MUSIC TOOLKIT FOR FUZE (1.0.00)

OVERVIEW

This sequencing system works a little bit like a tracker in the sense that information about audio events is stored in arrays, and those arrays can be combined, reused, and modified to create modular audio sequences without having to explicitly redefine every single event. Unlike a tracker, this system currently has no front-end interface -- everything is done by modifying arrays within the **buildAudio()** function. The audio events available include all of those natively implemented by FUZE, though their notation differs somewhat because they are wrapped in structs rather than being written as function arguments.

AUDIO CLIPS

Audio clips are the large-scale building blocks that comprise an audio sequence. A clip is a collection of audio events that can be reused and modified with custom inputs, and its role is loosely analogous to that of a function in code.

Audio clips are defined in the **buildAudio()** function's **audioClips** array. Each entry in this array represents a single clip. A clip itself is also an array, and it consists of three parts: name, parameter header, and event data.

Element 0 in a clip is always its name (a string). This is the handle that other clips and the assembler use to reference this clip. Each clip's name must be unique.

Element 1 is always a parameter header (an array). This header defines the parameters in the clip that are exposed to modification when the clip is referenced (see **PARAMETER HEADERS**).

All other elements are event data and can be one of three things: an audio event (a struct), which tells FUZE to call an audio function (see **AUDIO EVENTS**); a clip reference (an array), which inserts another clip's events with optional parameter modifications (see **REFERENCING CLIPS**); or **"forceWrite"** (a string), which makes the assembler write its sort buffer to the file (see **EVENT ORDERING AND "FORCEWRITE"**).

AUDIO EVENTS

The core components of a clip are audio events, which are what actually tell FUZE to play a sound. An event is written as a struct, with the struct fields serving as parameters that define how the audio event is interpreted. A typical event looks like this:

```
[ .beatPos = ["beatPos", {0, 0}], .pitch = ["pitch", 0], .func = "playNote",
      .ch = 0, .wave = 3, .spd = 20, .vol = 1, .pan = 0.5 ]
```

Let's ignore the **.beatPos** and **.pitch** parameters for a moment and look at the others, which tell us that this event will trigger the **playNote()** function on channel 0. Arguments to the **playNote** event are given with **.wave**, **.spd**, **.vol**, and **.pan**, which correspond to the **playNote()** function's arguments 2, 5, 4, and 6, respectively.

EXPOSED PARAMETERS

The **.beatPos** and **.pitch** parameters are each two-element arrays, which indicates that these are exposed parameters. When a parameter is exposed, its value can be modified externally. This is similar to how a function can accept an argument. In an exposed parameter, the first element gives the parameter handle -- the name that is used externally to access the parameter -- and the second element gives a default value.

If not specified, incoming modifier values are added to an exposed parameter's default value. This behavior can be changed by supplying an optional third element to the exposed parameter, such as **["vol", 1, "*"]**. Valid third elements are **"*"**, which yields *incoming * default*, and **"/"**, which yields *incoming / default*. Any other character yields *default / incoming*, and omitting the element entirely results in addition.

Any parameter can be exposed, exposed parameter names do not have to be unique, and exposed parameter names do not have to match the parameter tag that they apply to (e.g. **.pan = ["panopticon", 0]** is valid).

.BEATPOS

The **.beatPos** parameter takes a vector2 that represents measure and beat -- **{MEASURE, BEAT}**. Note that both measure and beat follow coding convention and begin at 0 rather than 1, so the first beat of the first measure is written as **{0, 0}**. The beat can be expressed as a float: the fractional value determines distance through the beat. For example, **{0, 1.25}** is a quarter of the way through the second beat of the first measure. The number of beats in a measure at any given point is determined by the tempo map (see the **TEMPO MAP** section).

The value of **.beatPos** is subject to addition and subtraction and can be modified when exposed as a parameter. Typically, a note sequence within a clip that is going to be reused in multiple positions will begin at **{0, 0}** and expose its **.beatPos** parameter. This allows references to the clip to modify the **.beatPos** parameter and move the entire sequence of notes to a new position by adding measures and/or beats to the default position.

.TIMEPOS

The **.timePos** parameter is a time offset given in seconds that is not subject to tempo changes. It can be used alone or in conjunction with **.beatPos**. A typical use for **.timePos** is creating instrument effects that don't vary with the tempo, such as an automated pan whose speed is constant rather than tempo-based.

.PITCH

The **.pitch** parameter can accept two types of input: an int representing MIDI pitch (like the argument accepted by FUZE's **note2Freq()** function) or a case-insensitive string in the form *pitch-(accidental)-octave*, such as **"C4"**, **"Gb2"**, or **"a#5"**. Middle C is represented as **60** or as **"C4"**.

OMITTING PARAMETERS

Not all relevant parameters need to be given for each audio event. If a parameter is not given, the most recently used parameter value for an event with a matching **.func** parameter is used. The exceptions are **.ch**, **.beatPos**, **.timePos**, and **.func** -- these parameters are applicable to almost all events, and if omitted they will default to the values used in the most recent event regardless of whether **.func** matches.

To demonstrate:

```
[ .beatPos = ["beatPos", {0, 0}], .pitch = ["pitch", 0], .func = "playNote",
        .ch = 0, .wave = 3, .spd = 20, .vol = 1, .pan = 0.5 ],
[ .func = "setFilter", .type = 1, .cutoff = 800 ],
[ .beatPos = ["beatPos", {0, 0.25}], .cutoff = 1200 ],
[ .beatPos = ["beatPos", {0, 0.5}], .func = "playNote" ]
```

Like the first event, the second event also occurs at beat position **{0, 0}** and on channel 0 even though the **.func** parameter is different because **.beatPos** and **.ch** apply to all event types. The third event copies the second event's **.func** parameter, so it is also a **setFilter** event, and **.ch** and **.type** also carry over. The fourth event is identical to the first event except that it is played half a beat later -- **.pitch**, **.wave**, **.spd**, **.vol**, and **.pan** are all copied from the first event because it is the most recent **playNote** event.

PARAMETER AUTOMATION

Parameters can be automated, meaning that they can be set to change over time, by using **.func = "setAuto"**. This is a special type of event that doesn't correspond to a FUZE audio function. The **setAuto** event works by automatically creating interpolated events between two boundary events.

The **.rate** parameter sets the rate in seconds at which interpolation happens. Values lower than **0.0166** are clamped to **0.0166** (which is equivalent to sixty events per second, matching FUZE's max framerate). It is extremely easy to create very dense event data with automation, and extensive automation will substantially increase assembly time when you write the sequence to the file. To reduce assembly time, use high **.rate** values to produce sparser automation when you can.

The **.interpType** parameter sets the shape of the interpolation via an interpolation handle. This is identical to how FUZE's interpolate() function uses interpolation handles. Valid handles are **constant**, **linear**, **ease_in**, **ease_out**, **ease_in_out**, **expo_in**, **expo_out**, **expo_in_out**, **bounce_in**, **bounce_out**, **bounce_in_out**, **elastic_in**, **elastic_out**, and **elastic_in_out**.

The **.events** parameter is an array of two boundary events to interpolate between. You cannot interpolate between two different types of event, so the events' **.func** parameters must match. In theory, any other parameter can be interpolated, but not all have been tested and some may exhibit odd behavior (e.g. interpolating **.ch**). The only events that must be interpolated are **.beatPos** and/or **.timePos**: there needs to be a timespan for interpolation to happen, and the second position must be later than the first. Any parameters that you don't want to interpolate should be left the same (or omitted) in the second event.

A typical **setAuto** event looks like this:

```
[ .func = "setAuto", .rate = 0.032, .interpType = linear, .events = [
       [ .beatPos = {0, 0}, .func = "setVolume", .ch = 0, .vol = 2 ],
       [ .beatPos = {0, 1}, .vol = 0.5 ] ] ]
```

This creates a volume fade from **2** to **0.5** across one beat. Note the omission of **.func** and **.ch** in the second event. As discussed in the **OMITTING PARAMETERS** section, **.func** and **.ch** are copied from the previous event when not explicitly given.

EVENT LIST

This is a complete list of audio event types and the parameters that apply to them. Most parameters behave identically to their native FUZE function argument counterparts.

.func = "playAudio"

- .ch (int)
- .beatPos (vector2)
- .timePos (float)
- .sampleIdx (int) -- ***Index of a sample in buildAudio()'s samples array (see the SAMPLES section).***
- .root (int or pitch) -- ***This is the pitch of the sample at speed = 1, which is needed to calculate .pitch. You can still use .pitch if you don't set .root, but pitches won't match other instruments' pitches.***
- .pitch (int or pitch)
- .vol (float)
- .pan (float)
- .spd (float) -- ***Ignored if .pitch is also given, because pitch is applied by automatically changing speed.***
- .loops (int)

.func = "playNote"

- .ch (int)
- .beatPos (vector2)
- .timePos (float)
- .wave (int)
- .pitch (int or pitch)
- .freq (float) -- ***Ignored if .pitch is also given.***
- .vol (float)
- .spd (float)
- .pan (float)

.func = "setClipper"

- .ch (int)
- .beatPos (vector2)
- .timePos (float)
- .thresh (float)
- .strength (float)

.func = "setEnvelope"

- .ch (int)
- .beatPos (vector2)
- .timePos (float)
- .spd (float)

.func = "setFilter"

- .ch (int)
- .beatPos (vector2)
- .timePos (float)
- .type (int)
- .cutoff (float)

.func = "setFrequency"

- .ch (int)
- .beatPos (vector2)
- .timePos (float)
- .freq (float)

.func = "setModulator"

- .ch (int)
- .beatPos (vector2)
- .timePos (float)
- .wave (int)
- .freq (float)
- .scale (float)

.func = "setPan"

- .ch (int)
- .beatPos (vector2)
- .timePos (float)
- .pan (float)

.func = "setReverb"

- .ch (int)
- .beatPos (vector2)
- .timePos (float)
- .delay (float)
- .atten (float)

.func = "setVolume"

- .ch (int)
- .beatPos (vector2)
- .timePos (float)
- .vol (float)

.func = "startChannel"

- .ch (int)
- .beatPos (vector2)
- .timePos (float)

.func = "stopChannel"

- .ch (int)
- .beatPos (vector2)
- .timePos (float)

.func = "setAuto"

- .rate (float)
- .interpType (interpolation handle)
- .events (two-element array of events)

PARAMETER HEADERS

We've already looked at parameter exposure within individual events, but in order for those parameters to be accessible from outside of a clip, we must also expose the parameters via a parameter header. A parameter header is an array that always occurs as the second element of a clip, and it collects all of the exposed parameters used within the clip. The syntax for each element of a parameter header is exactly the same as for parameters exposed within events -- element 0 is name, element 1 is default value -- except that it cannot have a third element (a header's defaults can only be added to incoming values).

A typical parameter header looks like this:

```
[ ["beatPos", {0, 0}], ["pitch", 0], ["vol", 1] ]
```

If no parameters need to be exposed, the header array can be empty, but it must not be omitted entirely.

REFERENCING CLIPS

The first element of each clip is a handle that can be used by other clips to reference it. These references are used similarly to audio events and can be thought of as shortcuts to the referenced clip's contents. A clip reference has two parts: a parameter header that lists the exposed parameters that we want to modify and a string that specifies the name of the clip we're referencing.

The syntax for the parameter header is the same as is used for parameter headers within clips. A parameter header in a reference can omit any parameters that don't need to be modified, and as with clips' headers, if no parameters are needed, the array can be left empty but cannot be omitted.

If any of the parameters given in the reference's header is not actually exposed by the referenced clip's parameter header, it is ignored.

Here's what a normal clip reference looks like:

```
[ [ ["beatPos", {11, 0}], ["pitch", 12], ["vol", 0] ], "clip1"]
```

In this reference the clip named clip1 is being placed at measure 11 and transposed up an octave. Volume is not being changed and could be omitted.

TEMPO MAP

Because the **.beatPos** parameter refers to measure and beat position, we need to define the tempo and time signature for a sequence so that these values can be resolved into actual time positions during playback. This is done by modifying **buildAudio()**'s **tempoMap** array.

The **tempoMap** array contains an element for each tempo/meter definition, and each element is a struct with three fields: **.beatPos**, **.bpm**, and **.timeSig**. There must always be at least one element in **tempoMap** because every sequence needs at least one tempo and one time signature. Additionally, the **.beatPos** of the first (or only) element must be **{0, 0}**, which places it at the start of the sequence.

An example of **tempoMap**:

```
var tempoMap = [
        [
                // Measure and beat where the tempo/time signature occurs. First
                // one must be {0, 0}.
                .beatPos = {0, 0},
                .bpm = 120, // 120 beats per minute.
                .timeSig = 4 // Four beats per measure.
        ],
        [
                // Subsequent tempo/meter changes can occur anywhere.
                .beatPos = {4, 0},
                .bpm = 90,
                .timeSig = 3
        ]
]
```

SAMPLES

Unlike FUZE's native **playAudio()** function, the **playAudio** event does not directly reference an audio file handle. Instead, it uses the **.sampleIdx** parameter to index into **buildAudio()**'s **samples** array. This array contains filepaths for audio samples, and you should modify its contents to fit your needs. When the audio sequence is loaded from the file for playback, the audio files that these paths point to are also automatically loaded, and the **playAudio** events' index references are resolved to the actual audio file handles.

An example samples array looks like this:

```
var samples = [
	"Gijs De Mik/FX_Retro_Car_02",
	"Gijs De Mik/FX_Laser_15"
]
```

.ROOT

In order to accurately repitch audio files via **.pitch** in a **playAudio** event, the root pitch of the audio file must be set with the **.root** parameter. This root pitch represents the pitch of the audio at its default speed of **1**. FUZE does not have metadata about audio file pitch, so you must determine this root pitch on your own (for example, by matching the pitch against a piano). **playAudio** events can still be repitched if **.root** isn't set or is set inaccurately, but the resulting pitches won't match with other pitched audio events.

EVENT ORDERING AND "FORCEWRITE"

Events of the same type on the same channel must be listed in chronological order. If they are listed out of chronological order, some of them may be deleted during assembly. This is by design: in the absence of a more complicated prioritization algorithm, deleting later events when placing earlier events of the same type allows for the creation of instruments with parameters that change over time without the concern that events placed for one note will conflict with subsequent notes. Instead of retaining the events that would cause conflict, the subsequent note's events simply delete any of the previous note's events that would have overlapped.

Events that are not of the same type or that are not on the same channel can be placed out of chronological order; however, because the assembly process ultimately needs to sort events chronologically, it is a good idea to keep the timespans within which events need to be sorted short. This is done by periodically telling the assembler to write the contents of the sort buffer to the file using the **"forceWrite"** keyword, which also (partially) clears the sort buffer. **"forceWrite"** can be placed at any index in a clip except at indices 0 and 1.

Depending on event density, it is usually a good idea to call **"forceWrite"** from the top-level clip every measure or two. Longer spans without writing will still assemble properly, but assembly speed may suffer.

The time boundaries between writes do not need to be clean. If, for example, the final event before a write occurs at beat position **{2, 1}** and the first event after the write occurs at **{1, 3.5}**, the events will most likely still sort and write properly. The sort queue is not completely cleared when a write is forced, so events near the boundary will still sort against one another.

If you encounter a place in playback where the audio pauses briefly before continuing, it is likely caused by an event that has not been sorted properly. The solution is to move the event to the other side of a write, to move the location of the write, or to remove the write altogether.

It is not necessary to force a write at the end of a sequence -- the assembler will automatically write whatever remains in the sort buffer when it reaches the end of the sequence.

ASSEMBLY

To assemble an audio sequence, call **assembleAudioSequence(_clipName, _saveAsName, _buildResult)**. This function both assembles a clip and writes it to the file. Its first argument, **_clipName** (a string), is the handle of the clip to assemble, and the second argument, **_saveAsName** (a string), is the name to save the sequence under. The third argument, **_buildResult** (struct), is the return from calling **buildAudio()**.

Typical use:

```
assembleAudioSequence("musicClip", "ExampleMusic", buildAudio())
```

The assembler can process only a single clip, but that clip can itself contain any number of clip references, and there is no recursion limit on how deep a chain of references can go as long as circularity doesn't occur. When building a sequence, everything in the sequence should ultimately be contained via reference in this top-level clip that gets passed to the assembler.

The assembly process takes some time but only needs to be performed once. After the sequence has been assembled, it exists permanently in the text file unless deleted or overwritten, and it can be streamed from the file almost instantly.

STREAMING

To stream a saved sequence from the file, we first call **initAudioQueue(_name, _file, _timeOffset, _deferLimit)** and store its return in a variable. This function has two required arguments: **_name** (string) is the name of the sequence to load, and **_file** (file handle) is the file to read the sequence from.

initAudioQueue() also has two optional arguments. **_timeOffset** (float) (default 0) is a value in seconds to offset playback. It can be used to delay the start of playback (negative value) or to fast-forward into playback (positive value). **_deferLimit** (int) (default 1) determines how many additional file reads are allowed per frame if the streamed data queue is immediately cleared. Large values may reduce the framerate.

Once we've initialized the queue, calling **streamAudioQueue(ref _queue, _file)** every frame will stream audio events from the file. The function takes two arguments: **_queue** (struct) is the return from **initAudioQueue()** that we stored in a variable, and **_file** (file handle) is the file to read the sequence from.

streamAudioQueue() both returns an updated version of **_queue** as well as updating the original variable passed as **_queue**. Among other things, this update includes storing the current frame's audio events in **_queue.lastAudio**, which (as in the demo) can be used to drive a visualizer.

The loop for playback looks like this (note that the file must remain open while streaming occurs):

```
var file = open()
var queue = initAudioQueue("ExampleMusic", file)

loop
	streamAudioQueue(queue, file)
repeat
```

LARGE-SCALE EXAMPLE

A typical workflow is to define clips that represent instruments with exposed **.pitch** and **.beatPos** parameters then use references within other clips to pass pitch and position values to them.

Here's what creating and referencing an instrument clip looks like in practice:

```
function buildAudio()
	var audioClips = [
		// Instrument definition
		[
			// Clip name
			"Synth1",
			// Parameter header exposes .beatPos and .pitch.
			// Defaults are 0, so incoming values will not be
			// changed.
			[ ["beatPos", {0, 0}], ["pitch", 0] ],

			// playNote() is called at the position and pitch passed
			// from outside.
			[ .beatPos = ["beatPos", {0, 0}], .pitch = ["pitch", 0],
				.func = "playNote", .ch = 0, .wave = 3,
				.spd = 20, .vol = 1, .pan = 0.5 ]
```

```
            // A sound file whose path is given in the samples array
            // is played in addition to the synth tone defined
            // above. This sample's default pitch, given with
            // the .root parameter, is A3.
            [ .beatPos = ["beatPos", {0, 0}], .pitch = ["pitch", 0],
                   .func = "playAudio", .ch = 1, .sampleIdx = 0,
                   .spd = 1, .vol = 4, .pan = 0.5, .loops = 0,
                   .root = "a3" ]
     ],
     // Note sequence
     [
            // Clip name
            "Notes",
            // Parameter header exposes .beatPos and .pitch, so
            // these notes can be moved and transposed.
            [ ["beatPos", {0, 0}], ["pitch", 0] ],

            // Calls the clip named "Synth1" and gives it the
            // parameter values ["beatPos", {0, 0}] and
            // ["pitch", "c4"].
            [ [ ["beatPos", {0, 0}], ["pitch", "c4"] ], "Synth1"],
            // MIDI pitch values can be used interchangeably with
            // pitch strings. (MIDI pitch 62 = "D4".)
            [ [ ["beatPos", {0, 1}], ["pitch", 62] ], "Synth1"],
            [ [ ["beatPos", {0, 2}], ["pitch", "e4"] ], "Synth1"],
            // Beat 2.5 means halfway through the third beat.
            [ [ ["beatPos", {0, 2.5}], ["pitch", "f#4"] ],
                   "Synth1"],
            [ [ ["beatPos", {0, 3}], ["pitch", "g4"] ], "Synth1"],
            // First beat of the second measure.
            [ [ ["beatPos", {1, 0}], ["pitch", "c5"] ], "Synth1"]
     ],
     // Any clip can be referenced by another clip (but don't create
     // circular references). This clip transposes "Notes" up a perfect
     // fifth (seven semitones) by passing it a pitch modifier.
     [
            // Clip name
            "NotesUpFifth",
            // Parameter header doesn't expose .pitch, so anything
            // that references this clip can't change its pitches.
            [ ["beatPos", {0, 0}] ],

            [ [ ["beatPos", {0, 0}], ["pitch", 7] ], "Notes"]
     ],
     // This clip combines our note sequences into a longer passage. It
     // is the one we give to the assembler.
     [
            "Melody",
            // Empty parameter header means no exposed parameters.
            [],

            [ [ ["beatPos", {0, 0}] ], "Notes"],
            // Play the clip starting at the third measure.
            [ [ ["beatPos", {2, 0}] ], "NotesUpFifth"],
			// Make the assembler write to the file to reduce the
			// length of the sort buffer.
			"forceWrite",
			// Play the clip starting at the fifth measure and
			// transpose it down an octave.
			[ [ ["beatPos", {4, 0}], ["pitch", -12] ], "Notes"],
			// Stop both channels at the end of the sequence to make
			// sure the last events don't hang (see KNOWN ISSUES).
			[ .beatPos = {5, 1}, .func = "stopChannel", .ch = 0 ],
			[ .beatPos = {5, 1}, .func = "stopChannel", .ch = 1 ]
		]
	]

	// The samples array is used by the playAudio event's .sampleIdx parameter.
	var samples = [ "Gijs De Mik/FX_Misc_35" ]

	var tempoMap = [
		[
			// The first (or only) tempo/meter definition must be
			// placed at {0, 0}.
			.beatPos = {0, 0},
			.bpm = 96,
			.timeSig = 4
		]
	]

	var result = [
		.clips = audioClips,
		.samples = samples,
		.tempoMap = tempoMap
	]
return result

// This call to the assembler can be commented out or removed after running it once.
// It only needs to be rerun if the data in buildAudio() is modified.
assembleAudioSequence("Melody", "InstrumentExample", buildAudio())

var file = open()
var queue = initAudioQueue("InstrumentExample", file)

loop
	streamAudioQueue(queue, file)
repeat
```

KNOWN ISSUES

- Final events may hang at the end of the sequence. This can be avoided by providing a **stopChannel** event for affected channels.
- There may be a tempo hiccup at the start of playback because the sequence wants to begin before the load has completed. This can be avoided by giving a slight negative value for **initAudioQueue()**'s **_timeOffset** argument.
- Playback at a low framerate may suffer from timing inaccuracies. FUZE can't call audio functions faster than the framerate, so reducing the framerate also reduces the timing accuracy of audio events.