Audio Cue overview

Johan Sundhage, Klevgränd produktion 2012

Contact	
E-Mail	audiocue@klevgrand.se
Twitter	@audiocue
Web	http://klevgrand.se/audiocue

1. Adaptive Music

1.1 Adaptive music - what is it?

A short definition: ***adaptive music*** is music that responds to *interactions*. An interaction can be anything from a user clicking a button on a web site to a dramatic mood-altering event in a game. There are many different ways of how music can adapt, all from simply changing the volume of a track to altering the harmonies in a strings section.

The original purpose of this product was to make it possible for educated composers and music-producers to make their music *adaptive* (respondent to interactions) without ending up with weird sounding music when implemented in an application. It was also meant to provide a maximum amount of separation between the developer and the composer. How stuff sounds should never have to depend on anyone but the one in charge of sound and music; the composer.

1.2 Adaptive music - what is the problem?

Composing music for film and TV has been done for decades and there are several techniques to achieve synch between the visuals and the music (yes in almost every case the music is composed for the movies' final cut, not the other way around). Epic film scoring composers, like *Ennio Morricone* or *John Williams,* have a very deep knowledge of how to make the music *mean something* when combined with moving pictures. This wisdom is passed on to music students world wide, so there is a whole bunch of really talented professionals out there who knows about the most important part of writing music for other media; what its dramatic, aesthetic and functional purpose is, how it should sound and *when* it should sound.

So, if a team of software developers wants some Hollywood-ish film-score-sounding music to their interactive application, why not just hire some film-scoring composer and make him or her deliver those epic tracks in a week or two? Many have probably tried, but it will almost certainly end up in the wrong kind of drama. This is why:

The interactive application is not *linear*. A film-scorer wants a *timeline* to hook the music up to. This might be weird-sounding for non-scorers, but these hooks (or synch points) are extremely important to a film scorer as they're used to define tempo, time signature, form; i.e. everything that makes the music seamlessly fit into the movie. It's not pure luck when the key changes and a contrabass riff starts playing as the bad buy shows up facing the camera in that dark alley.

An interactive application does not have a timeline and that will give the composer troubles with how to actually compose. Many of the synchronisation techniques will be unusable because the hooks (or synch points) will change depending on what happens in the application. A carpenter needs a hammer to hit the nail and a film scorer needs linear media to make good music. The lack of linearity forces the composer to figure out new ways of composing music, which most certainly will take up a lot of time and disturb the creative flow. Finally it will end up in some "not-as-good-as-if-it-was-a-movie" compositions.

The developer and the composer speak two different languages. Even if they (at least) think they are on the same page during brief meetings, and agree on when different pieces of music should start and stop playing, there *will* be problems during the development process. If the composer delivers a couple of loop-able music pieces (with a few transitions) and tries to explain how they should be implemented it could sound something like this:

- *Here's the music. I separated the horns into two dynamic layers so if you should fade them; use the pianissimo one's when it should be quieter. And never start them before the third bar; just don't use them if the adagio section passed that position. All transitions should work fine, but never play the second one combined with the last piece, just go straight to the last piece on any beat.*

And here's the developer-friendly translated version.

- *Here's the music. The horns (horns_*.ogg) are divided into two gain layers, never play them at the same time. Do not multiply the gain level with the horns track when rendering the output mix! Instead; if the gain-level exceeds a factor of 0.6, replace horns_staccato_forte.ogg with horns_staccato_pianissimo.ogg. If the playhead position is greater than 2.33 secs (or 102 753 frames) and softmx.ogg is playing; never start the horns_* files at all. trans_to_1.ogg, trans_to_2.ogg and trans_to_3.ogg should be played when changing track (trans_to_1 should play through the end before song1 starts etc). The tempo is 120 BPM, and transitions could only start at an even beat. That means when a track position in frames is dividable with 22050 (60/120s -> 22050 frames @ 44.1kHz) the change can take place. You'll have to cue that up. If there is no transition for a track, just play that track on the next beat. It's very important that change will take effect on that exact frame. If it doesn't, I'll hear it (you won't).*

Even if the composer and the developer somehow manage to sort out what all the above means it will be quite a task for the developer to implement all the features above and the composer won't be certain of how it will actually sound in the application until the implementation is done, the app is compiled and finally handed over to the composer for viewing. After reviewing it the composer probably wants to make a couple of small changes and the whole process has to iterate again... Pretty soon some executive producer who is responsible for deadlines, budgets and such will step in and put an end to it, and the application will end up having "not-so-very-good-implemented" music.

The composer will say to himself: *"No one understands me! I'm going straight back to film scoring and`ll never get my hands dirty with this stuff again"*

The developer will say: *"We should have skipped the music and used some ambient sound-effects downloaded from freesound.org instead. I wanna code fancy triangle culling algorithms and shaders using OpenGL, not have to deal with stupid italian musical linguistics."*

The producer will say: *"I agree with the developer."*

The composer won't be able to establish a decent work flow. Even if the composer is able to produce fantastic music to a screen capture of the application, it will be very tricky to actually test it in its final environment - the real application. Without being able to run the music with the application it's impossible to be certain of if the music actually will pan out as expected. There are a lot of pitfalls, unique for each project, which can be avoided if the composer frequently has the ability to independently test and iterate the compositions during development.

There are probably a lot of more cases and examples of why a traditional film-scorer will have troubles making adaptive music for interactive applications, but the ones listed above are the obvious ones. Fortunately Audio Cue resolves most of them.

1.3 Adaptive music - what is the point?

Seamless sound and music in interactive applications will make the graphical implementation look and feel better. When experiencing sound and graphics at the same time, the human mind combines the two medias into one. Just like a very good graphical implementation makes the music sound better, a good musical implementation makes the moving pictures look better. A good musical implementation means music that synchronises and plays well with graphics in an aesthetic, dramatic, narrative and functional way.

Adaptive music makes a better immersive design. A virtual world that doesn't make any noise won't convince many users. If the music reflects some of the world's attributes, it's narrative and the level of dramatic tension, end-users will get more involved and hopefully spend more time in i it.

Non-adaptive music will sound repetitive and eventually get boring no matter how good it is. Since a composer won't be able to make synchronised music for every possible interaction on all possible timings, there *must* be some degree of re-use of musical content. If the implementation just loops one music piece through the whole game it will sooner or later become annoying, even if the music is a top-notch composition and performed by the London Symphony Orchestra. Changing the music in a musical way based on application events or user interactions prevents it from sounding repetitive since the actual change never will happen on the exact same song position.

It's just freaking cool! Yep, a well-made and fairly complex implementation of adaptive music *will get noticed* and in the end generate some buzz around the application. This might change over time, but since good musical adaption still is rare, it will be a pretty unique and noticeable feature of the application.

2. Brief

2.1 What is Audio Cue?

Audio Cue is a middleware audio and music engine that focuses on *adaptive music*. It basically consists of two parts; an editor where the composer defines all sound, music and rules of how it can - *and should* - be play-backed and a pre-compiled library (one for each platform) to be used for implementing the composers work.

The engine is based on an event system. The composer defines a set of events to be called by the developer. When an event is dispatched (in a real situation it could be when a game starts, ends or the player reaches a new level) Audio Cue looks up what *actions* the composer has defined for that particular event, and the sound and/or music responds to it. All events return a timing value that tells the developer *when* something will happen; for example how many seconds there is until the musical change to the refrain or when the contrabasses have faded out.

The Audio Cue core engine is built from ground-up in C/C++ with very few dependencies. This makes it portable to platforms such as Adobe Flashplayer via the Alchemy tool chain and Google Chrome’s Native Client (NaCl).

2.2 Who should and can use Audio Cue?

Audio Cue is targeting two types of professionals: composers/music producers for the creating of adaptive music, and developers for implementing and using in it in real applications.

A composer/music producer who is used to working in a digital environment (a DAW like Logic, Pro-Tools or Cubase) will have no problems understanding the concepts of Audio Cue and can easily start creating adaptive music.

Any developer implements Audio Cue into their application in a few simple steps.

2.3 What is the difference between Audio Cue and other audio engines?

Even if Audio Cue handles ordinary sound effects, realtime DSP effects and audio routing in different ways, the main focus is *making the music adaptive*. The whole framework is developed and frequently tested by music professionals and the Audio Cue developers are very serious about letting the composer keep full control of how the music will sound and adapt in the finished and deployed application. If a composer doesn’t have a creative workflow - *with easy-to-use GUI based tools and sophisticated fast-iterating testing environments* - there will be no good musical adaption. It actually doesn't matter how fancy the core engine algorithms are if the composer is frustrated during the creative process.

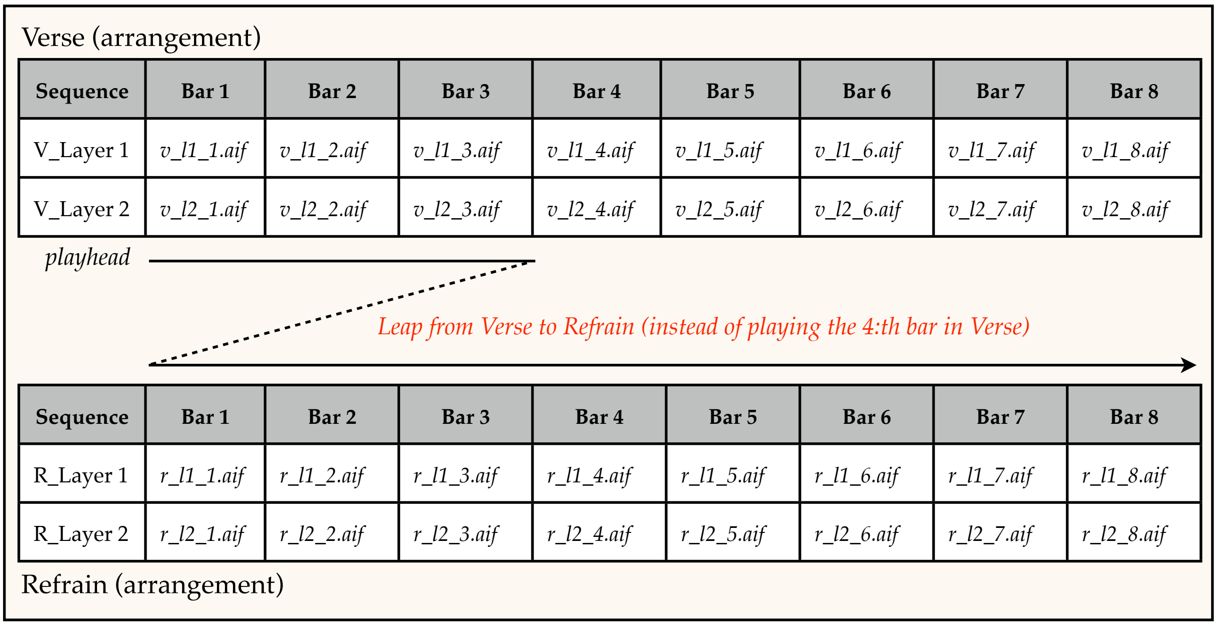
In addition to an easy-to-use editor the Audio Cue Core API has support for scripted plug-ins. These plug-ins are dynamically parsed and can be added/removed during runtime and won't need to be recompiled or have changes done in any binary files. A composer will not have to learn how to script new plug-ins to make great adaptive music but have the possibility to import and use them in an easy way.

3. The Audio Cue engine

The purpose of the Audio Cue playback engine is to make the music sound as if there actually were real musicians interpreting and performing a song, fully comparable to how a live theatre orchestra would do it; when the actor reaches a pre-defined line in the script, the bandleader cues the orchestra to move on to the next piece in a musical and "nice-sounding" way without breaking the tempo, tampering with harmonics or signature in a way that makes the music sound weird.

A composer has his/her own workflow when it comes to composing, recording and mixing music. There are several commercial DAW's[1] *out there and most of them are nowadays equally competent. Audio Cue encourages the composers to work in their familiar environment as long as possible, keeping the creative process and workflow they're used to when producing music.*

The Audio Cue structure is all about breaking up the music into small pieces and then letting the music engine re-assemble them in realtime based on rules made up by the composer.
A composer creates, for example, a couple of 8-bar pieces of music that makes up a song, splits up each piece into several shorter sounds (1 bar each) and then puts them back together in different *sequences* and *arrangements*. The duration of these short sounds defines when Audio Cue can leap over to a different *sequence* of sounds without messing with the musical continuity. Note that it's actually the *composer* that decides the length of the sounds and in the end when the music has the ability to change.



Verse (arrangement)

Sequence	Bar 1	Bar 2	Bar 3	Bar 4	Bar 5	Bar 6	Bar 7	Bar 8
V_Layer 1	*v_l1_1.aif*	*v_l1_2.aif*	*v_l1_3.aif*	*v_l1_4.aif*	*v_l1_5.aif*	*v_l1_6.aif*	*v_l1_7.aif*	*v_l1_8.aif*
V_Layer 2	*v_l2_1.aif*	*v_l2_2.aif*	*v_l2_3.aif*	*v_l2_4.aif*	*v_l2_5.aif*	*v_l2_6.aif*	*v_l2_7.aif*	*v_l2_8.aif*

playhead
Leap from Verse to Refrain (instead of playing the 4:th bar in Verse)

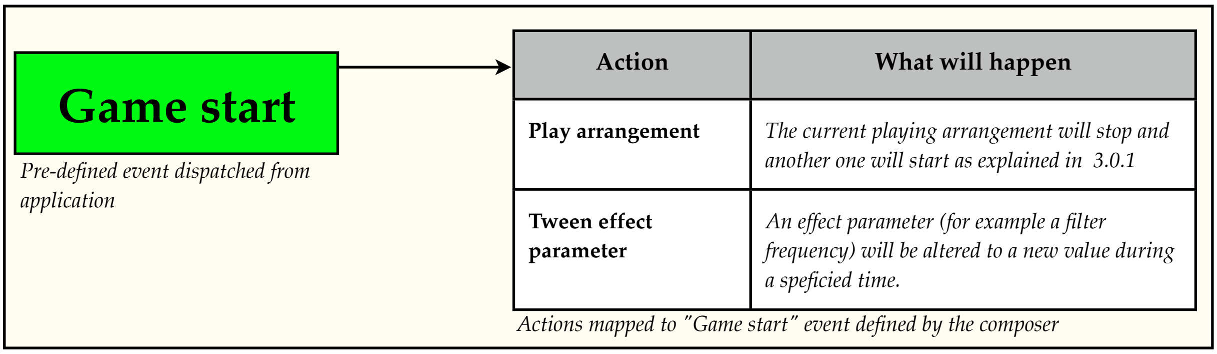
Sequence	Bar 1	Bar 2	Bar 3	Bar 4	Bar 5	Bar 6	Bar 7	Bar 8
R_Layer 1	*r_l1_1.aif*	*r_l1_2.aif*	*r_l1_3.aif*	*r_l1_4.aif*	*r_l1_5.aif*	*r_l1_6.aif*	*r_l1_7.aif*	*r_l1_8.aif*
R_Layer 2	*r_l2_1.aif*	*r_l2_2.aif*	*r_l2_3.aif*	*r_l2_4.aif*	*r_l2_5.aif*	*r_l2_6.aif*	*r_l2_7.aif*	*r_l2_8.aif*

Refrain (arrangement)

3.0.1 Musical change between two arrangements

[1] *Digital Audio Workstation - software for recording, editing and mixing sound and music.*

These compilations of sound *sequences* and *arrangements* can be started, stopped and altered by different *actions*, which are executed by *events* (defined by the developer and the composer together) and dispatched by the application. In the theatre world the application is the actor, the engine is leading the band and the composer still gets to be the composer - with some extra tools to keep control of everything.



Game start

Pre-defined event dispatched from application

Action	What will happen
Play arrangement	*The current playing arrangement will stop and another one will start as explained in 3.0.1*
Tween effect parameter	*An effect parameter (for example a filter frequency) will be altered to a new value during a speficied time.*

Actions mapped to "Game start" event defined by the composer

3.0.2 One event triggering two actions

The composer also has full control of internal audio routing including adding DSP effects and tweening their parameters at runtime, everything via *actions* and *events*.

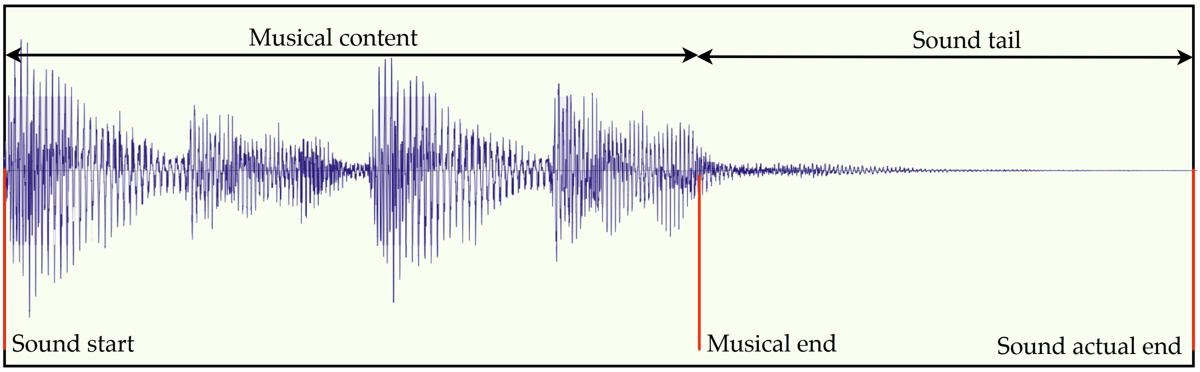
3.1 Musical Structure

This chapter will in detail explain the musical structure used in Audio Cue. This part is very important as it explains the foundation of how Audio Cue makes the music adaptive.

All music consists of a hierarchy based on four different levels.

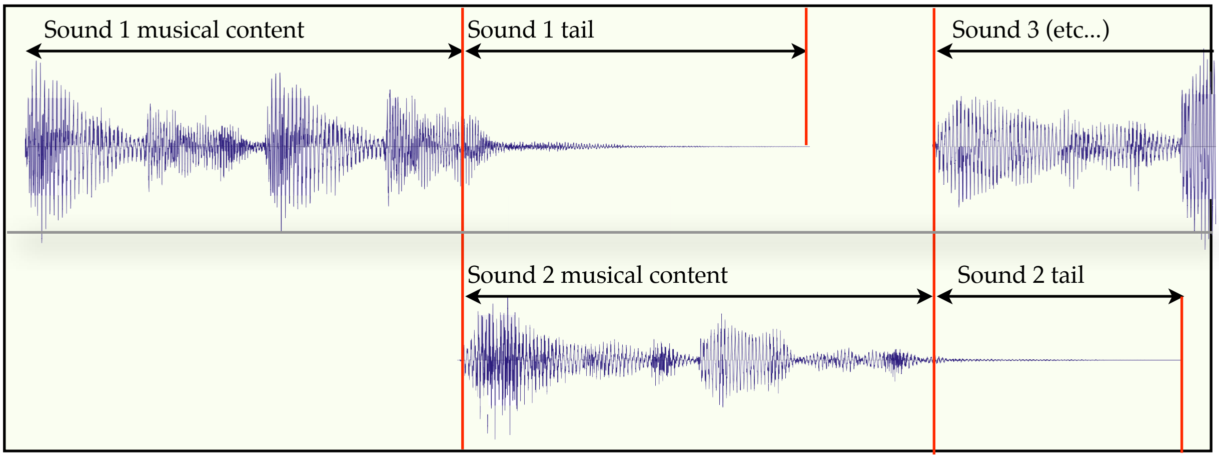
Level / Object	What is it?	DAW equalient
4 / Sound	An almost-ordinary sound file (see 3.1.1).	A sound in the bin
3 /Sequence	A series of sounds	An audio track
2 / Arrangement	A list of sequences that will play at the same time.	A music part (verse, intro, refrain, interlude...)
1 / Domain	A list of arrangements that belong together.	A project or song

3.1.1 Sound



3.1.1.1 A sound and its tail

A Sound is the smallest building block of a song (or *domain*). Unlike an ordinary sound file, an Audio Cue sound has *two* length definitions. The obvious one is the complete sound length, and the other, not-so-obvious one, is the *musical length*. The purpose of defining a musical length is to let the music engine know when it's permitted to start a cued sound, of course while still playing the tail of the previous one. Letting the composer define a musical length for each sound is one of the key features to making it possible to let the music sound as good as if it was interpreted by real musicians in realtime. When bouncing a bar of music in a DAW one should be explicit about including the reverb and the decay of the playing instruments after the end of the bar - also called the *tail*. In some DAW's the tail can be configured to automatically include when bouncing a selection.

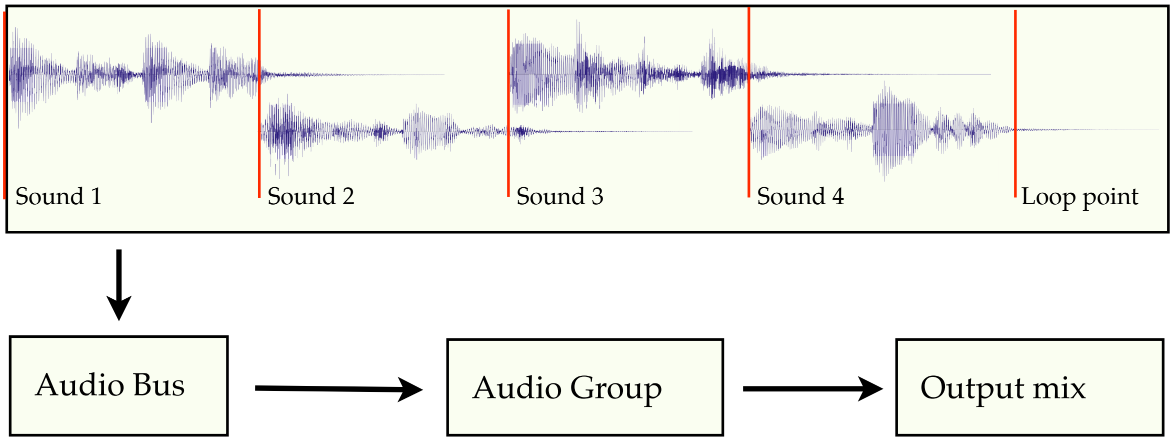


3.1.1.2 Playing cued sounds in musical synch

Audio Cue is by default playing the full length of the sound, but if there is another sound on cue it will start exactly at the running sounds' *musical end*. Audio Cue will in that case be playing two sounds at the same time; the newly started sound and the previous sounds' *tail*. This is very important if the next sound to be played is a transition to another part - *or actually anything that wasn't recorded in the original order* - playing the tail of reverb and instrument decays prevents the feel of a hard cut in the mix and will sound like if the musicians just cued out to a new part of the song.

A special type of sound is *Silence*. A *silence* sound is just a void to be used for leaving gaps/pauses in sequences, which in some cases can be necessary.

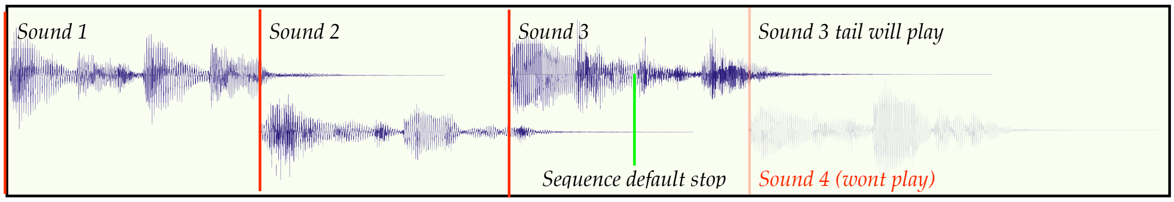
3.1.2 Sequence



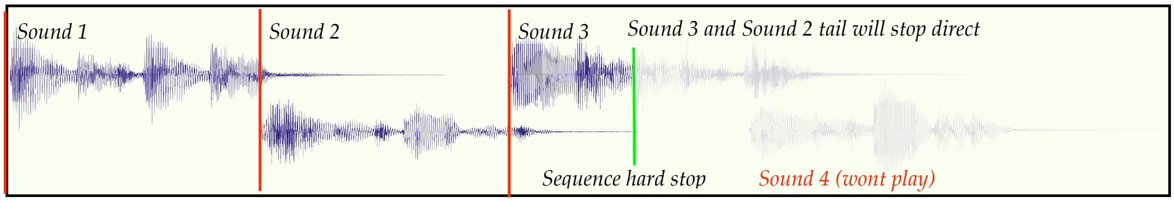
3.1.2.1 A sequence and its way to the sound card output

A sequence is made up of a series of sounds and is also the first level of playable (and loopable) objects. A looped sequence will play the series of sounds over and over again until it is stopped. It is also routable and should be attached to a *bus* and optionally a *group*.

When creating a sequence of sounds it is not possible to manually define timing offsets. The sounds musical lengths automatically defines the sequence length, and sounds in a sequence can not be played in parallel except when a sound's tail is playing and the next sound starts (*see image 3.1.1.2*). To make gaps between sounds, one should use specially created *silent* sounds.



3.1.2.2 A default sequence stop

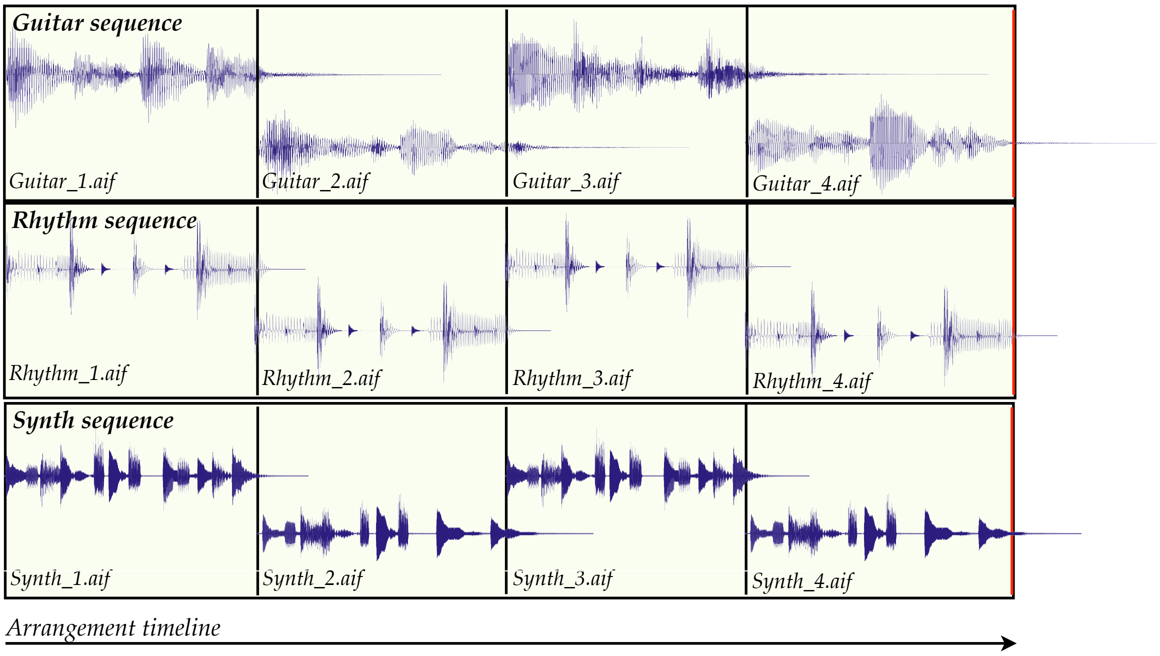


3.1.2.3 A hard sequence stop

A sequence can be stopped in two ways. By default the sequence stops after having finished playing the current sound and its tail. The other option (called *hard*) is to stop the current playing sound immediately without caring about musical timing.

To musically make a leap from one playing sequence to another Audio Cue lets the current playing sound reach its musical end and starts the next sequence at that exact sample position. This renders a natural-sounding (on a sound-engineering technical level) transition as the first sound's tail will be overlapping while the new sound starts playing.

3.1.3 Arrangement



3.1.3.1 Arrangement structure example

A group of sequences that should be played simultaneously is called an *arrangement,* which in musical terms corresponds to a music part (like an intro, verse, refrain or ending). Arrangements are not *routable* (attachable to groups or busses), but is the most common way of playing music in Audio Cue. Even if a song is built on single sequences they should be put in separate arrangements to easily be played using proper *actions*. An arrangement must be attached to a *domain*.

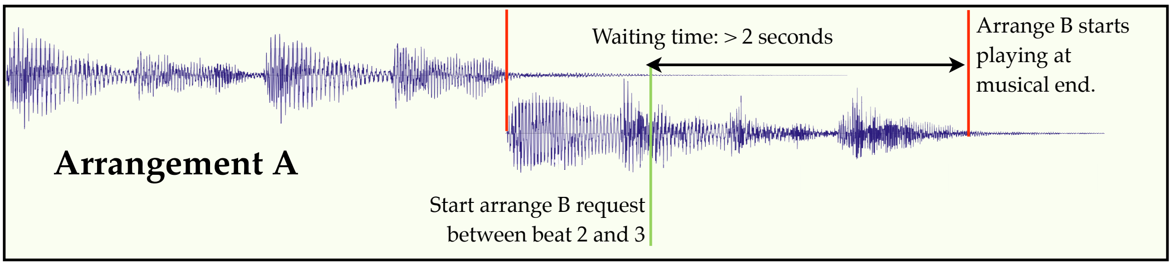
Note that all sequences in one arrangement are not obligated to have the same number of sounds. Each sequence's timeline is individual and will loop internally. An arrangement containing sequence *A* with instruments playing a 4 bar chord progression can be combined with sequence *B* containing 32 bars of vocals. When the arrangement's playhead reaches its fifth bar, sequence *A* will restart at the first bar if its loop flag is set.

If different sequences have sounds with different musical lengths Audio Cue will resolve that and make the change when the last sound reaches its musical end.

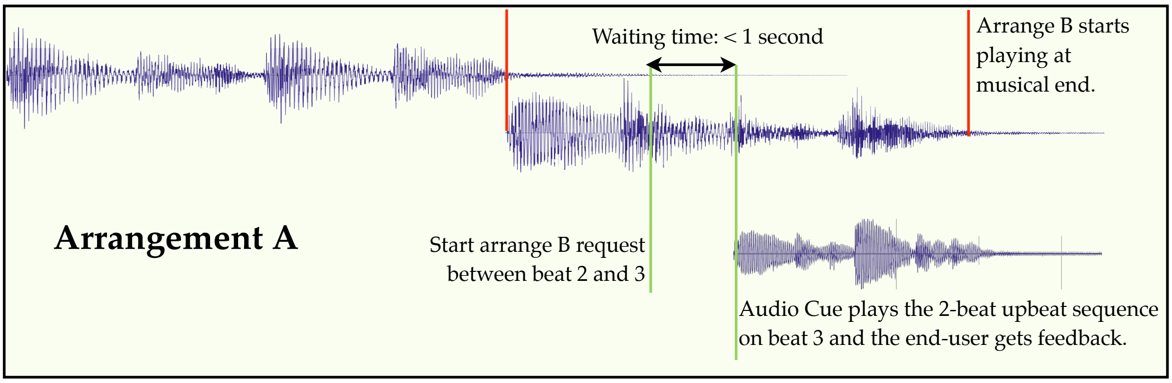
When planning a musical structure for Audio Cue, always try to keep as few parallel sequences as possible. Most times it’s enough to play one sequence at a time; the more sounds that are playing at the same time the more CPU is drained, and on platforms like Adobe Flash or on mobile devices all parts of an application should be as optimised as possible.

An arrangement can in addition to ordinary sequences also contain *upbeat* sequences. These sequences are commonly short and should only contain one sound each. Which one (if any) of these sequences will be played depends on how much time there is until the arrangement has the ability to start.

For an example, let’s say we’re playing an arrangement (*A*) where the sequence(s) are divided in 1-bar-sounds at 60 BPM - each sound has a musical length of 4 seconds. If we want to start arrangement *B* while arrangement *A* is playing, the change could only take place when arrangement *A* has its playhead position at an even 4-second-period. In some cases this will feel like a very long time (up to four seconds, see 3.1.3.2), but the music won’t permit a higher resolution - in this particular composition it would sound awful changing music in the middle of a bar.


Waiting time: > 2 seconds
Arrange B starts playing at musical end.
Arrangement A
Start arrange B request between beat 2 and 3

3.1.3.2 Non-Upbeat sequence example

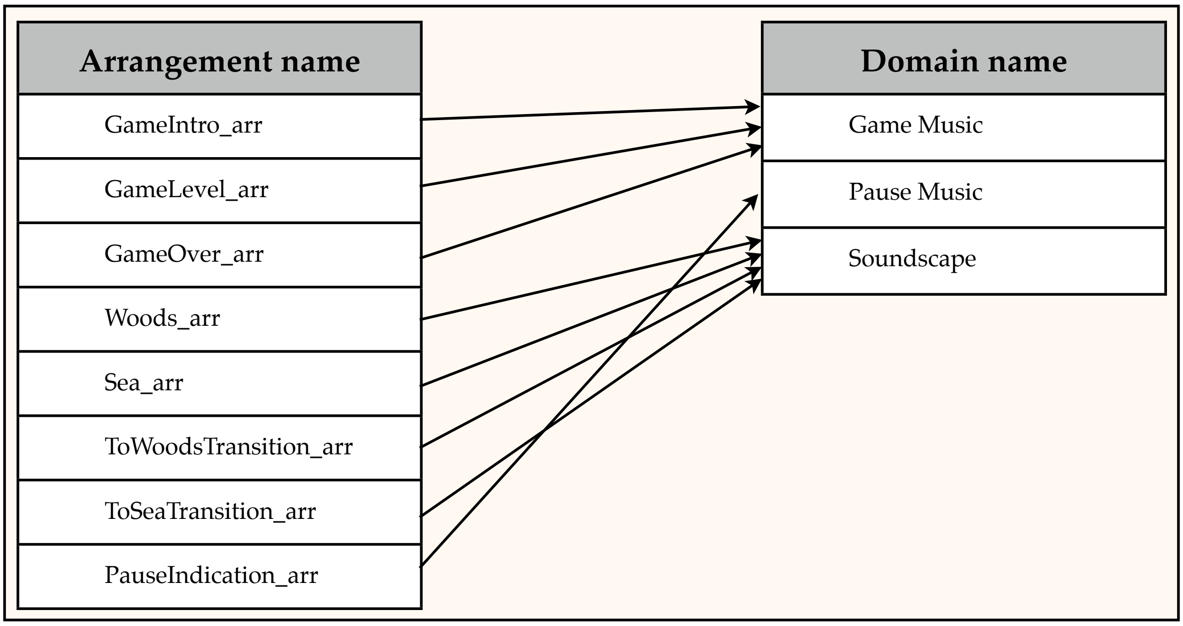

Waiting time: < 1 second
Arrange B starts playing at musical end.
Arrangement A
Start arrange B request between beat 2 and 3
Audio Cue plays the 2-beat upbeat sequence on beat 3 and the end-user gets feedback.

3.1.3.3 Upbeat sequence example

If we create a couple of sequences which have a 1-beat, 2-beat and 3-beat length containing nice stuff like drum fill-ins or similar we could add them to the upbeat list of the arrangement. Audio Cue will choose the proper one when starting the arrangement, i.e. the one that will fit into the time space and play most sudden (see 3.1.3.3). This means the user will have a more instant musical feedback and the music implementation will sound and feel more responsive.

3.1.4 Domain

A domain is the top-level of music playback. It has no parameters more than its name tag and is used to organise the arrangements. Domains cannot be started, but there are methods to pause, resume and stop any playing arrangement in a domain. Each domain is normally representing one song.



3.1.4.1 Relations between arrangements and domains

Arrangements in the same domain cannot be played simultaneously. If an arrangement called *verse* belongs to domain *A* and is started while some other arrangement in the *A* domain is playing, that other arrangement will be cued to stop at next sounds' musical end, and when *that* happens the *verse* arrangement will start.

A typical scenario where arrangements must be divided into two different domains is when an adaptive soundscape (built on sequences and arranges) should be played simultaneously with some other music.

3.2 Sound Objects

Sound Objects are like *sequences* made-up of a list of *sounds* and are playable and routable (should be attached to a *bus* and optionally a *group).* The difference from a *sequence* is that a Sound Object is used to play ordinary effects and doesn't mind musical lengths or other musical synchronisation. When playing a Sound Object, one of the sounds in the list is instantly played and will finish by default. One Sound Object can play several sounds at the same time and won't cancel an on-going sound if a new one is triggered in the same list. A Sound Object can be configured into three different modes:

- **Random trig**
  Plays a random sound in the list, but will never play the same sound twice if there are more than one sound in the list.

- **Step trig**
  Iterates and loops the list on each play. If a Sound Object has a list of three sounds (sound1, sound2, sound3) it will start sound1 on the first play, on the second sound2 etc...

- **Velocity trig**
  Maps the sound list to different velocity layers. The velocity value should have a factor between 0 och 1. If a Sound Object contains ten sounds (sound1 to sound10) and that Sound Object is triggered with a velocity of 0.15, sound2 will play. If the velocity is 1, sound10 will play and so on...

- **Loop**
  This mode only permits one single sound in the list. When started it will play and loop the sound infinitely until it's stopped.

3.3 Audio Routing

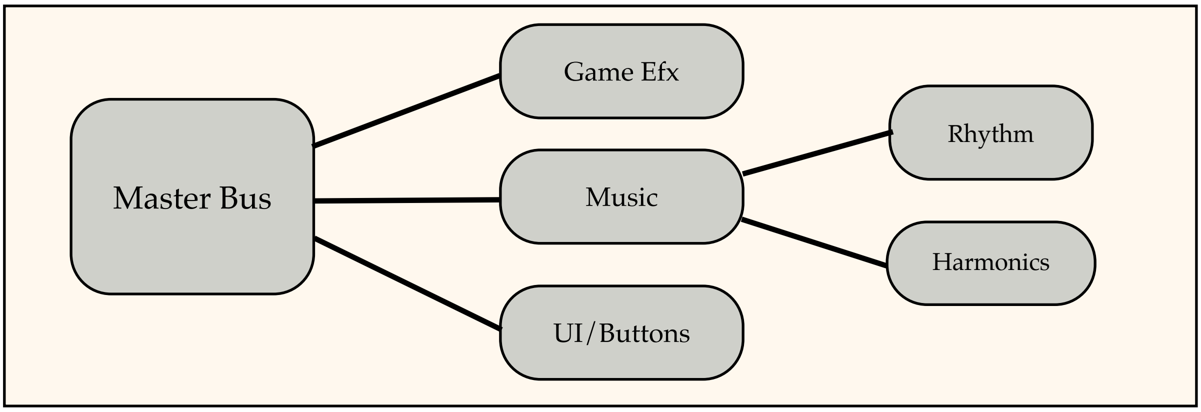
The sounds added to Audio Cue are meant to be pre-mixed and should sound as if they were ready for delivery. The audio routing systems purpose is to ***alter the mix in realtime and not to create a mix from scratch****. The composer must have full control over the "normalised" mix and be able to use his/her favourite mixing tools during the music production process. The Audio Cue editor can not (****and should not!****) compete with other DAW's excellent user interfaces and DSP algorithms.*

Audio Cue provides a small but yet powerful system for realtime mixing. Via *events* and *actions* it's possible to alter the volume and/or an effect parameter on individual tracks. The structure differs from how an ordinary mixing console works; instead of one channel strip with EQ, pan, volume faders, DSP inserts per track, the composer can create busses (for altering volume) and groups (for adding low-cpu-consuming DSP effects) and attach tracks[2] to these. Both a bus and a group can have multiple tracks attached to them.

3.3.1 Busses

Each track *must* be attached to a *bus*. A bus only handles the volume level. This makes it possible to save a lot of the processor's power since there is no need to re-render the audio content for each bus it passes; Audio Cue calculates the final volume level and will only render the proper output once. This means the composer can use a lot of busses without draining cpu usage.

Busses can be nested, which makes it possible to create a master bus and still be able to alter the volume on separate tracks that are attached to a child bus (or a child's child bus). When altering a bus volume via actions the composer also has the choice to define a *fade time*. If a bus' volume originally is set to 0dB and the composer execute a change to -6dB with a time value of 2.5, Audio Cue will perform a smooth realtime fade from 0dB to -6dB in 2.5 seconds.

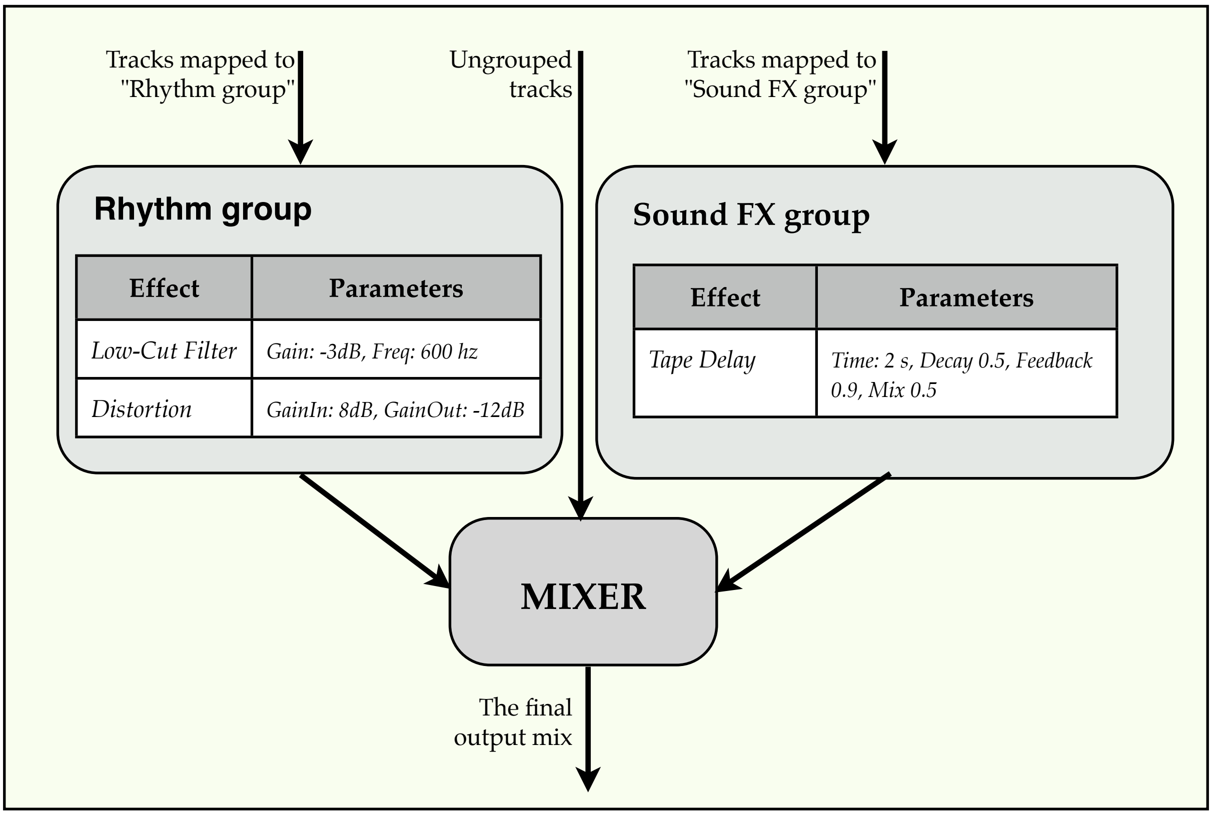


3.3.1.1 Nested bus structure illustration

3.3.2 Groups and audio effects

In addition to the *bus* system it's also possible to attach a track to a *group*. A group contains a list of serially connected realtime DSP effects - much like the insert plug-ins section in a DAW's channel strip - which parameters' (such as a filter's frequency or a flanger's speed) can be altered in realtime. An effect can also be bypassed during runtime to save cpu usage. Anyhow, the internal structure of the audio rendering engine forces the audio data to be re-rendered once for each group. This means the composer should avoid creating a lot of groups with a lot of effects and instead try to include the effects in DAW when bouncing sound assets.

[2] *A track is the equivalent of a routable Audio Cue object from now on.*



Effect	Parameters
Low-Cut Filter	*Gain: -3dB, Freq: 600 hz*
Distortion	*GainIn: 8dB, GainOut: -12dB*

Effect	Parameters
Tape Delay	*Time: 2 s, Decay 0.5, Feedback 0.9, Mix 0.5*

3.3.2.1 Group and effects audio flow illustration

Groups can not be nested with other groups, and it's not currently possible to receive any auxiliary sends from a track or bus to a group; the whole track signal will be running through the group's effects and never split. For compensation, effects like *delay* and *flanger* comes with an "amount" parameter that sets the wet/dry mix internally.

Another technique to alter the amount of delay in realtime without using groups and effects is to bounce one dry and one wet version of the sound and put in two different sequences. Then add these two sequences to the same arrangement and attach them to different busses. Changing the wet sequence's volume will sound exactly the same as modifying the "amount"-parameter of a delay effect instance in Audio Cue. In some cases this is not possible - a filter sweep could not be achieved this way for example - and if that's what needs to be done, using groups and effects are the proper way to go.

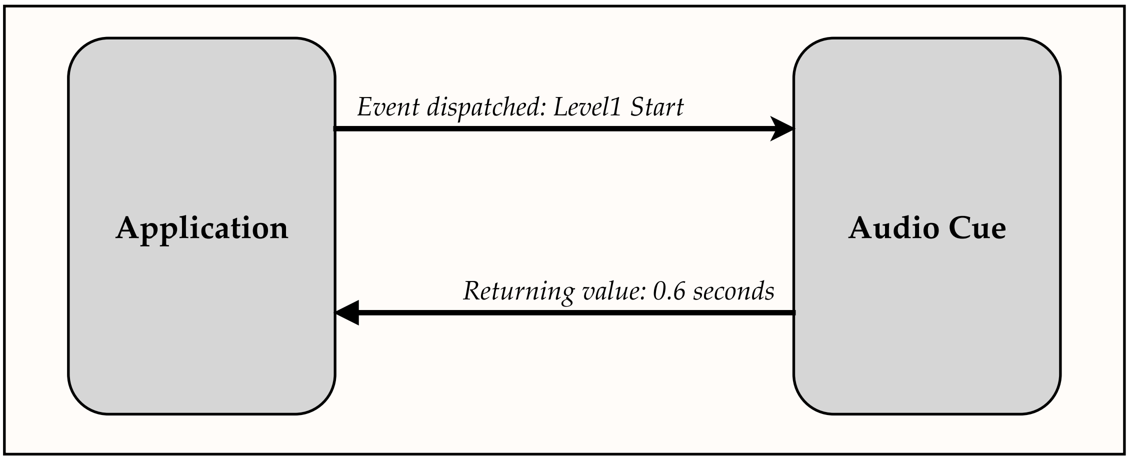
3.4 Events and Actions

This part of Audio Cue is the bridge between the developer and the composer. All the developer has to know about the sound and music structure is what events he/she should dispatch and when they should be dispatched (and in some cases what return values he/she should expect back).

3.4.1 Events

Events are basically simple strings that define application interactions and should therefore be named by what *happens in the application*. It's very handy for the composer to call an event **"Play Verse 1"** when the first level of a game starts, but that will not make any sense for the developer. This way of naming events is also a potential project-chaos creator: if the composer changes the music implementation and decides to start the *refrain* when the "Play Verse" event is dispatched an event called "Play Verse" will start the song's refrain. That's not very good practice.

Instead: use "Level1 Start". That event name will hopefully be valid through the whole production time, the composer will keep the order in the Audio Cue project and the developer will understand what that event actually *means* and *when to dispatch it*.



3.4.1.1 Illustration of an application dispatching an event and gets a timing value back

When the developer dispatches an event, a numerical value will be returned. This value describes how long time there is until the musical change will take effect.

A typical real-life scenario is when the developer dispatches an event on a scene change (let's say from main menu to game intro). When the end-user presses the "start" button, the application starts a graphical transition from the menu scene to the intro scene. Depending on what position the current playing music has, the time until the music actually changes will be different every time the "start" button is pressed. That time is returned by Audio Cue, and the developer can then use that value to synchronise the graphical transition and let the game intro start exactly when the new music part starts. **This will to the end-user sound like the music were exclusively written for someone who is pressing that button at exactly that time.** Since there is immediate graphical response (the transition) that leads to a full response of both graphical and musical adaption (game intro) the end-user will not notice the delay in musical change. This is an example of effective and simple-to-implement musical adaptation!

3.4.2 Actions

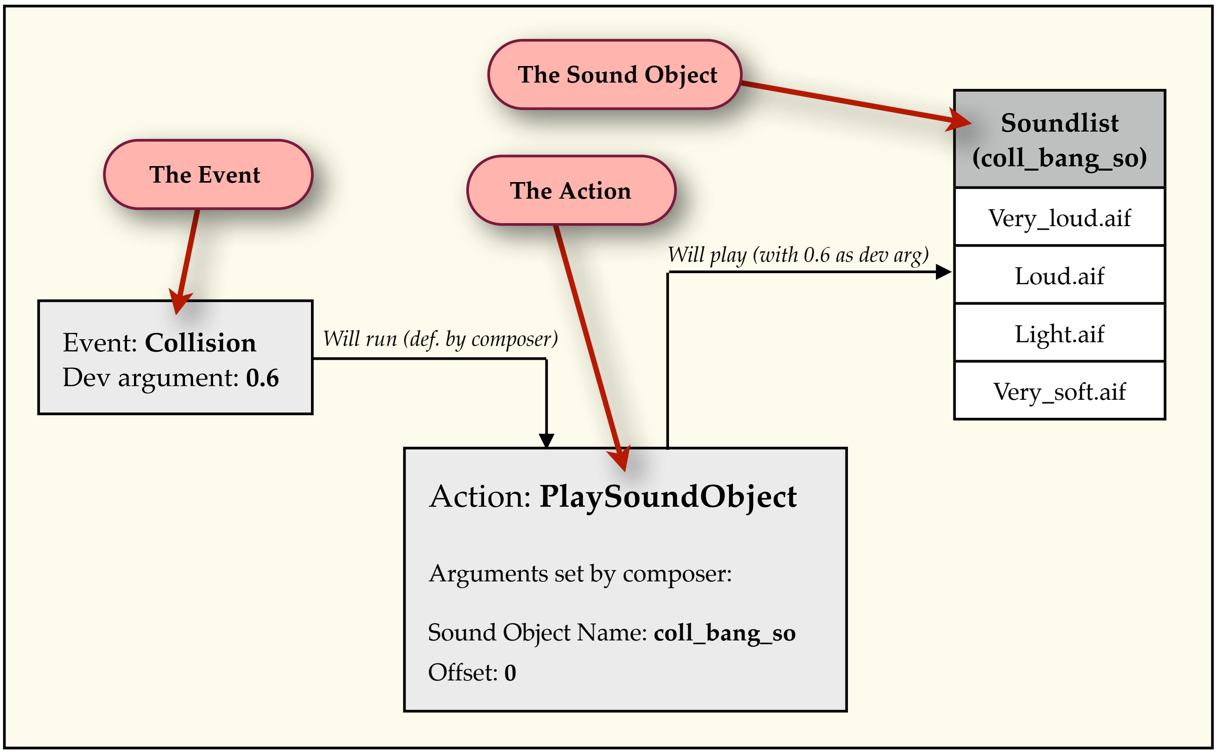
Each event should contain one or more *actions*. Audio Cue comes with a bundle of native actions for controlling the playable objects[3], bus volumes and effect parameters. Each of these actions takes a number of pre-defined arguments, all set by the composer. An argument can be the name of an arrangement or the gain value of a filter effect.

When a developer dispatches an event, he/she can optionally pass an extra argument (called *developer argument*). This feature is supported by some actions, for an example the action of playing a sound object defined as *velocity trig*[4].

A real world example could be when the developer attaches an *event* to a collision between two flying objects in a game; when they hit each other it will be easy (programmatically) to also send a damage variable to Audio Cue. The composer then creates a *sound object* defined as *velocity trig,* fills it up with proper sounds and maps a "Play Sound Object action" to the event. That makes Audio Cue play different sounds depending on the collision damage - if there was a lot of damage; play a loud bang, if the objects barely touched each other; play a softer sound.

[3] *such as arrangements and sound objects*

[4] *See "Sound Objects" chapter 3.2*



3.4.2.1 An event dispatched with developer argument and an attached action illustration

In addition to the bundle of pre-defined native actions, Audio Cue also supports actions as plug-ins. These actions are written in LUA[5] and will, when imported, behave exactly as the native ones in the Audio Cue editor. Plug-in actions could be created for project-specific occasions that won't be possible to solve using the native actions, or just to extend the functionality of how the music should adapt.

[5] *LUA is a commonly used dynamic scripting language. See http://www.lua.org for more details.*