THE WANDERSONG AUDIO ENGINE

You can also view this document online in a Google doc, in case I make changes or improvements to it later.

DISCLAIMERS

1. This is real functioning code that was shipped with a real finished game. As such, **it is horrible**. Like, just horrible. Things are hacky, undocumented, badly named, badly formatted, every sin under the sun that you're told you should never do in Programmer School™. I'm sorry.
2. **This was made for Wandersong**. I did my very best to make this project-agnostic, so you could implement it with any kind of game, but it might be missing or lacking in specific features that you want for your project, or it might not be perfectly optimized for the way you want to use it.
3. **I will not be doing regular support for this.** That's why I won't make anyone pay for it. I have my own games to make, and I can't be responsible for whatever issues arise for anyone who decides to use it. I WILL be using this in my own future projects, and I might even end up making small changes or improvements to it as I go then, but by that point it will be inextricably tied to the structure of that game, so those changes won't make it back to this version you're using now. I welcome you or anyone to host this on github and manage the repo and make further improvements to it.
4. **Using this will alter your workflow for audio.** It won't be easy to merge into an already-far-underway project. The editor makes assumptions about your project structure. You'll need to organize your audio folders a specific way to make use of it.
5. I recommend **version control** if you're using this. The audio editor isn't 100% user friendly and it can be easy to screw stuff up. This wasn't really meant to be shared with other random users; it was made for us to be used by us. We've had lots of time to find and fix issues, but you just never know… and I don't want to be responsible for you losing a bunch of work. Use it with caution!
6. **Good audio comes from good audio designers.** This provides you the means and tools to get creative with your audio in GameMaker, but in the long run it's up to how you use it!

OK! Let's begin.

PART 1. OVERVIEW

1A. WHAT IS THIS

THIS is a powerful layer of logic and interface that works with **GameMaker Studio 2** to give you greater creative control of the audio in your games. If you are familiar with audio engines like **FMod or Wwise**, this is kind of like that, but built directly into your project codebase. It's made out of Game Maker scripts, objects and rooms.

There are two main parts to this thing.

First, the **Audio Editor.** This is a **visual editor for audio logic**. It exists as a "room" in your Game Maker project. When you run your game and visit that "room," it will let you browse the audio in your Game Maker project and create logic for them. For example, you could have sounds play at a randomized pitch, or play a random sound from a list, or smoothly blend between two layers of a song… stuff that sounds do in games. It also gives you a wealth of tools for **mixing**: attaching sounds to different "busses," and controlling the relative volume of sounds to create a balanced audioscape. This was made in conjunction with Em Halberstadt, who did the audio design in Wandersong, and is basically designed for a sound designer to implement their audio as they would in an engine like Fmod or Wwise.

Then, there's the **Audio "Engine"** itself. At its base level, this **does not add any new functions to Game Maker's audio engine**. It simply uses the existing audio functions in nuanced ways, and adds a layer of logic and tracking to let you do complex behaviors with little effort. You can use our extended "container" functions to play audio that uses the fancy logic you defined in the audio editor, and it will update in real-time based on changing parameters, etc. It also handles 3D spatial positioning for audio, tracks BPM and timing events, and has functions to load in groups of audio to help you manage your memory in larger projects. It's fairly well optimized, manages its own memory, and comes with robust debug tools so you can see what's happening in your sound.

This code was written and iterated on for about 2.5 years. It has a lot of features!!!

The "Editor" portion was only ever tested on PC, but once you've defined logic for your game, it will ship and run on all sorts of computers and consoles.

1B. HOW TO IMPLEMENT THIS INTO YOUR PROJECT

Included in the download is a file called **Wandersong_Audio_Engine_importme.yymp**. Drag and drop that into your project to import all the assets.

Put **objAudio** into the first room of your game. It's a persistent object that will live forever in the background; it sets up data structures, manages the audio and handles overhead audio logic.

In order to use the audio editor, it needs to know the directory your actual .yyp GameMaker project lives in. This way it can scrape your project and generate data structures based on the audio you've imported.

Here's how to set your project directory without editing objAudio. This is designed so that different machines can work on the same project over version control, so the project directory is saved externally. You'll need to navigate to C:/Users/yourname/AppData/local/yourprojectname (a folder that's auto-generated by GameMaker when you save any files). In that folder make a file called **project_directory.txt,** and in that file should just be a path to the folder containing your .yyp project, including a final slash. Here's what mine looks like, as an example:

C:/Users/banov/Wanderrepo/wandersong/

You can change how that's implemented in the **create event of objAudio**. At the end of the day it just needs a **valid path assigned to global.project_directory**.

You should have a file called **audio_data** with no extension in the "Included Files" section of your project, in the root folder--don't put it in a subfolder or anything. This will hold all of the settings and changes you make inside the audio editor; it lives here in your project so that the settings come with your game when you publish it. The audio editor will write and modify this file directly inside your project, so just make sure it's there and you should be good.

Typically, from the audio editor you'll be able to jump in and out of gameplay by pressing **Control + Enter**. To make sure that works, you'll need to go to **objAudioEditor > Press Enter Key Event** and change the code in there to make sure it goes to the right place to initiate gameplay (like whatever your normal game start room is, or whatever).

Sweet. You're set up to start designing audio now! You can also check out **wandersong_audio_engine_demo_project** to see an example of implementation.

If you intend to make full use of this engine, it's going to impact your workflow in a few important ways...

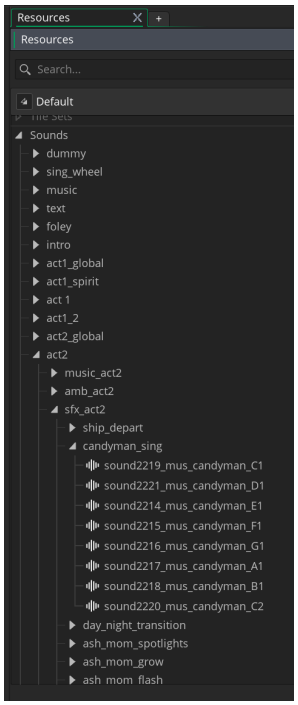
1C. AUDIO CONTAINERS

You'll no longer deal with audio assets directly in your code. Instead, audio is handled through **containers.**

A container is a collection of one or more pieces of audio, with associated logic. An example of logic might be something like, "when you play me, one of 10 different audio files will be played at random." Or "when you play me, I'll play several synced tracks of music simultaneously, and update which tracks are turned on and off based on changes in the game state." Or "when you play me, I'll play a looping sound whose pitch will change based on the game state."

Containers can contain containers. The logic nests too. So you can create really complex audio logic fairly easily. Containers are identified by a name, like a "string." So in your code, you'll write commands like container_play("grass_footstep"), where "grass_footstep" is the name of a container. In Wandersong, from that command the game would play 1 of any number of random grass footstep sounds, positioned in 3D space to the character who played the sound, with randomized pitch and volume.

The audio editor uses the folder structure inside the "Sounds" section of your project to create containers. So it behooves you to organize your files and name your folders well. For the system to work, you can't have multiple containers/folders with the same name. Here's a snapshot of what Wandersong's "Sounds" folder looked like.



So here, "candyman_sing" is a container that would play one of 8 note sounds in the game. But "sfx_act2" is also technically a container, as is "act2."

1D. WORKFLOW

Using this engine, the process for making audio in Wandersong went like this.

1. Em, the sound designer created audio assets using a DAW (her preference was Reaper).
2. She imported audio into the GameMaker project, organizing it into neat folders and naming the folders in a pleasant way with unique names for each one.
3. She ran the game and used the audio editor to define logic for the audio she added. She could define if it was looping, continuous, random, etc. She could create parameters for the audio and hand-craft how the pitch and volume for different sounds would change based on those parameters.
4. She'd send me a list of new containers that she'd created that needed to play in-game.
5. I would add commands to play the new containers in the relevant place in code.

That's it. (Ignoring all the new features and fixes I'd constantly have to add. :P But as we got farther into the project, this happened less and less often.)

We set it up such that if Em wanted to make changes in the audio editor, she'd change the room order in the project so that the game opened to **rmAudioEditor** when you compiled it. She'd make and test changes in there. For implementing her audio, I'd change the room order so rmAudioEditor was tucked away somewhere unreachable, and just test and play the game as if it didn't exist.

PART 2. THE AUDIO EDITOR

I'm not gonna explain how this works, codewise. It's a nightmare and the details are already foggy to me. I'm just going to explain the features of the audio editor as if it were a standalone piece of software, as if I were writing to an audio designer with no technical knowledge. Of course, this isn't a black box… all the code is right there, so you can feel free to dig in and make changes and learn from it.

But like… maybe you shouldn't.

2A. OVERVIEW

We made a video! **https://www.youtube.com/watch?v=AybTLKScpT0**
Watch that!!!!!

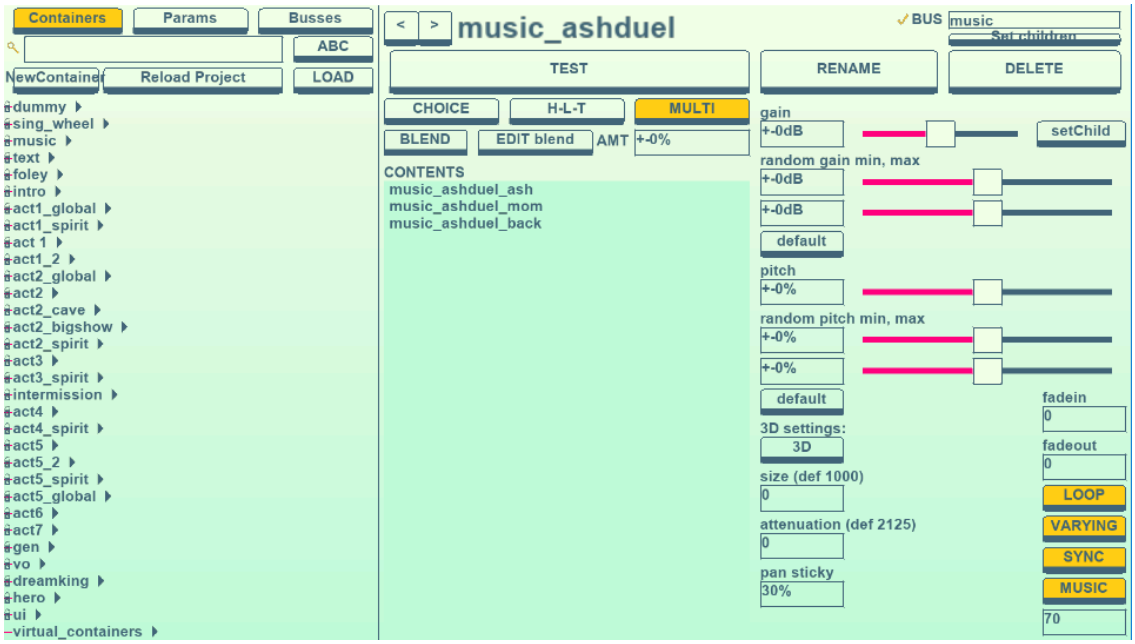
Press **SPACE** to test your sounds. **CTRL+ENTER** to exit the audio editor and jump into your game, and the same command to return from gameplay. **CTRL+S** to save your current settings. **CTRL+Z** to your previously saved settings if you mess something up. **CTRL+X** to load forward, like a "redo." **ALT+C** changes the color palette to a "dark" theme.

Fundamentally, the editor is divided into 3 major tabs, located at the top left: **containers, params and busses**. We'll start with the most complex, containers.

2B. CONTAINERS

2b1. FINDING AND SORTING CONTAINERS

Oh god, a sub-sub-subsection. OK. Here's what the audio editor looks like when I open it up in Wandersong:



On the left is a list of containers, mostly loaded from the GameMaker project. The “lock” symbol next to them indicates that they are pulled from your project, so they can’t be moved, deleted or renamed from inside the audio editor.

There’s a **search bar** that will let you find a specific container or sound by name.

ABC is a toggle to sort the containers alphabetically, which is useful for searching, but by default they display in the same order they appear in your project.

NewContainer is a button to create a new container. I call these “virtual” containers because they don’t exist in your project itself. TBH, they can be a little bit unstable in the editor. I would use them sparingly. But sometimes they’re necessary to combine assets from different containers that don’t otherwise make sense to be nested in your project.

Reload Project Deletes cached project info and reloads the folder structure and names from your GameMaker project. Generally this should happen automatically if it detects any changes, but this is here just in case.

LOAD Lets you load in named audio groups. Audio groups are defined in code, and I’ll talk about them in the engine section. This is to keep memory usage low and tidy.

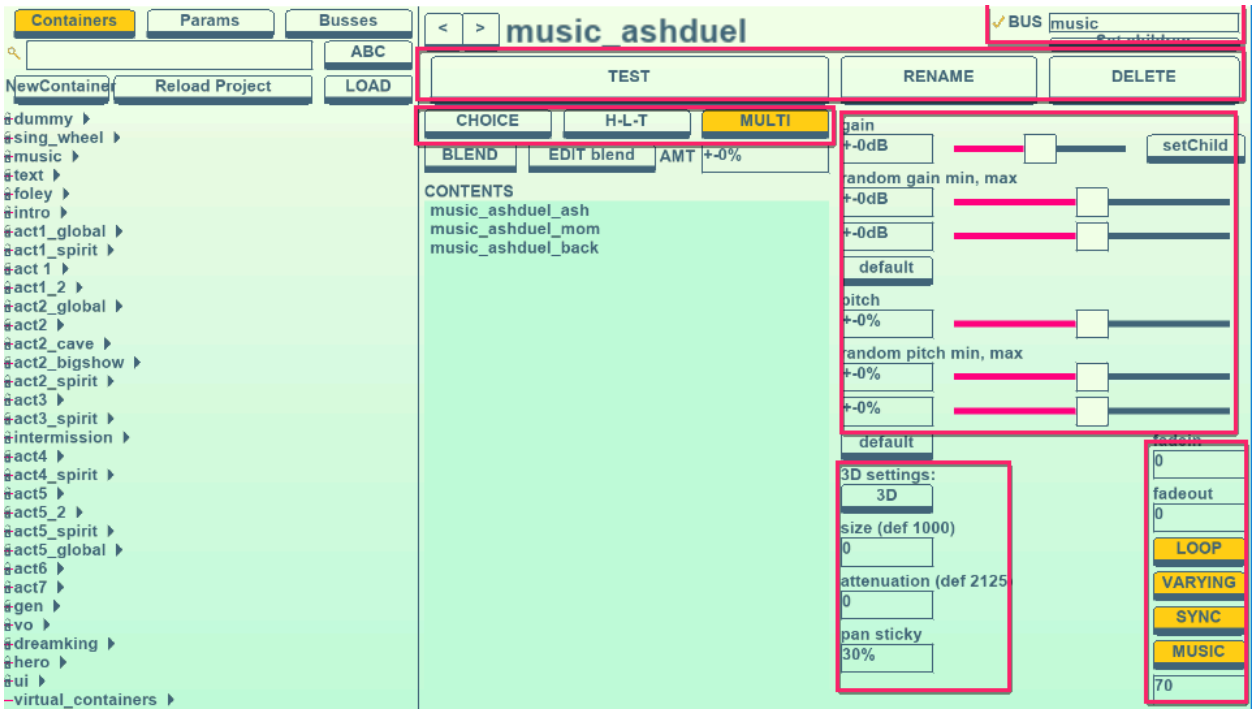
Containers can be dragged and dropped to reorganize them.

DELETE+CLICK can delete an item from the list of container contents on the right.

ALT+CLICK and drag to copy attributes from one container to another. Be careful with this. But it's useful if you need to make a bunch of containers with identical logic.
CONTROL+CLICK to cut and paste a container from one location to another. Usually when you click and drag a container to somewhere it kinda "copies" the reference, so it appears in an additional place, but if you move it with control+click it'll remove it from wherever you moved it from.
DOUBLE-CLICK one to open it.

2b2. CONTAINER ATTRIBUTES

These buttons and attributes are visible for any kind of container.



BUS: Type in the name of a bus to assign the container to that bus. If the name is a valid bus, a checkmark appears. Busses are created in the "bus" tab, and basically they're a way to group sounds up for mixing. For example, you might want to group all your "sfx" into one bus, and all of your "ambiance" into a different bus, and then define broadly that ambiances are turned down a bit and sounds are turned up a bit. There's a hierarchy to it, so "sfx" can have sub-busses like "footsteps" and "voices" and "UI," and if you make changes to the parent "sfx" bus everything on the child busses will update accordingly. This is getting a bit long. Talk to me in the "bus" section.

Set children: This sets the "bus" of every child container and audio asset to match the current one, recursively. So you can make a big parent folder called "ambiances," and put all of your ambiance containers inside that folder, and then in the audio editor go to the "ambiances" folder and set that container to "ambiance" and hit this button and then all of the ambiances will be on the same bus. Woot!

TEST: You can use the SPACEBAR to do this too. Plays the container. This simulates what it would sound like if you wrote **container_play([container name])** in your game code. Press it again while the container is playing to stop the container (**container_stop([container name])**).

RENAME: Works only on unlocked containers. Renames it!

DELETE: Only on unlocked containers. Delete it!!! (I hope this doesn't crash).

CHOICE, HLT and MULTI: This sets the type of your container. I'll cover each in a section below.

Gain slider: Adjust the volume of the sound. GAME MAKER CAN'T MAKE YOUR AUDIO ASSET ANY LOUDER THAN WHAT IT WAS IMPORTED AS, SO IT'S GENERALLY BEST TO IMPORT LOUD ASSETS AND TURN THEM DOWN IN THE EDITOR. But you can turn them up here to offset something else that might be turning it down, like a parent container or a bus.

Gain is calculated with proper nested logic. So if you play a container that is turned down -6 dB, and it contains containers that are turned up +6dB, then the resulting audio will be +0 dB.

setChild: I already forget, but I think this sets the gain of the child objects to be the same as the current one. It's like the "setChildren" bus button, but it is NOT RECURSIVE… it only sets the assets that are listed in the contents of the current container, and doesn't go any deeper. Useful if you want to quickly set the gain of a bunch of sounds at once.

Random gain min, max, and "Set default": This sets a range of values that the gain could be set to randomly. The "set default" button just sets it to a pre-determiend range that we liked using for footsteps. This way, repeatedly playing the same sound can yield different loudnesses.

Pitch slider: Sets the pitch of the sound. This one is fun.

Random pitch min, max and "set default": like the gain ones above but for pitch.

3D Settings/3D Button: This makes the sound "3D," so it has a position in world space when played in the game. THIS ONLY WORKS ON MONO WAV FILES! NO OTHER FORMAT. Sounds will sound like they're to the left or right based on their position, and will also get quieter the farther they are. Whatever object plays this sound, the sound will be "attached" to that object.

Size (def 1000): Sets the 3D size of the sound. If the sound is within this distance of the listener, it will be at its loudest. Leave it at 0 to use the default size, which for Wandersong was 1000. You can set this default size in objAudio Create Event.

Attenuation (def 2125): Sets the 3D attenuation size of the sound. When the distance between the sound and the listener is greater than this number, the sound will be inaudible. Leave it at 0 to use the default size, which for Wandersong was 2125. You can set this default size in objAudio Create Event.

Pan sticky: This feature is NOT IMPLEMENTED, because it's really specific to your camera and game system. But you can see the original gameplay code implementation in **objSpatialObject > End Step Event**. You can make it work based on that. The idea is that the higher this number is, the more "sticky" the sound is with the camera, making it sound closer to the center and panning less. 0% makes it treated as normal. 100% would it make it ALWAYS dead center. In Wandersong, we added this feature at the very end to make sounds pan less in general, because they felt too strongly panned left and right based on their screen position.

Fadein: How long the sound fades in for after starting, measured in SECONDS. So .25 is ¼ of a second. If left at 0 there's no fading at all, but for many sounds, especially looping ones, it can be nice to have the sound quickly fade in when it starts.

Fadeout: Like the above, but how long it spends fading out when you "stop" the sound. Leave it at 0 to have the sound stop instantly as soon as you stop it.

LOOP: Makes the sound loop

VARYING: This is mostly here for MUSIC. It's used for music that changes pitch, or has complicated timing implementation; it makes the time tracking more robust, so that BPM is tracked even as the song changes speed. It uses extra processing to track music when this is turned on, so I left it as optional.

SYNC: Use this for containers of multiple tracks of identical length. This forces the sounds to play in sync, so that they stay in time with each other. Game Maker has a built-in feature called audio sync groups that's meant to do this, but by default I left them turned OFF because they aren't well supported and were buggy on consoles just before we launched (lol). You can turn them on in **objAudio > Create Event** if you want. Otherwise, this just makes it so that when the top container loops, it updates the time of all the other containers to match up, so they don't fall out of sync.

MUSIC: Tag this container as "music" in the engine. At any given time, one container tagged as music has its BPM tracked and is treated as the primary background music. There's functions that will let you know when beats, half-beats, and measures occur… and various other functions to let you know the bpm of the current music. In Wandersong we tied these to all sorts of animations so that stuff moved in time to the music.

BPM: The BPM of this container, if it's music.

2b3. CHOICE CONTAINERS

Choice containers play one item at a time. By default, when you play a choice container, it picks one of its contents at random and plays it. *(((Actually, what it does is it shuffles the order of its contents, and then plays them in order one at a time, shuffling them when it gets to the end. This way it makes sure that no given sound ever plays twice in a row, hiding repetition)))*. If you have just one item inside it, it's basically the same as playing a single sound with **audio_play_sound().**

If the **SEQUENCE** toggle is turned on, it doesn't shuffle the order. This is useful if you want to play one container over and over again, but each time it plays a more intense sound, or plays the next note in a musical sequence. In Wandersong sometimes we would set it up so that your footstep sounds played a sequence, so that it felt like there was a musical pattern in your footsteps.

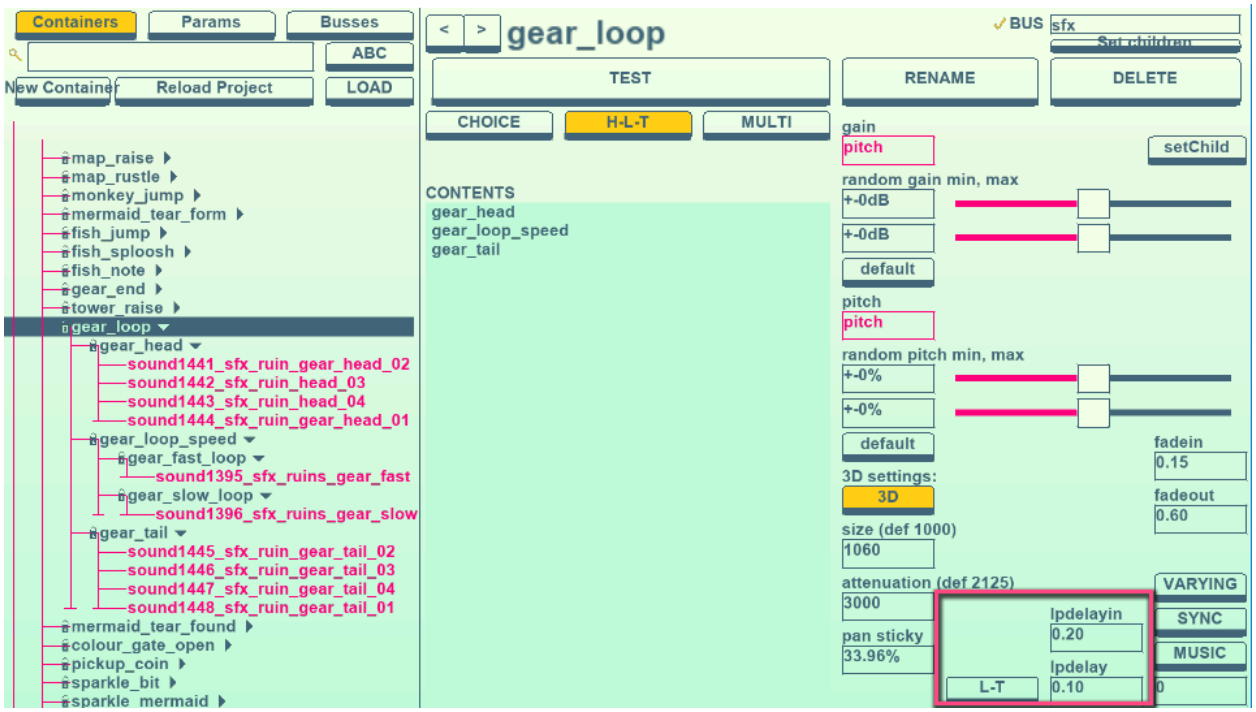
ONGOING makes it so that the container just keeps playing sounds as soon as the previous one ends, until the container is manually stopped.

Choice param lets you use a **parameter** to pick which sound plays. Drag and drop a parameter into this box to attach it. This way you can use the game state to change what sound plays, rather than it being random. Parameters can have a value between 0 and 100… 0 means the first sound plays, 100 means the last sound plays, 50 means the middle one, etc. I'll explain those more later.

Spec min and spec max are generally for “spec” sounds, which are sounds that play occasionally. They set a minimum and maximum time that the sound will repeat within if it’s set to ONGOING. So for example you could have a bunch of bird calls, and use these settings to say a bird call plays every 5 to 10 seconds.

RANDOM XYZ places the sound at a random spatial position each time it plays, so the aforementioned bird calls could sound like they’re coming from all around you. Make sure the sound is tagged 3D! The random distance is based on the default sound size and attenuation you set, so you hear a variety of volumes.

2b4. HLT (HEAD-LOOP-TAIL) CONTAINERS



Head-loop-tail containers are for fancy LOOPING sounds. They can contain 1, 2 or 3 items. If there’s only 1 item, that one is automatically set to be looping. If there’s at least 2 items, the first item is a “head” and the second one is a “loop.” They play at the same time when you play the sound, but the “head” sound is allowed to end while the “loop” sound continues on. We would use this for music, for example to have an intro that played once and then a looping section that followed it. If you have 3 sounds, the last one is a “tail” that plays once when you STOP the sound. We would use this to make an outtro to music that was ending. In the screenshotted container, we had a gear sound that audibly started when the sound started and slowed to a halt when it stopped.

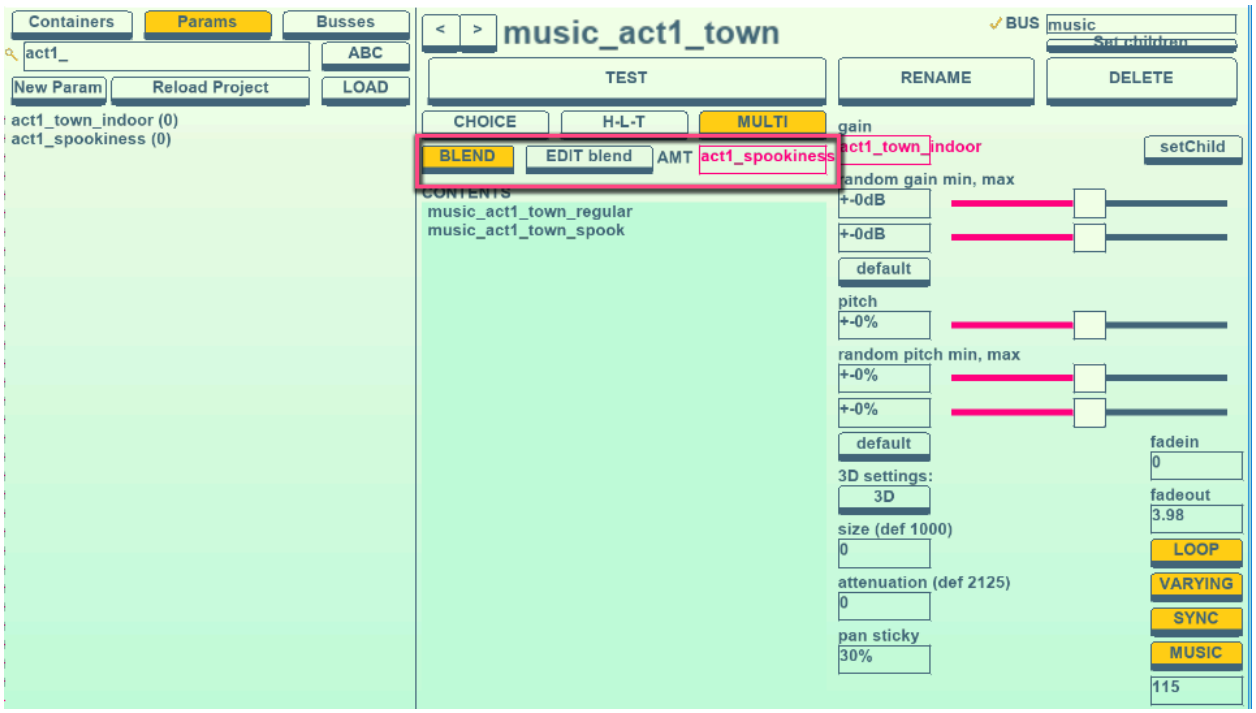
L-T is supposed to make it so that a HLT containing 2 sounds is a loop and a tail instead of a head and a loop, but I think it straight up doesn’t work. Sorry. :(We didn’t use it at all.

LpDelayIn Sets a delay, in SECONDS, for the loop to start after the head.

LpDelay Sets a delay for the loop to stop after the sound stops and the tail begins to play.

Also notably, **fadein and fadeout affect only the loop container**, so altogether this gives you nice tools to create a really precise transition from a head to the loop and from the loop to the tail. We used this feature a LOT to make the singing sound just right in Wandersong.

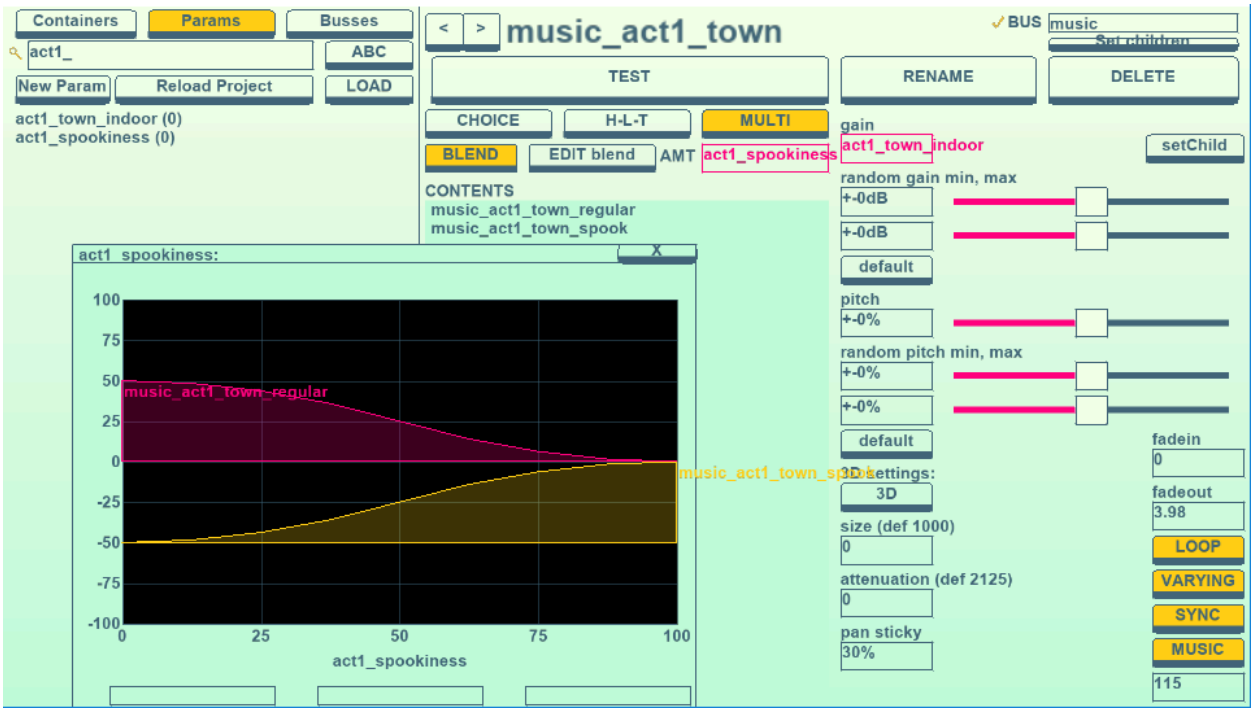
2b5. MULTI CONTAINERS



Multi containers play all of their contents simultaneously. We typically use these for layered sounds like ambiances or songs with instruments that come in and out.

By default, all items play at the same time at the same volume. But by toggling on **BLEND**, you can add logic to change how different layers blend into one another.

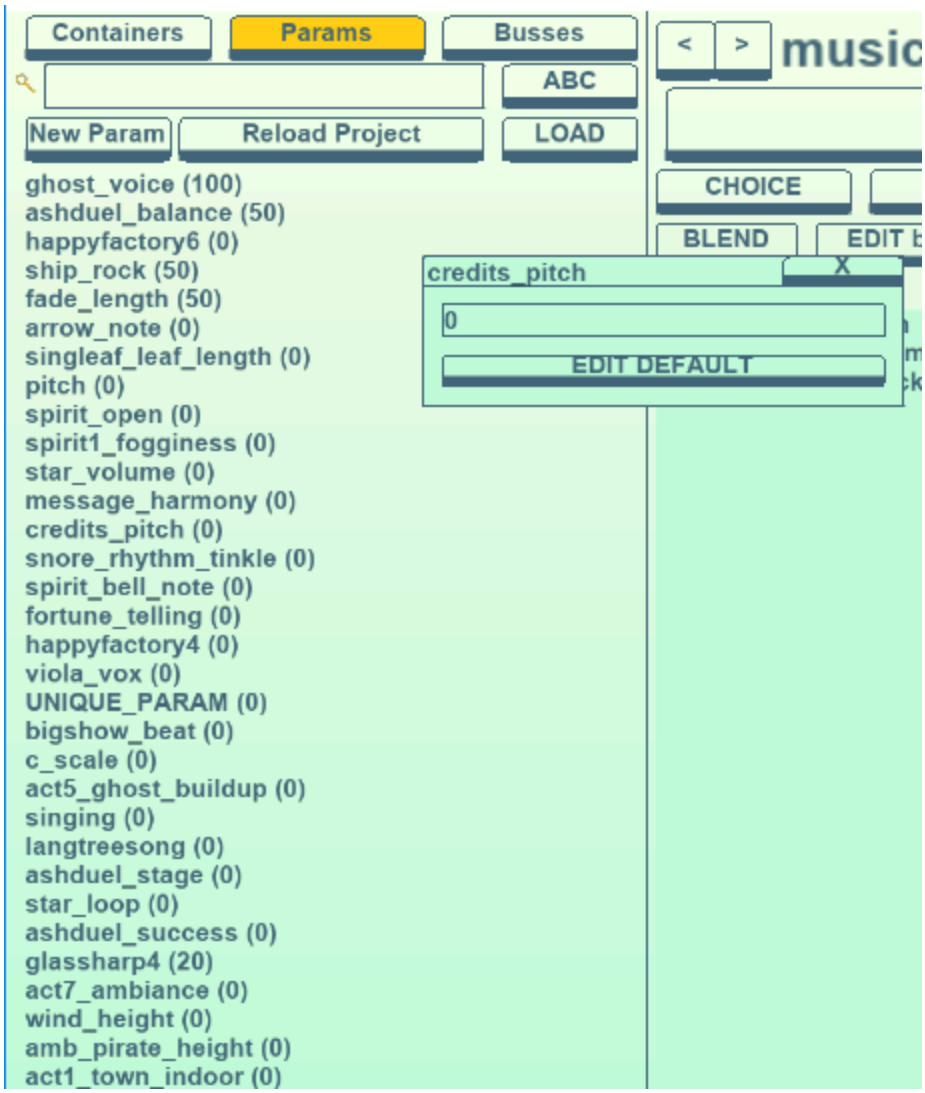
99% of the time, your blend will be driven by a **parameter**, which I'll explain more about soon. Drag and drop your chosen parameter into the **AMT** box to attach it to the blend, then click **EDIT BLEND** to edit how the container changes based on the parameter.



This is the window that will come up when you hit EDIT BLEND. You can click and drag on the colored shapes to change how they relate to the parameter. The taller the shape, the louder the associated item is. So in this example, you can see that when the parameter act1_spookiness is 0, music_act1_town_regular is at full volume and music_act1_town_spooky is completely silent. But as act1_spookiness increases, the town_regular song gets quieter and the town_spook song gets louder. While testing the sound, you can click and drag left and right on the parameter name below the graph to change the parameter and hear how the sound changes in response. If you make changes to the graph, you'll need to stop and re-start the sound to hear the change.

Blends can contain any number of sounds, though the graph can get very crowded when you have more than 6 or so.

2C. PARAMETERS



PARAMETERS are variables. You can attach them to different container attributes, and change their value with gameplay code to make sounds change in real time based on the game state. Parameters can have a value from **0 to 100.** To keep your project from having too many parameters, like we did (see above), it can be good practice to make parameters with more general names like “intensity” and “volume” and re-use them across multiple containers.

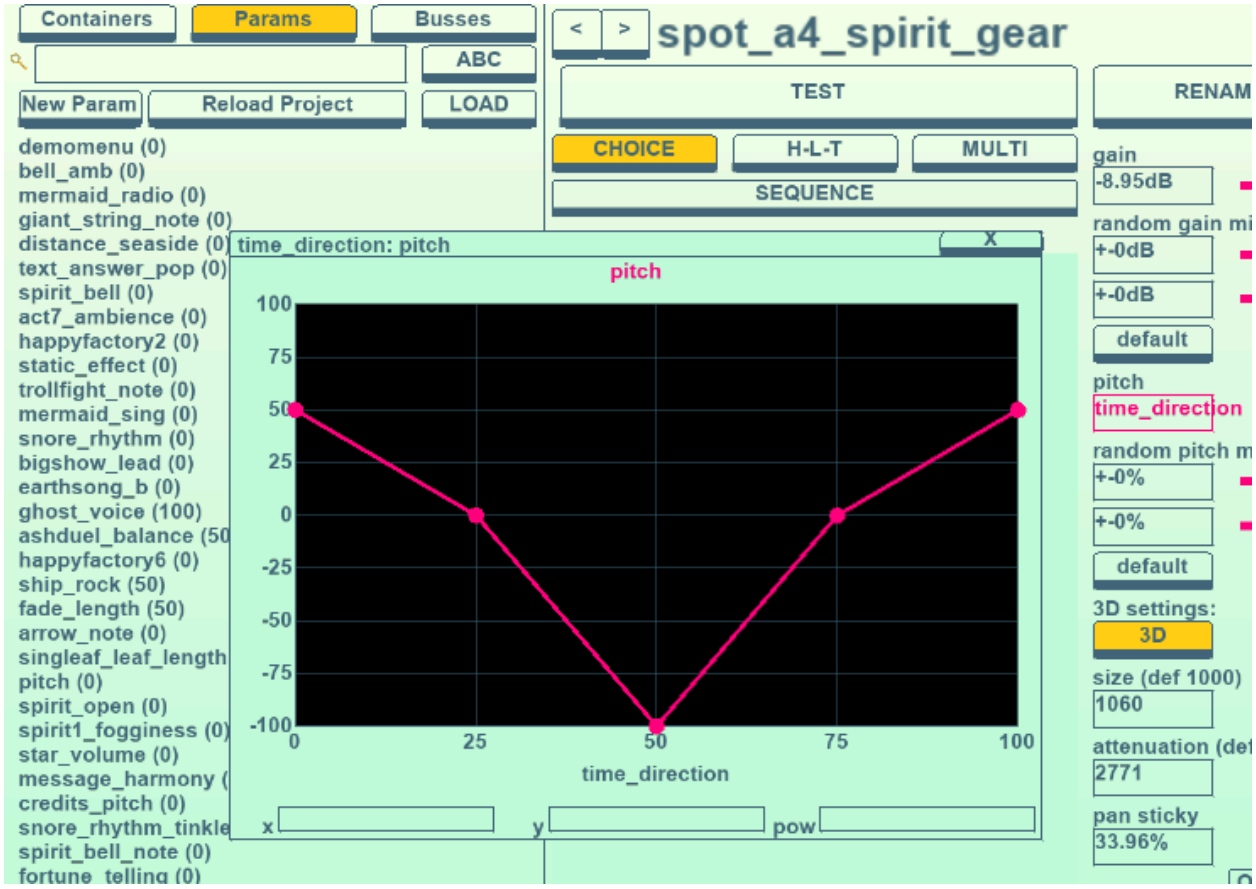
When a parameter updates, all currently playing containers and attributes attached to that parameter will update. The only exception is **UNIQUE_PARAM**. If you make a parameter with that name and attach it to things, you’ll be able to update that attribute only for specific instances of the sound with code. I’ll talk about that later.

New Param lets you name and add a new parameter.

DELETE+CLICK will let you delete a parameter. ...I think. We didn't use that feature much. As you can see.

DOUBLE-CLICK a parameter to open up a small dialog box and edit its default value. All parameters are set to their default value when the game loads. Use this to make sure that the first time you hear a sound or song, it sounds the way you expect.

To attach a parameter to an attribute, click and drag the parameter from the list of parameters onto the attribute you want to attach it to. *Parameters don't actually work with every single attribute.* I implemented them to work with **blends, gain, pitch, fadein and fadeout.**



In the above example, the **time_direction** parameter is attached to the **pitch** attribute. Once a parameter is attached, you can click on the attribute box to open up an editor like above.

You can click and drag to make and move points on the graph. The X-Axis (left to right) represents the parameter; all the left is 0, right is 100, middle is 50, etc. The Y-Axis (top to bottom) represents the attribute output. In the above example, pitch is +50% when the parameter is at the extreme ends of 0 or 100, and -100% when the parameter is 50.

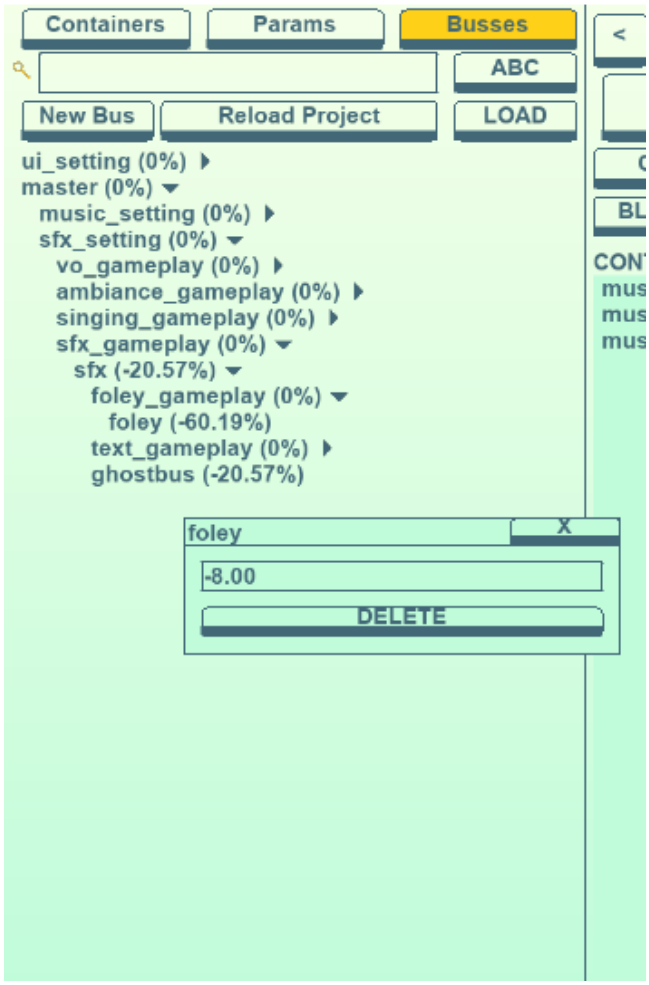
You can click and drag left and right on the parameter name at the bottom to change the parameter, even while the container is playing, to hear what it sounds like. If you change the graph at all, you'll need to stop and re-start the sound to hear the change.

While a point is selected, you can type in values for X and Y in the bottom boxes and hit Enter to set it. The pow box does nothing (sorry).

This graph is also a little unstable so please be careful. (sorry again).

2D. BUSSES

Think of busses like **pipes that your audio flows through**. You use busses to change the volume of sounds that pass through them. They have a hierarchy, with all busses eventually joining at the highest level, the **master** bus, before they reach your speakers. Busses group sounds together and let you control the relative volume of things in a refined way. Below you can see some of the bus hierarchy used in Wandersong.



Changing the volume of a bus changes the volume of all the sounds inside of it and in lower busses. So if we lowered the volume of the sfx bus, we'd also lower the volume of the foley_gameplay bus, the foley bus, the text_gameplay bus, the ghostbus, and all the busses inside those too. But if we lower the volume of the foley_gameplay bus, most others would remain the same, except the foley bus which would also be lowered.

Sorry for the crappy explanation. I'm not an audio designer!!!!!!

Anyway so click **New Bus** to make a new bus. You can drag and drop busses to move them, and **double-click** them to edit them. From the editor box, you can change the gain (volume) of the bus, measured in dB, or **DELETE** them. In the hierarchy view, the volume adjustments are listed as percents, but Em was more comfortable working in dB so that's why we use that in the editor box. Just like with nested container gain settings, bus settings also stack, so if you play a sound from a -6dB/-50% bus that's inside another -6dB/-50% bus, the resulting audio output should be at -12dB/25% gain.

The hierarchy can be messed up pretty easily when you're dragging and dropping because it isn't coded the best, but usually you'll set this stuff up once and rarely ever touch it.

All the volumes you set here are treated as default when the game loads, but then you can update the volume of any bus live with code while the game is playing. For Wandersong, I left most busses untouched so that Em could adjust the volumes of them to make the mix sound just right. I created special busses that would update during gameplay, like sfx_setting, which would update based on what the user set their "sound" volume to in the options menu. I also had busses like music_gameplay to do little effects like make the music softer or fade out completely in certain locations or in scenes. You could use this to make sounds temporarily get quieter after the player is hurt or something. The possibilities are limitless!!!!

OK, that's it for the audio editor.

PART 3. THE AUDIO ENGINE

3A. OVERVIEW

I won't explain how EVERYTHING works. There's just too much. I'm going to focus on just what you need to know to play and manage audio for the purposes of implementation. If you want to know about how the gears inside the box work, you're welcome to poke around.

At any time during gameplay, you can press **Alt + M** to toggle on or off audio debug. This displays information about all of the currently players containers, their 3D position, their current state, their volume, their bus, etc. as well as details about the current BPM and everything else you might need info on. If something doesn't sound how you expect, this might help you figure out why.

3a1. OBJECT OVERVIEW

objSpatialObject is a generic object that your gameplay objects are meant to inherit from. It has a bit of code to attach 3D audio to itself. That way when you play 3D audio, the object that played it also can give information about where on the screen it's playing from, how far it is from the listener, etc.

objLocationSound is a special case object made by container_play_position(). It's just a placeholder object to play a given sound at a given location. Everything cool it does is inherited from objSpatialObject.

objAudio is the controller object that manages overhead audio logic. It sets up structures at the start, manages the currently playing background music and ambiance, updates the global listener position, and manages other general functions like audio loading/unloading and debugging. You can feel free to use its features as much or as little as you like, but they're there for you.

Each instance of **objAudioContainer** controls one audio container. One object can control multiple instances of the same container. They handle all of the live audio logic that each unique sound can do; fading in and out, delaying, adjusting based on parameters, counting time and beats, you name it. There will be a bunch of these while you're playing audio, but they're pretty well optimized. If I did my job well enough, you should never have to look inside this.

The rest are objects specific to the audio editor room. You don't need to worry about those.

3a2. SCRIPT OVERVIEW

I grouped all the scripts into 2 main folders, **audio_engine** and **audio_gameplay**. Audio_gameplay contains all the scripts that you would actually use to implement audio into your game. It's not that many! :) Audio_engine is everything else--helpers, functions, logic and other stuff that helps it work in ways you shouldn't need to worry about. I'm going to overview the contents of audio_editor here though, just so you have a picture of what you're importing into your project.

_engine>generic are misc. helper scripts that are here to do little things here and there. They aren't anything special, most do small one-off functions and are reusable in lots of applications. You could use them too if you want.

_engine>point is my little helper class to make and manipulate 2D points. This was something I ended up using a lot for Wandersong, and I used it for some odd ends in the editor. So they're here.

_engine>draw another little thing for the graph editor

_engine>easing These are easing functions to make smooth curves and animations. They're not terribly complex. I had more of them but I cleared out the ones that weren't used by the engine. You give them a value from 0 to 1 and they give you back a new value from 0 to 1 but moving along a fancier curve. Used to calculate audio fades and other things like that.

_engine>data_structs These are functions I wrote to extend GameMaker's existing data structure library with more things that I often do. I create, manipulate and destroy a lot of data structures and these things just help me save on typing.

_engine>audioeditor All kinds of crap that does audio logic in the background in the editor or during gameplay.

_engine>compatibility Is just here for the auto-generated __view_ scripts that game maker generated when going from studio 1 to 2. I have no idea how views work anymore so I'm still using these. I basically just use them to place and measure things against the screen bounds. In Wandersong the camera only changed once per frame, so I would store the current view x, y width and height to global variables and use those in place of these slow function calls.

I'll talk about each script in **audio_gameplay** individually later so you know how to use 'em.

3B. objAudio

This object is the one you'll need to look at and make changes to so that the audio engine can work with your game. The rest of the objects are auto-generated during gameplay and meant to handle themselves and shouldn't require any input from you.

3b1. SETUP

You'll want to look at **objAudio > Create Event** to set all the special numbers and features for your game. There's a few default numbers for default 3D sound size, camera size, and other things that will be specific to your project. Most of it is documented in the event, so take a browse through!

3b2. TURNING IT ON/OFF

You can can stop objAudio from doing any audio logic by setting **global.audio_gameplay** to **false**. (Or **true** to turn it on). By default, in **objAudio > Room Start**, it sets global.audio_gameplay to true in any room that isn't rmAudioEditor. But you might want to stop it at another time for some other reason. This doesn't stop audio, it just stops objAudio from trying to manage it.

3b3. AUDIO LOADING

GameMaker Studio comes with a feature called "**audio groups**," which let you group your sound files together for loading in and out. For a huge game with lots of audio like Wandersong, we had to make ample use of these to make sure computers and consoles didn't crash. What complicated this for a game like Wandersong was that there wasn't a clear way to group our sounds. For example, sound effects for footsteps on grass needed to be in many places in the game, whereas sounds for specific characters only need to be in sections with those characters. Any given area of the game would have a broad mix of unique and re-used audio from other areas.

So we used the GameMaker audio groups to organize our audio into the most specific categories we could--sounds that would always be loaded in together. Grass footsteps were an audio group. Sand footsteps were an audio group. Miriam's voice was an audio group. Sounds of windy puzzles were an audio group. Etc. By the time the game came out we had nearly 100 audio groups… and to help manage those, objAudio has a system for creating "**grouplists**," collections of those groups for loading multiples of them in together.

First, you'll still want to create groups for your audio like you would in any GameMaker project. Use the **Tools > Audio Groups** dropdown to create audio groups and organize your files.

Then in **objAudio > Create Event**, you'll define grouplists so that you can load in exactly what's needed for each area and nothing extra.

You can see some examples roughly on line 25, commented out. You'll want to define your grouplists in a similar fashion, with a name and a list of sub-groups. Those names will be how you load them in and out with functions later. At any given time, only one named grouplist can be loaded in at a time. I'll talk more about those functions in the Scripts section.

I also left some example code commented out in **objAudio > Draw GUI**. Feel free to uncomment that if you want. It displays stuff while audio is loading!

3C. SCRIPTS

OK. Here is a detailed overview of every script in the **audio_gameplay** folder, what they're for and how to use them. This is everything you need to know to implement audio defined in the audio editor.

3c1. CONTAINER FUNCTIONS

These are your bread and butter functions for implementing audio. Each one takes just one argument--the string name of a container which you've defined in the audio editor. For example: container_play("glass_break") to play the "glass_break" container.

container_play(string nameOfContainer) plays the named container. It should sound just like when you hit "Test" in the audio editor. This returns the **objAudioContainer** object handling the sounds. **To get the specific id of the sounds that were just played, you need to get the value that's in that object's play_id variable.** So it might look like this:

```
var break_sound = container_play("glass_break");
break_sound_id = break_sound.play_id;
```

With this code, you'll be able to use break_sound_id later on to stop or manipulate the specific instance of the sound you just played.

If the container is a 3D container, it will be attached to the object that played it and sound as if it's coming from that object. This is important to keep in mind if you're doing spatial audio stuff. You can attach containers to other objects by using the with() format, ie. with(objPlayer){container_play("glass_break");} will make the "glass_break" container sound like it's coming from the player, regardless of who executes this code.

container_play_position(string nameOfContainer, x, y) is identical to above but you can also provide an x and y position for the sound to play at, and if it's a 3D container then it will sound like it's coming from that spot, rather than being attached to anything. Neato!

container_stop(string nameOfContainer) stops the named container, if it is playing. You can optionally provide a sound id as a second argument to stop only that instance of the sound. So for example, you could say:

```
for (var i=0;i<10;i+=1){
        var newsound = container_play("glass_break");
        break_sound_id[i] = newsound.play_id;
}
container_stop("glass_break", break_sound_id[5]);
```

The above code would play the "glass_break" container 10 times at once, and then stop the 6th one instantly afterward. Why would you do this? I don't know, but you can.

container_is_playing(string nameOfContainer) returns 1 if the named container is playing, or 0 if it isn't. Even if the container has an extra decay or fadeout after it's stopped, this function will start to return 0 as soon as you tell it to stop.

container_reset(string nameOfContainer) is a fairly special case one and removes some cached data from the named container, so that next time you play it, it's as if it was the first time. This is basically in place for **Sequence** sounds, so that you can start the sequence over at the beginning if you need to.

3c2. PARAMETER FUNCTIONS

These are for checking and changing the value of parameters that you've defined in the audio editor. Containers that are attached to those parameters will update in real time if they detect any changes! Remember to provide the parameter name as a string, ie. to set a parameter called "intensity" to 100 you'd say audio_param_set("intensity",100);

audio_param_state(string nameOfParameter) tells you the current value of the given parameter.

audio_param_set(string nameOfParameter, real newValue) sets the given parameter to the new value. Remember, our audio editor graphs are designed assuming that parameters will be set between 0 and 100, so keep that in mind.

audio_param_unique_set(string nameOfContainer, real soundId, real newValue) Typically, when you change a parameter, ALL of the containers and attributes attached to that parameter will update together. But in the audio editor section, I mentioned the special parameter **UNIQUE_PARAM** you could attach to containers. This is what that's for. Containers with a UNIQUE_PARAM attribute can be adjusted individually with this function. For example, imagine if the container "glass_break" had a UNIQUE_PARAM attached to its pitch attribute.

```
for (var i=0;i<10;i+=1){
    var newsound = container_play("glass_break");
    audio_param_unique_set("glass_break",
        newsound.play_id,
        (i/9)*100);
}
```

This would play "glass_break" 10 times at once, but each of the 10 sounds would be set to a different pitch.

You can't attach UNIQUE_PARAM to multiple attributes on the same container and update them individually. That case never came up for us so we didn't implement it. :P

audio_param_unique_get(string nameOfContainer, real soundId) This will return the UNIQUE_PARAM value associated with the given container. It will return 0 if the container has no unique param or it hasn't been set. It will return the keyword **noone**, or **-4**, if the container doesn't exist.

3c3. BUS FUNCTIONS

Just like containers and parameters, you identify busses by string name.

Busses are typically set from -100 to 0. -100 means 0% volume, or completely turned down. 0 means 100% volume, or not turned down at all. **GameMaker can't make your audio assets any louder than what they're imported as,** so it's rare to set stuff higher than 0. But sometimes it's useful if you need to offset something else that's turning it down, like a higher level bus or a nested container.

bus_get(string nameOfBus) tells you the current gain setting on the named bus.

bus_set(string nameOfBus, real newValue) sets the given bus to the given value. All containers attached to that bus or a child of that bus will update their audio in real time.

3c4. MUSIC AND AMBIANCE

You might want to set this up a different way for your project, but here's some nice little functions for setting the current background music and ambiance for your game. Crucially, you can set either or both to be containers, and generally that's what I recommend. You can also set them to be straight-up GameMaker audio assets, but I pretty much never did this.

When the music or ambiance changes, the previous one will fade out, and then the new one will start and gently fade in. You can change the default length of this fade inside the objAudio > Create Event.

music_set(string containerName OR real soundAsset) This sets the background music to be the container or audio file you specify. If you set it to a negative number, like -1, the background music will fade out and it will go silent.

music_scene_set(string containerName OR real soundAsset) This sets the "scene" background music. This is a background song which OVERRIDES the normal background music. Set it to **-4** to disable it and resume normal background music. You can also set it to **-1** to make it so that the background music goes silent.

So for example, you could be in a level with background music "forest", but when a certain story event occurs and a character appears, you could set the scene music to be that character's theme "witch." During the cutscene, the "witch" theme will play. Then at the end, you can set the scene music to -4 and the "witch" theme will fade out and the "forest" theme will resume. This way, you could code the special story event to happen anywhere in your game, and it will always resume to the correct background music after the scene is over.

ambiance_set(string containerName OR real soundAsset) This sets the background ambiance to be the container or audio file you specify. If you set it to a negative number, like -1, the background ambiance will fade out and it will go silent.

3c5. EVENT AND TIMING FUNCTIONS

These are the most fun!!!! Beats are tracked from the current background music, or the most recently played container tagged as "music" (I think).

beatEvent() returns FALSE, except on a frame where a beat is happening, in which case it returns TRUE. You can use this to make it so that an NPC does a little jump whenever a beat happens, or whatever. This is used *EVERYWHERE* in Wandersong.

doubleBeatEvent() is like beatEvent() but it returns true TWICE per beat, or you could say, on half-beats. This is to do really minute timings halfway between beats, if you need to do that.

measureEvent() returns FALSE, except on a frame where a new measure starts, in which case it returns TRUE. There's not much to this, it basically just counts beatEvent()s and only returns true once every 4 beats. You can change that number with **measureCountSet(real beatsPerMeasure).** This function is really useful if you want to wait for a good spot to transition to the next section of a song, or something like that.

beatMs() tells you how many milliseconds long a beat is at the current BPM (1000ms = 1 second). I used this to scale the speed of animations. For example, to draw a circle that lightly floats left and right in time to the beat, I would say:

```
draw_circle(x+(25*sin(pi*current_time/beatMs())),y,100,false);
```

And then no matter what the background music is, the circle would be moving in time to the song. Neato.

beatFrames() tells you how many frames long a beat is at the current BPM. Keep in mind that it may be a fractional number, especially if your BPM doesn't divide nicely with your room speed. I would use this to re-time the speed of frame animations to match the BPM, like the Bard's dancing animations.

3c6. AUDIO LOADING

To get these working, you'll need to set up grouplists in **objAudio > Create Event**. I detailed more about them in the objAudio section of the manual (3B).

audio_grouplist_load(string nameOfGrouplist), Loads in the named grouplist. What this actually does is 1. Unload audio groups that are currently loaded in and aren't in the new list, then 2. Load in audio groups in the new list which aren't loaded in yet. This way, if you're changing between two grouplists with some overlap, fewer things will load/unload. At any given time, you can only have one grouplist loaded in. And if you call loadAudioGroup() on a grouplist that's already loaded in, nothing happens.

In Wandersong we tagged each level with an associated audio grouplist, and would load that grouplist at **room start**, so that whenever a level loaded it would also load that grouplist in. If you were coming from an adjacent screen that used the same grouplist, nothing would happen; but if you were loading the game for the first time, or crossed a boundary between two different audio areas, there would automatically be a loading process without any extra checking. Nice!

audio_grouplist_isloading() just returns TRUE if audio is currently loading or FALSE otherwise. You can use this to pause the game/wait for loading to finish before advancing to the next scene, or something like that.

audio_grouplist_progress() returns a number from 0 to 100 to tell you how far along loading has progressed. Use this to draw a nice little loading bar or something while audio loads!

3c7. GAMEAUDIO FUNCTIONS

These are kinda ugly functions, but I made them so that you could provide either a GameMaker audio asset is or a string container name and they would play/stop/check the correct thing. So if you have a variable that can hold either a GameMaker sound id or a container name, this will be able to play it no matter what. I hope that makes sense. Honestly, you shouldn't use them. But they exist for the audio editor and some special cases in objAudio.

~THE END~