GTMY.Audio Framework

Updated: December 22, 2020 by Roger Crawfis

This document provides an overview of the music and sound effects (Sfx) software developed by Games That Move You, Pbc (GTMY). All software is copyrighted by GTMY.

# Overview

The framework consists of several interfaces and the several implementations of these interfaces. This may seem overly complicated but allows for great flexibility. The audio sources can be from a pool, use mixers, follow a moving object, or handled by Unity efficiently. Once set up, usage is quite easy. For instance, to play a collision sound effect, just call:

**AudioManagerSingleton**.*PlaySfx*(“collision”);

AudioManagerSingleton helps manage several audio players and a music player. It contains the “master volume” and adjusts the volume levels as this property is changed. It also has Mute and UnMute functions. If Mute() is called, all audio players and any music player that are registered with the AudioManagerSingleton will have their volumes adjusted.

# Music Player

The IMusicPlayer is an interface that supports Play, Stop, Pause, UnPause, and Shuffle behaviors as well as the ability to get and set the properties: Volume, and FadeTime. This design chose to keep a master volume and a player volume separate. A music player keeps track of the master volume. This is a push design choice, rather than a pull where all classes would know about the AudioManagerSingleton. When the master volume is changed, AudioManagerSingleton sets the master volume property for any players (including music) that are registered.

The current music player can be fetched from the AudioManagerSingleton or played directly like so:

AudioManagerSingleton.Instance.Music.Play();

Concrete implementations of the IMusicPlayer include a simple MonoBehaviour that takes a list of clips and plays them, called MusicPlayerExplicit and one that uses Addressables, called MusicPlayerAddressables. The latter will find all songs with a label of “music” as well as any additional user supplied Addressables labels. It takes a list (possibly empty for all music) of strings.

# Sound Effects (Sfx)

For sound effects and ambiance, there are four key interfaces as described below:

* ISfxAudioPlayer – A sound effects player that managers the audio source construction and clip selection. There are 3 properties and 3 behaviors associated with this interface.
  + SfxType – A string that identifies the “theme” of this player, for instance “explosion” or “ambiance”. Different players can be used for different themes.
  + GlobalVolume – Set and controlled by AudioManagerSingleton.
  + LocalVolume –This specific type or “theme”’s volume is multiplied by the GlobalVolume.
  + Play(volume) – Play a sound effect, scaling the volume down even further.
  + Stop() – Stop the clip. Note, since some Sfx’s are assumed to be short, an implementation may ignore any calls.
* IAudio – A wrapper for an audio channel and associated properties. This is used to ultimately play the sound / music. Concrete classes for Unity3D usually will encapsulate at least one UnityEngine.AudioSource. There are three behaviors:
  + Play(clip, volume) – plays the clip with the specified volume.
  + Stop() – Stops a clip that is playing.
  + SetAudioPosition(Vector3 position) – set the location of the audio source.
* IAudioClipProvider – Provides a UnityEngineAudio clip when asked for. Allows clips to be played via themes, rather than specific files, although a concrete implementation allows for a single specific clip object. Supports two behaviors:
  + GetNextClip() -> UnityEngine.AudioClip – provides a clip.
  + Shuffle() – Changes the order clips are provided.
* IAudioFactory – Provides instances of IAudio when needed. There are two behaviors:
  + CreateAudioSource -> IAudio – Returns an instance of IAudio.
  + ReleaseAudioSource(IAudio) – Lets the class know that this instance of IAudio is no longer needed.

## Example Usage

An example of the use of these interfaces is the following. Our AudioManagerSingleton can take a theme, find an instance of ISfxAudioPlayer that is associated with this theme and then asks that player to play a sound effect. The instance of ISfxAudioPlayer may ask an instance of a IClipProvider, that is contained within the instance, to provide a clip to play. The ISfxAudioPlayer instance may either use a contained IAudio to play the clip or use an IAudioFactory to get an audio source that can then play the clip.

Usage is simple, for instance to play a sound on a collision, just call:

**AudioManagerSingleton**.*PlaySfx*(“collision”, 0.8f);

This method assumes there is an ISfxAudioPlayer associated with the keyword “collision”. The constant is an optional scale on the volume (assumed between 0 and 1). It is implemented within AudioManagerSingleton using a Dictionary as follows (equivalent to sfxPlayers[soundType].Play(volumescale) ):

public void PlaySfx(string soundType, float volumeScale = 1)

{

    if (sfxPlayers.TryGetValue(soundType, out ISfxAudioPlayer sfxPlayer))

    {

        sfxPlayer.Play(volumeScale);

    }

}

## ISfxAudioPlayer Implementations

Currently, there are two implementations of ISfxAudioPlayer. The class SfxAudioPlayer which takes an instance of a IAudioFactory and a derived class, SfxAudioPlayerUsingAudioFactoryRegistry, which takes a registered audio factory name. The latter uses the AudioFactoryRegistry to look-up the audio source factory. The AudioFactory Registry should be set-up prior to creating an instance of SfxAudioPlayerUsingAudioFactoryRegistry

The concrete SfxPlayer class implements the *Play* method as follows:

public void Play(float localVolumeScale)

{

    UnityEngine.AudioClip clip = clipProvider.GetNextClip();

    float volumeScale = localVolumeScale \* LocalVolume \* GlobalVolume;

    var factory = AudioFactoryRegistry.Instance.GetAudioFactory(audioType);

    currentAudio = factory.CreateAudioSource();

    currentAudio.Play(clip, volumeScale);

}

This class holds (contains) an instance of an IClipProvider and uses it to acquire an audio clip. Once acquired, the overall volume is calculated, an audio source is acquired or created using a factory and then the clip is played on this audio source at the desired volume.

You can register a ISfxAudioPlayer with the AudioManagerSingleton. Once registered, you can also get or access the audio player using the string keyword.

### Creating an ISfxAudioPlayer using the SfxAudioPlayerFactory

Instances of ISfxAudioPlayer’s can easily be created using the SfxAudioPlayerFactory instance. This will create the audio player as well as register it with the AudioManagerSingleton for future use.

## IAudio and IAudioFactory Implementations

In general, the programmer or user will not deal with IAudio’s directly, but rather use an IAudioFactory. Creating new factories may require new implementations of IAudio. There are several IAudio implementations available. These include:

* AudioSource2DOneShotSingleton – uses Unity’s OneShotMethod on a reusable AudioSource to play a sound effect. An immutable AudioSource is created and SetAudioPosition is not supported. The position is always the origin. The Stop method is also a no-op as clips are assumed to be short.
  + Created by AudioFactoryOneShot2D using the “OneShot2D” type. This is a pre-registered factory.
* AudioSource3DOneShot – uses Unity’s OneShotMethod on a reusable AudioSource to play a sound effect. An AudioSource is provided in the constructor. The Stop method is a no-op as clips are assumed to be short and Unity’s OneShot does not support stopping them.
  + No default factory is associated with this type yet.
* AudioSource3DBuiltIn – uses Unity’s PlayClipAtPoint static method. The AudioSource is not exposed and hence cannot be changed. The Stop method is a no-op as clips are assumed to be short and Unity’s PlayClipAtPoint does not support a stop method.
  + Created by AudioFactory3DBuiltIn using the “3D” type. This is a pre-registered factory.
* AudioSourceGameObjectAdaptor – uses the AudioSource attached to a Unity GameObject. It will create an AudioSource if one is not found.
  + Created by an AudioFactoryPrefab once registered.
* AudioSourceComposite – contains a selection of audio sources (IAudio’s) that are picked at random each time a clip is requested. Allows for a variety of pitch / volume, etc. Not sure how this will work with clean-up and pooling.

More factories are needed to handle object pooling, …

## IAudioClipProvider’s

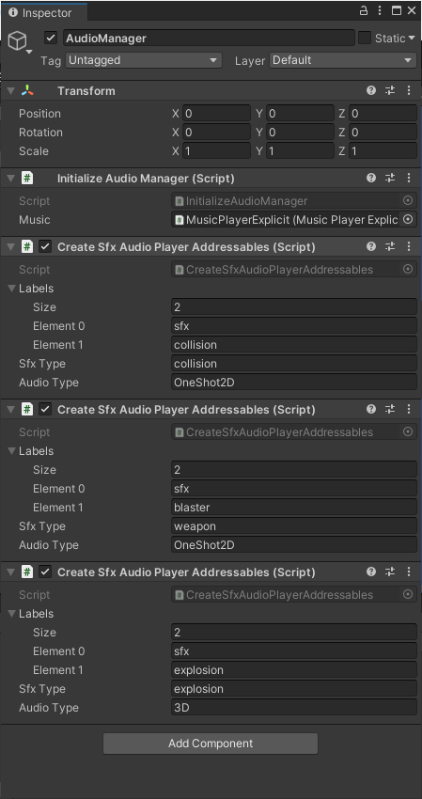
Audio Clip Providers allow for

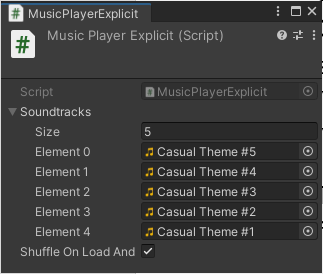
1. Some variety of sound effects.
2. Separation of concerns – what to play versus how to play it.
3. Avoidance of hard-wired file locations

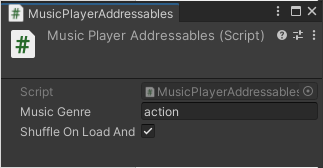
There are currently 3 different IAudioClipProvider’s:

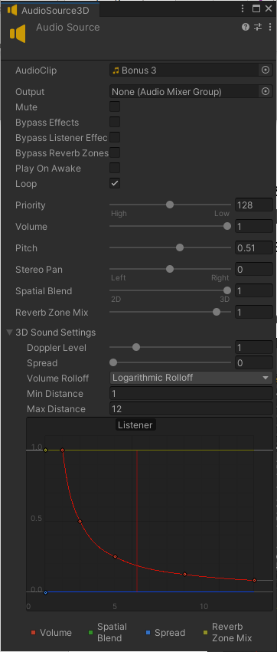
* AudioClipProviderSingleInstance – always returns the same clip. Think of this as replacing a hard-wired clip in a SfxAudioPlayer.
* AudioClipProvider – contains a list of AudioClip’s and randomly selects one.
* AudioClipProviderAddressablesPreLoaded – derived from AudioClipProvider. It uses Addressables and string keywords to load the targeted sound effect(s) by calling LoadAllClipsAsync.

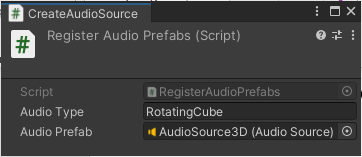
## MonoBehaviours and Testing

There are a few Unity scripts to help test and illustrate the framework. Most of these are for testing only. Here is an example empty GameObject that initializes some sound effect players using Addressables. The image shows two scripts: InitializeAudioManager and CreateSfxAudioPlayerAddressables. In this case, three instances of CreateSfxAudioPlayerAddressables are created. Two of these, collisions and weapons, are played on the OneShot2D audio source. Explosions are played on the “3D” audio source. These use labels in the Addressable system. The “weapon” sound effects finds all assets that are labeled as both a “sfx” and a “blaster”.

The InitializeAudioManager script sets the IMusicPlayer for the AudioManagerSingleton class instance. An example of a MusicPlayerExplicit is below. It takes a list of AudioClip’s and has a Boolean to indicate whether to shuffle them or not. In this case, it will shuffle between 5 different songs

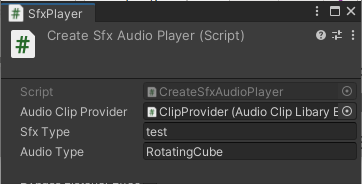
The MusicPlayerAddressables script is even easier. It simply takes a list of labels. The label “music” is always added to these. In this case, we are asking for all music labelled as action. That is all Addressables with both the label “music” and the label “action”.

For a final example, we create an Audio source. This audio source will move around the player in 3D space. This audio source will serve as a template for newly created audio sources when associated with an audio type. The GameObject instance that this AudioSource is attached to will act as a prefab and be Instantiated. These new prefabs will be created as siblings of this gameobject, hence moving as its parent does. The AudioSource used as a prefab should thus be an Empty GameObject with only an AudioSource component.

The image on the right is a simple test (or utility) script that defines an audio type and sets the Prefab to create any time this audio type is requested. It does this by creating a factory, in this case an AudioFactoryPrefab, and registering it with the AudioFactoryRegistry:

var factory = new AudioFactoryPrefab(audioPrefab);

AudioFactoryRegistry.Instance.RegisterAudioFactory(audioType, factory);

The chart in the audio source shows the distance of the AudioListener from the AudioSource. Another test script can now use this audio source factory name as shown below with the script CreateSfxAudioPlayer. This is an example of using another test script to provide an IClipProvider when creating SfxAudioPlayer’s. Here, anytime a “test” sound effect is asked to play, the RotatingCube audio factory will create a new AudioSource and play the clip. In this case, a AudioClipLibraryExplicit script is used to take in a list of AudioClips and create a new AudioClipProvider that contains these clips.

Note: When the audio source is no longer in use, then the audio source will be released back to the factory. In this case, the prefab will be deleted. Other implementations can implement an object pooling framework. An audio source is not in use if either:

1. Stop is called
2. The end of the clip of reached and the audio source (prefab) is not set to loop. Hence, looping audio sources should be in their own factory.

## Todo:

1. ~~Have any prefab’s created return them to the factory that created them.~~
2. Add Next() to MusicPlayer
3. Add Pooling once Unity 2021.1 is out with its pooling infrastructure.