)
Shared Driver	None (Property Driver)
Travel	1
Strength	1
Clipping	1
On Beat	☐
Material Index	0
Track Color	
Size	7
Element 0	🟥
Element 1	🟪
Element 2	🟦
Element 3	🟦
Element 4	🟩
Element 5	🟨
Element 6	🟧
Resting Color	⬜

PROPERTIES

Inherits From

PropertyDriver

Material Index

The index to a material for the attached mesh.

Track Color

The colors used to color tracks. If there are more tracks than colors, then colors will repeat. This changes the `_Color` property in the shader. If a `_Color` property doesn't exist, then the object will not be colorized.

Resting Color

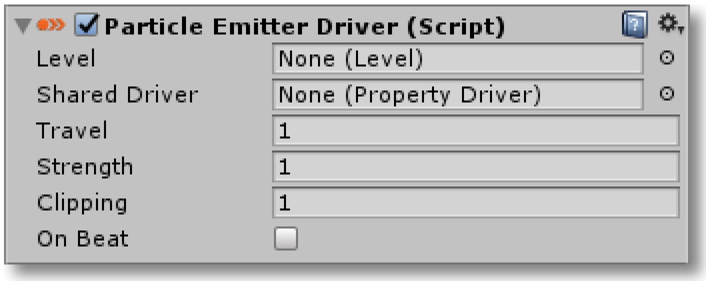
This is the color used when the MIDI note velocity is close to zero.

Particle Emitter Driver

SUMMARY

Emits specified number of particles.

INSPECTOR



Particle Emitter Driver (Script)

Level	None (Level)
Shared Driver	None (Property Driver)
Travel	1
Strength	1
Clipping	1
On Beat	☐

PROPERTIES

Inherits From

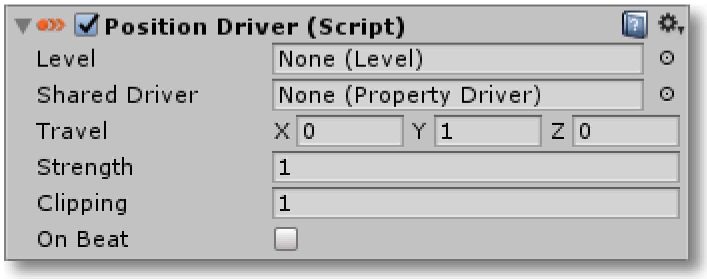
PropertyDriver

Position Driver

SUMMARY

Moves an object.

INSPECTOR



Position Driver (Script)

Level	None (Level)
Shared Driver	None (Property Driver)
Travel	X 0 Y 1 Z 0
Strength	1
Clipping	1
On Beat	☐

PROPERTIES

Inherits From

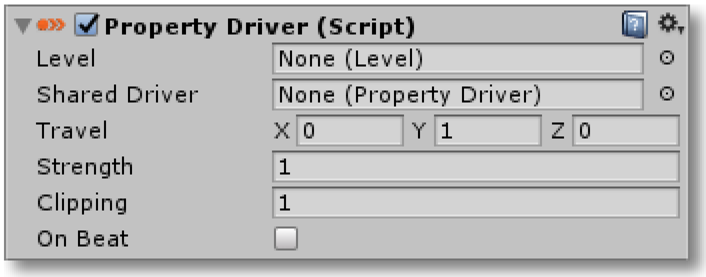
PropertyDriver

Property Driver

SUMMARY

PropertyDrivers use the magnitude of a frequency stored in a **Level** to drive property values. This class is meant to be inherited from. All its properties are common and editable in the following scripts included with Sound Reactor: **PositionDriver**, **RotationDriver**, **ScaleDriver**, **ParticleEmitterDriver**, **ColorDriver**, **ForceDriver**, and **EventDriver**.

INSPECTOR



Property Driver (Script)	
Level	None (Level)
Shared Driver	None (Property Driver)
Travel	X 0 Y 1 Z 0
Strength	1
Clipping	1
On Beat	☐

PROPERTIES

Level

The **Level** this driver uses to calculate the travel distance. If this is set to None, then it'll try to find a Level at runtime on the GameObject it's attached to by looking up through the hierarchy.

Shared Driver

Travel, Max Level, Strength, and On Beat are all overridden by this driver.

Travel

Defines the travel distance. If a level reaches 100% of its magnitude, and the travel is set to 10, then the travel distance will be 10.

Strength

Scales the level's magnitude. This is useful to get values to reach their max travel distance sooner.

Clipping

Clips the level's magnitude. A value of 1 will allow the level to reach 100% of its magnitude.

On Beat

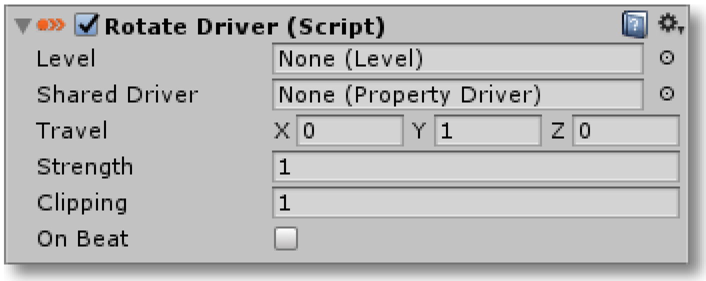
Causes the Travel to reach 100% once per frame per beat. This is useful for physics and particle emitters.

Rotation Driver

SUMMARY

Rotates object.

INSPECTOR



Rotate Driver (Script)

Level	None (Level)
Shared Driver	None (Property Driver)
Travel	X 0 Y 1 Z 0
Strength	1
Clipping	1
On Beat	☐

PROPERTIES

Inherits From

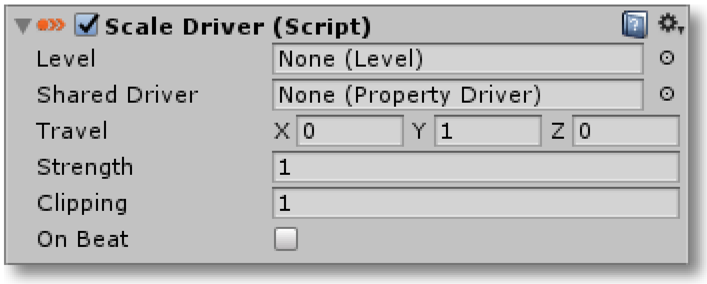
PropertyDriver

Scale Driver

SUMMARY

Scales object.

INSPECTOR



Scale Driver (Script)

Level	None (Level)
Shared Driver	None (Property Driver)
Travel	X 0 Y 1 Z 0
Strength	1
Clipping	1
On Beat	☐

PROPERTIES

Inherits From

PropertyDriver

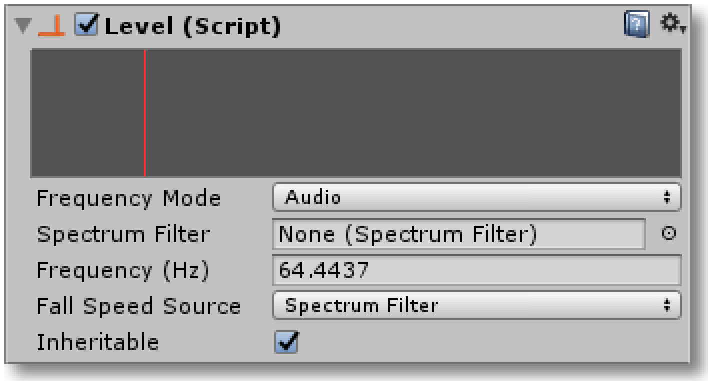
Level

SUMMARY

A Level points to a specific frequency in **SpectrumSource**, and its value is the magnitude of that frequency after it has been **normalized**. The level itself is always falling at a certain rate, which is defined by the **SpectrumFilter** it points to.

Some features are only available in Sound Reactor Pro.

INSPECTOR



Frequency Mode	Audio
Spectrum Filter	None (Spectrum Filter)
Frequency (Hz)	64.4437
Fall Speed Source	Spectrum Filter
Inheritable	✓

PROPERTIES

Frequency Mode

Affects which frequency values should be displayed. Only available in Sound Reactor Pro.

- **Audio** is ~20Hz to ~20KHz
- **MIDI** is ~8.18Hz to ~12543.85KHz

Spectrum Filter

The filter to grab spectrum data from. If this is set to None, then it'll try to find a **SpectrumFilter** at runtime on the GameObject it's attached to, then by looking up through the hierarchy for the first **SpectrumFilter** it finds.

Frequency (Hz)

The audio frequency that the level is tracking. You can either set the frequency directly in the edit field, or by left clicking the mouse inside the frequency window. When using the **SpectrumBuilder**, the frequency is automatically set for each level created.

Fall Speed Source

The source to get the fall speed from.

- **Spectrum Filter** gets the fall speed from the *Spectrum Filter→Fall Speed* property
- **Level** gets the fall speed from this *Level→Fall Speed* property. The *Fall Speed* property shows when this option is set.

Inheritable

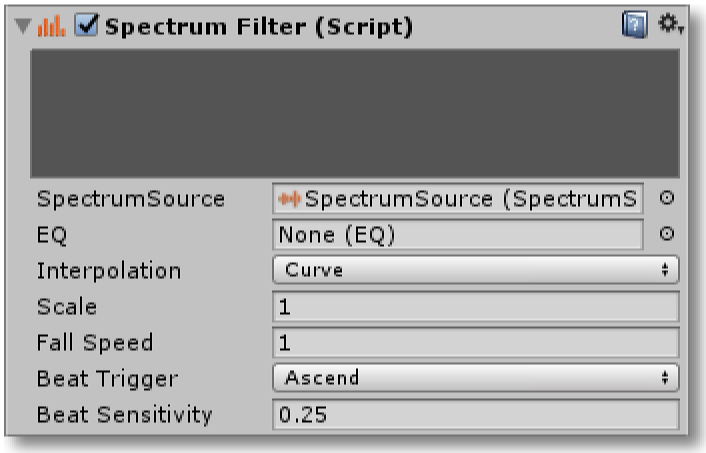
Tells a child **PropertyDriver** if it can inherit a Level from another GameObject or not. The only exception is if the **PropertyDriver** is attached to the same object as the **Level**.

Spectrum Filter

SUMMARY

The SpectrumFilter is used to modify the shape and scale of the spectrum. It also allows you to set the falling speed and beat detection sensitivity of all levels parented to this filter.

INSPECTOR



SpectrumSource	SpectrumSource (SpectrumS
EQ	None (EQ)
Interpolation	Curve
Scale	1
Fall Speed	1
Beat Trigger	Ascend
Beat Sensitivity	0.25

PROPERTIES

Frequency Window

Red Line Represents the frequency.
Green Line Represents the normalized spectrum.
Blue Line Represents the falling level.
Black Line Will change positions whenever the threshold under *Beat Sensitivity* is reached.

Spectrum Source

The **SpectrumSource** to filter. If this is set to None, then it'll try to find a **SpectrumSource** at runtime on the GameObject it's attached to, then by looking up through the hierarchy for the first **SpectrumSource** it finds.

EQ

This gives you a little more control over how the spectrum is scaled. See **EQ** in this document for more details.

Interpolation

Sets whether values in between levels are interpolated as straight lines or curved.

Scale

Scales all incoming band magnitudes acquired from the attached **SpectrumSource**.

Fall Speed

The speed at which the levels should fall. This is in **normalized space**, so setting the value to 1 will bring the level from full height down to zero in 1 second. Setting this value to 2 would bring it all the way down in a half a second.

Beat Trigger

Tells a beat to be triggered when the level: Ascends, Descends, or both.

Beat Sensitivity

Just like fall speed, this value is in **normalized space**. Setting this value to 0.75 would cause a beat to happen when a level descends 75%. Setting the value to 0.5 would cause a beat to happen when the level descends 50%.

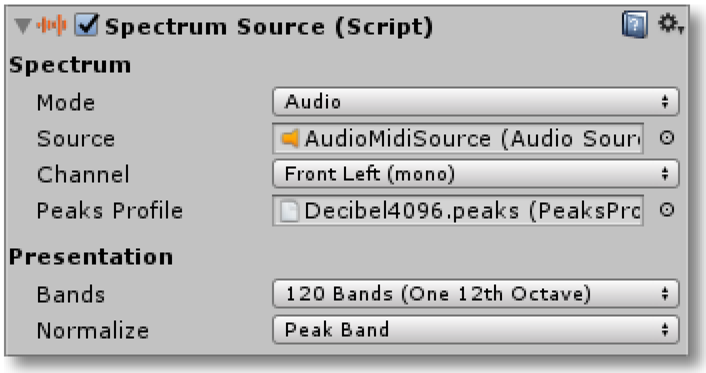
Spectrum Source

SUMMARY

Converts a *AudioSource* or *MidiSource* to spectrum data, and then processes it for output to **SpectrumFilter**.

Some features are only available in Sound Reactor Pro.

INSPECTOR



Spectrum Source (Script)

Spectrum

Mode	Audio
Source	AudioMidiSource (Audio Sour
Channel	Front Left (mono)
Peaks Profile	Decibel4096.peaks (PeaksPrc

Presentation

Bands	120 Bands (One 12th Octave)
Normalize	Peak Band

PROPERTIES

Mode

Tells the *SpectrumSource* what source to use. It can either be Audio or Midi.

Source

The source to pull spectrum data from. In Sound Reactor Pro this can either be Midi or Audio.

Channel

The audio channel to pull spectrum data from. If an *AudioSource* is attached and points to a valid *AudioClip*, this channel will be restricted to the highest supported channel for that clip.

Peaks Profile

A profile that stores peak data along with sample rate and amplitude type.

Bands

This sets how many bands the spectrum will be divided into.

Normalize

Normalizes the attached levels to a range of 0 to 1.

- **Peak** Normalizes all levels based on the spectrum's peak.
- **Peak Band** Normalizes each level based on its own peak.
- **Raw** No scaling whatsoever, i.e. the unprocessed data straight from Unity’s GetSpectrumData function.

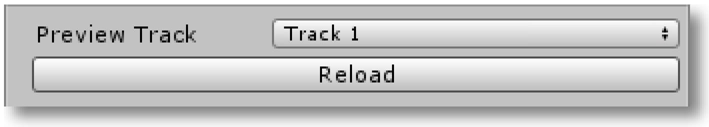
Midi Clip

SUMMARY

A MidiClip is a `ScriptableObject` that has been generated from a .mid or .midi file. It contains most of what is stored in a MIDI file, and some additional things that make it easier for developers to use it in games.

This feature is only available in Sound Reactor Pro.

INSPECTOR



Preview Track: Track 1

PROPERTIES

Preview Track

The track to preview in the preview window.

BUTTONS

Reload

Reloads the MIDI file that this asset was generated from.

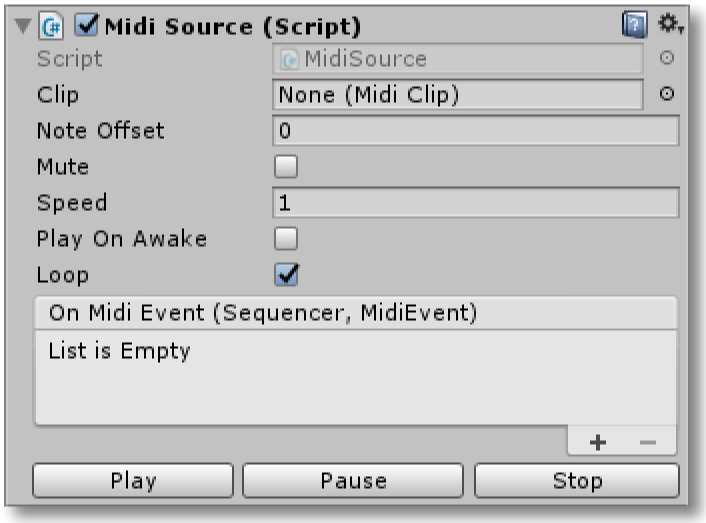
Midi Source

SUMMARY

Plays a `MidiClip`. The properties for this class are similar to Unity's `AudioSource` class.

This feature is only available in Sound Reactor Pro.

INSPECTOR



Midi Source (Script)

Script	MidiSource
Clip	None (Midi Clip)
Note Offset	0
Mute	☐
Speed	1
Play On Awake	☐
Loop	✓

On Midi Event (Sequencer, MidiEvent)

List is Empty

PROPERTIES

Clip

MidiClip to play.

Note Offset

Offsets the MIDI note being played. The new offset value is what gets passed to **SpectrumSource** and `OnMidiEvent`.

Mute

Sound Reactor stops reacting, and OnMidiEvent stops being called, but time continues to pass.

Speed

Sets the playback speed. A value of 1 will play the MIDI sequence at normal speed.

Play On Awake

Automatically plays the MIDI sequence when the scene is loaded.

Loop

When set to true, the MIDI sequence will start over again at 0 when the end of the sequence is reached.

On Midi Event

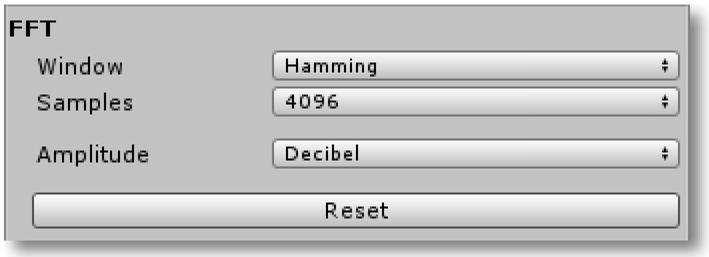
Takes a custom `MidiEventHandler`.

Peaks Profile

SUMMARY

The PeaksProfile contains peaks used for normalizing spectrum magnitudes. See **Record Peaks** for recording custom peaks, otherwise the ones included with Sound Reactor work just fine.

INSPECTOR



FFT

Window: Hamming

Samples: 4096

Amplitude: Decibel

PROPERTIES

Window

The quality of the spectrum data. The list is sorted so that low quality starts at the top, and high quality at the bottom. The higher the quality the lower the performance, and vise versa.

Samples

The number of samples. The higher the number the higher the quality. The higher the quality the lower the performance, and vise versa.

Amplitude

Sets whether to use decibel or linear magnitude.

BUTTONS

Reset

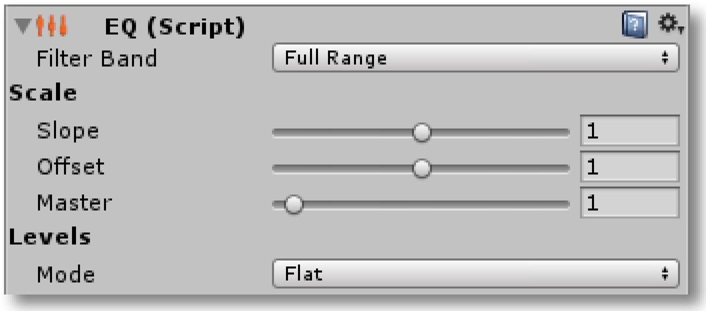
Button that marks the peak as dirty so it can be recorded to again.

EQ

SUMMARY

Modifies the **normalized magnitudes** of the bands coming out of **SpectrumSource**.

INSPECTOR



EQ (Script)

Filter Band: Full Range

Scale

Slope: 1

Offset: 1

Master: 1

Levels

Mode: Flat

PROPERTIES

Filter Band

All bands outside the specified range will be scaled to zero.

Slope

Scales the slope defined in Mode.

Offset

Offsets all the levels.

Master

Scales the levels

Mode

Defines a shape for the levels.

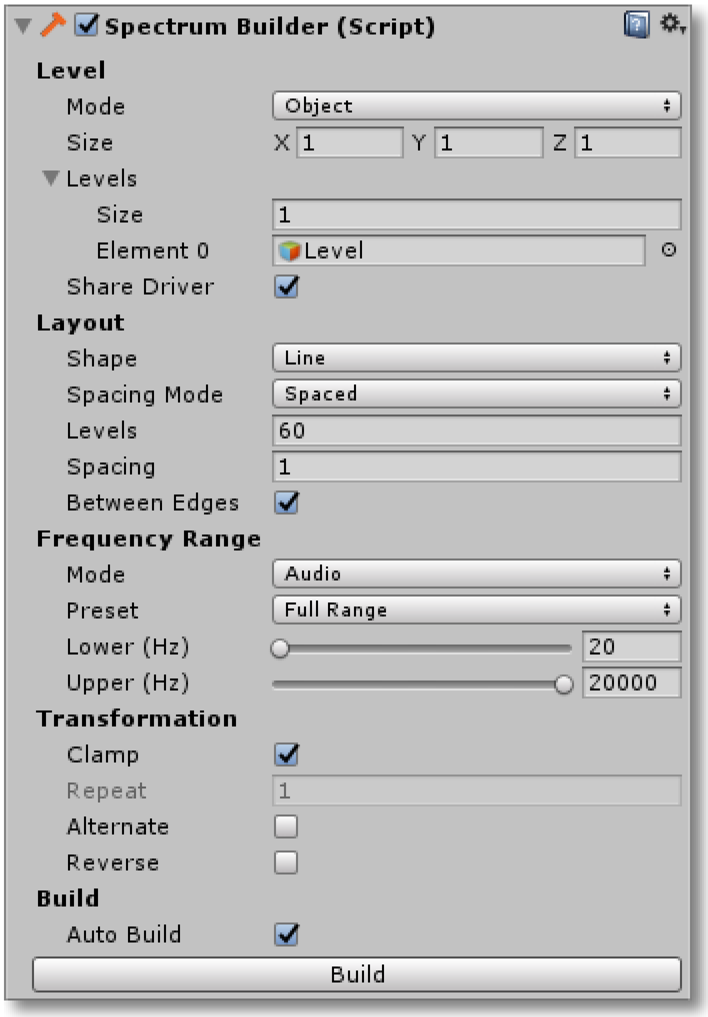
Spectrum Builder

SUMMARY

The SpectrumBuilder is the quickest and easiest way to build a visualizer. It makes it possible to create frequency ranges in a variety of shapes, and even allows you to apply simple transforms to those frequencies to further customize their arrangement along the shape.

Piano mode and associated properties are only available in Sound Reactor Pro. See **MidiSetup** for example setup.

INSPECTOR



Spectrum Builder (Script)

Level
- Mode: Object
- Size: X 1 Y 1 Z 1
- Levels
  - Size: 1
  - Element 0: Level
- Share Driver: ✓

Layout
- Shape: Line
- Spacing Mode: Spaced
- Levels: 60
- Spacing: 1
- Between Edges: ✓

Frequency Range
- Mode: Audio
- Preset: Full Range
- Lower (Hz): 20
- Upper (Hz): 20000

Transformation
- Clamp: ✓
- Repeat: 1
- Alternate: ☐
- Reverse: ☐

Build
- Auto Build: ✓

Build

PROPERTIES

Mode

- **Object** creates an array of GameObjects from the levels specified in the Levels property.
- **Vector** creates it's own GameObjects that become points in a curve.

- **Piano** creates a piano from the specified piano keys. (only available in Pro)

Size

The **Level** is scaled by this size.

Levels

The GameObjects containing **PropertyDrivers**. The GameObjects will be created and repeated in the order that they exist in the list. When Mode is set to Object.

Share Driver

Tells the builder to attach the **PropertyDrivers** from the **Level** to the instanced Levels. When Mode is set to Object.

Color Driver

The **ColorDriver** to use for coloring the vector. When Mode is set to Vector.

Travel

The distance the vertices should move. When Mode is set to Vector.

Anchored

Anchors the bottom or inside edge of the vector. When Mode is set to Vector.

Anchored Diam.

Diameter of the inside edge of the vector. When Mode is set to Vector and Anchored is checked.

Shape

The shape to arrange the Levels into.

- Line
- Circle
- Rectangle
- Segmented Levels

Spacing Mode

How the **Levels** should be spaced.

- **Spaced** evenly space levels based on their Level Size.
- **Divided** evenly divide the Layout Size into levels.

By Edge

Add space between level edges. Otherwise add space between level centers. Appears when *Spacing Mode* is set to **Spaced**.

Fit Inside

Divide the layout so that the level edges are tangent to the *Layout Size*. Otherwise just divide the layout as if the *Level Size* were zero. Appears when *Spacing Mode* is set to **Divided**.

Levels

The number of levels to build up the spectrum with.

Layout Size

The size of the layout. For *Lines* it's the length, *Circles* the diameter, and for *Rectangle/Segmented Levels* the number of rows and columns

Mode

This can be set to either *Audio* or *MIDI*. *Audio* frequency uses a sub range of ~20hz to ~20000kz, and *MIDI* uses a sub range of 8.18hz to 12543.85hz.

Preset

Frequency range presets.

Lower (Hz)

Lower frequency of the range.

Upper (Hz)

Upper frequency of the range.

Clamp

Keeps the last level from becoming the first level.

Repeat

The number of times to repeat the frequency range along the shape. Only available when *Clamp* is unchecked.

Alternate

Tells the frequency to reverse every time it repeats.

Reverse

Causes the levels to be assigned frequencies starting with the highest frequency first.

Flip Level

Causes the levels to display upside down. Only works with *Segmented Levels*.

Auto Build

Toggles auto build.

Audio Peaks

SUMMARY

Audio peaks are an important part of visualizing spectrums, and even more important when you want things to react to the magnitude of a frequency. This topic exists to help developers understand what peaks are and how they are used to make spectrum values useful.

Peaks

A **peak** is considered the highest possible magnitude of a frequency for a given sound. If we divide a magnitude by it's peak, the result is a value that ranges from 0 all the way up to 1. This technique is called normalization, and it is very useful when visualizing sounds, or writing events that react to sounds.

Raw Spectrum

Unity returns spectrum data at a very small scale, around 0.01, and those values are not very useful to us. Moreover, the scale changes slightly depending on the sample rate, FFT window, and magnitude type, i.e. dB vs. Linear. What we need to do then is save the peaks of a sound so they can be used later to normalize the sound, and Sound Reactor let's you do just that by allowing you to save peaks to a file called a **PeaksProfile**.

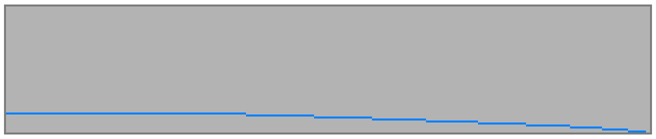
Peaks Profile

A **PeaksProfile** stores: sample rate, FFT window, magnitude type, peaks for 30 bands, and one peak for the entire spectrum. There are two types of peaks profiles that can be recorded:

- **Unique Peaks** peaks used to normalize the magnitudes of one sound. Recording a peaks profile like this guarantees that the highest magnitude of that sound will always reach a value of 1, but it means recording the peaks of every sound you intend to play.
- **Generic Peaks** peaks used to normalize any sound. This method doesn't guarantee that the highest magnitude will reach a value of 1, but it does means that you only need a minimum of one profile.

Generic Peaks

In order to record generic peaks you need a special sound file called a *sweep tone*. A *sweep tone* is a sound that plays a tone from 20Hz all the way up to 20,000Hz at a constant dB, which is essentially creating a ceiling of peaks. The result is a curve that looks like the following:



Sound Reactor already comes with a few profiles that can be used, just look for: **Decibel2048.peaks**. However, if you need to record your own, you can use the provided sound file, or generate one from a website that generates sweep tones, then follow this **recording guide**.

Bands

Last but not least. Sound Reactor doesn't record the peaks of every sample in the spectrum, it only records the number of bands specified in **SpectrumSource**. Furthermore, when the peaks are saved to a **PeaksProfile**, only 30 bands are saved. These peaks are automatically interpolated when saving, and when loading.

MIDI

SUMMARY

MIDI stands for **Musical Instrument Digital Interface**. It's a standard communication protocol designed for digital instruments such as: pianos, digital drums, and other various electronic musical equipment. The standard has been around since 1982, and has since been used by thousands of devices, even ones that don't play sound.

You can read up on the MIDI standard at **MIDI Association**.

This feature is only available in Sound Reactor Pro.

Sounds

There is actually no such thing as a MIDI sound. MIDI is merely a messaging protocol, and it's job is to tell MIDI software and devices what to do. The most notable one, of course, is to play sound, which is what it was originally designed to do. Today, however, MIDI has been adapted to control many things, such as: lighting, switches, movements, and so on.

Events

MIDI is an event driven messaging system. There are many types of events, ranging from notes turning on and off, to communicating meta messages that include tempo, cues, words/syllables, and even copyright notices. The device listening to the events can do whatever it wants with them. It can play sounds, or do something else with them. For example, Sound Reactor Pro uses the values to modify properties such as: rotation, scale, position, or any other property you tell it to.

Frequency

MIDI supports 128 notes, and since it was originally designed to be used with sound, they chose to assign frequency values to those notes. The range they chose was: ~8hz to 12.5Khz..

INTEGRATION INTO SOUND REACTOR

Sound Reactor Pro handles MIDI events and messages in two ways:

- The reactor part of Sound Reactor Pro uses the note on/off events and their velocities to affect level magnitudes. In other words, it works exactly the same as it's audio counterpart, except all the notes are individualized instead of being a blended mess of sound. This means you can drive properties to individual notes without bleed over from other notes.
- There is a new `MidiSource` class included with Sound Reactor Pro that lets you handle MIDI events and messages directly. For example, if you want to display lyrics, which is something that can be stored in a MIDI file, then they can be handled and displayed to the screen using a custom **MidiEventHandler**.

MIDI Support

Sound Reactor Pro can play back pre-processed MIDI files. It supports a majority of MIDI events and messages, but it does **NOT** support playing soundfonts/synths or communicate with MIDI devices. See the following for what is, and is not supported.

Supported

- MIDI type 0 and 1 files.
- MIDI file playback
- Seeking to specific times
- Speed multiplier

Unsupported

- Will not connect to MIDI devices
- Will not save MIDI files
- Will not generate MIDI events
- Will not play soundfonts or synths.

Unsupported Events

- System Exclusive

Unsupported Meta Messages

- Device Name
- Channel Prefix
- MIDI Port
- SMPTE Offset
- Sequencer Specific

MIDI Files

Sound Reactor Pro doesn't open MIDI files directly, instead, they need to be converted to a .asset file (Script-ableObject). Sound Reactor Pro adds extra info that MIDI doesn't include, like how long a note is held down for, how many notes exist, and so on. See **MidiFileSetup** for how to create a .asset file.

NOTEWORTHY

Black MIDI

Since Black MIDI is a thing, and it was used extensively to test the MIDI integration into Sound Reactor Pro, I feel it should be covered here. Black MIDIs are MIDI files that contain more notes than you can shake a stick at. In other words, someone thought it was a good idea to see how many notes their computer could handle before they could bring it down to its knees. Because of this, I relied heavily on these unique files to test every part of Sound Reactor Pro. So a big thanks to all of you in the Black MIDI community!!!