HAL open science

An Interactive Audio System for Mobiles

Yohan Lasorsa, Jacques Lemordant

To cite this version:

Yohan Lasorsa, Jacques Lemordant. An Interactive Audio System for Mobiles. 127th AES Convention, Oct 2009, New York, United States. inria-00425231

HAL Id: inria-00425231

https://inria.hal.science/inria-00425231v1

Submitted on 20 Oct 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Audio Engineering Society

Convention Paper

Presented at the 127th Convention
2009 October 9–12 New York, NY, USA

The papers at this Convention have been selected on the basis of a submitted abstract and extended precis that have been peer reviewed by at least two qualified anonymous reviewers. This convention paper has been reproduced from the author's advance manuscript, without editing, corrections, or consideration by the Review Board. The AES takes no responsibility for the contents. Additional papers may be obtained by sending request and remittance to Audio Engineering Society, 60 East 42nd Street, New York, New York 10165-2520, USA; also see www.aes.org. All rights reserved. Reproduction of this paper, or any portion thereof, is not permitted without direct permission from the Journal of the Audio Engineering Society.

An Interactive Audio System for Mobiles

Yohan Lasorsa[1], Jacques Lemordant[1]

[1] INRIA Rhône-Alpes, France
{yohan.lasorsa, jacques.lemordant}@inria.fr

ABSTRACT

This paper presents an XML format for embedded interactive audio, deriving from well-established formats like iXMF and SMIL. We introduce in this format a new paradigm for audio elements and animations synchronization, using a flexible event-driven system in conjunction with graph description capabilities to replace audio scripting. The concepts of this new format are explained through the building of a virtual interactive jungle environment. Then we have implemented a sound manager for J2ME smartphones and the iPhone. Guidance applications for blind people based on this audio system are being developed.

1. INTRODUCTION

Generating audio by using digital chunks is a way to build a soundtrack, which is extensively used for console or PC games. It is now also possible on mobiles due to the presence of powerful mixers for digital audio. But for that, we need a format for interactive audio. We will show in this paper how an XML format for 3D and interactive audio on mobiles can be designed. This format will be called AMML, standing for *Advanced/Audio Multimedia Markup Language.* The grammar of the format will be defined by a schema, In order to use this format, soundtrack managers have been built for the iPhone and J2ME phones.

2. AN XML INTERACTIVE AUDIO FORMAT

In 1999, the *Interactive Audio Special Interest Group* (IASIG) published an Interactive 3D Audio Rendering Guideline Level 2.0 (I3DL2) [1]. Then in 2005, the Synchronized Multimedia Activity of the World Wide Web Consortium (W3C) designed the *Synchronized Multimedia Integration Language* version 2.0 (SMIL 2.0) [2] for choreographing multimedia presentations where audio, video, text and graphics are combined in real time. More than eight years after the completion of the I3DL2 guidelines, the IASIG announced the completion of a new interactive audio file format to complement I3DL2. This new format, based on the open-standard XMF file format, is called *Interactive XMF* (iXMF) [3]. The goal of the IASIG in designing this format is to put artistic control into the hands of the artists, keep programmers from having to make artistic

decisions, eliminate rework for porting to new platforms, and reduce production time, cost, and stress.

2.1. Why XML?

iXMF is a complex file format with a structured model and scripting. It is a powerful tool, but also a difficult one to implement on mobiles. iXMF is not an XML-based language as SMIL is. The advantage of using an XML format to specify 3D and interactive audio is that many tools exist to transform, edit and adapt XML documents. Moreover, an XML format can be used as a generic design format read by sound managers in Java code for J2ME phones or C code for smartphones. Another advantage is that graphical editors for audio composers can be developed more easily, by using the XML format as a serialisation mechanism.

2.2. Composite documents

With the advent of XML namespaces, it has become common to see multi-namespace XML documents, also called *compound documents*. Content authors use multiple namespaces to segregate content and to validate specific portions of a document against one set of constraints, and other portions against another one. Compound documents are especially useful for audio mobile applications, because localization is a key characteristic of mobility. Information related to localization is always described through the use of XML languages like *Keyhole Markup Language* (KML) [4] or *OpenStreetMap* (OSM) [5] among others. By using an audio language inside a geographical description language we can build sophisticated interactive 3D soundscape, useful in many mobile applications like audio guides and outdoor games for example.

3. INTRODUCING AMML

The description of the concepts of a new language might seem obscure until their application on a concrete example. Thus we have chosen to explain the various aspects of AMML through the progressive building of the Leonard J. Paul's interactive jungle [6] example. The basic idea of this application is to create a continuous soundtrack for an immersive jungle environment of a game, based on short samples of natural and animal sounds reacting to the player's actions. We will see then how the elements of our language can be used to resolve the problems posed in such a situation.

3.1. The audio hierarchy

When starting with any new audio related tool, the most important thing to know is how are organized and mixed the sounds. Our global audio system is divided in a 4-level hierarchy (fig. 1) inspired by iXMF, with some noticeable differences.

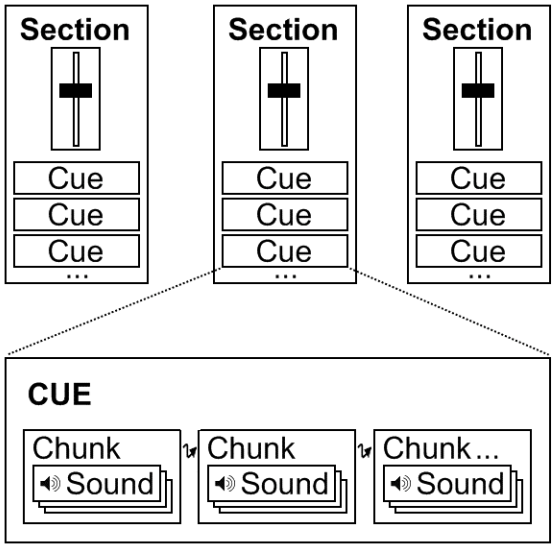

Figure 1 The AMML audio hierarchy

The most basic audio element of our hierarchy is the *sound*: a simple container for an audio file. Then we have the *chunk*, an audio fragment made from one or many sounds, or only a part of it. Each time a chunk is requested to play, if it contains multiple sounds, one is picked based on the rules defined. Sounds can be picked in order or randomly, excluding last played sound or not, and the chance of picking a particular sound over the others can be adjusted. On top of that, the *cue* is an identified container for a sequence of one or more chunks, which can be directly managed by the audio artist to be started and stopped when needed. Cues are the most important elements to the audio artist as they represent the directly playable soundtrack elements. The 3-level audio hierarchy of cues was designed to be simple, effective and adaptable to most situations, while leaving space for creativity. Like in a traditional mixing console, mix groups can be used to regroup multiple cues and apply mix parameters on all of them at the same time. In our format, we called them *sections* as, in addition to mixing multiple cues, they can also be used to add DSP effects and locate the audio in a virtual 3D environment. The main difference with traditional mix groups is that a cue can be a member of multiple

sections, and the effects of all of them will apply, making sections very versatile. We will now see how to take advantage of all these elements to build the base of our jungle soundtrack.

3.1.1. Defining audio cues

First, we need to decide what entities will be part of our virtual jungle. For the natural part, we would like a river and wind ambiance, and for the animal part, crickets, flies, birds and monkeys. To create the river ambiance, we use 6 short samples that we split into 2 chunks:

```
<cue id="river" loopCount="-1">
  <chunk pick="exclusiveRandom"
fadeOutType="crossfade" fadeOutDur="0.5s">
    <sound src="/river_1-1.wav"/>
    <sound src="/river_1-2.wav"/>
    <sound src="/river_1-3.wav"/>
  </chunk>
  <chunk pick="exclusiveRandom"
fadeOutType="crossfade" fadeOutDur="0.5s">
    <sound src="/river_2-1.wav"/>
    <sound src="/river_2-2.wav"/>
    <sound src="/river_2-3.wav"/>
  </chunk>
</cue>
```

The cue is set to loop indefinitely, and will play the chunks one after the other, so to create a seamless transition between them, we set the chunks to use a crossfade transition. We don't want our river ambiance to be repetitive thus each chunk will randomly chose a new sound each time it is played, excluding the previous one. A similar approach is used to define the cue for the wind ambiance. Concerning the animals, the mechanics are quite different as we do not want this time a continuous and seamless ambiance, but randomly triggered and spaced animal sounds of various duration. The cue for the crickets can then be organized with a single chunk containing variations of cricket sounds:

```
<cue id="crickets" dur="rand(3, 10)s">
    <chunk pick="random" fadeInType="simplefade"
fadeInDur="0.5s" fadeOutType="simplefade"
fadeOutDur="0.5s">
      <sound src="/crickets_1.wav" pickPriority="3"/>
      <sound src="/crickets_2.wav" pickPriority="2"/>
      <sound src="/crickets_3.wav" pickPriority="1"/>
      <sound src="/crickets_4.wav" pickPriority="1"/>
    </chunk>
</cue>
```

This time, each time the cue is started, a random cricket sound with duration between 3 and 10 seconds is played. Because we want some sounds to be played more frequently than other, we have changed the picking rules slightly: a sound with a *pick priority* of 3 will have 3 times more chance to be chosen than a sound with a value of 1. You can notice that we are only interested here with the methods to organize audio within our hierarchy, triggering parameters, controls over the resulting sound and instances will be detailed in later sections. As cues for other animals use a similar scheme, let's have a look now on the mixing part of those cues.

3.1.2. Mixing the cues

Once the cues are designed, the next important step it to specify how they are mixed together. The mixing concept used in our format is quite particular, due to the versatility of the sections, the top level elements of our audio hierarchy. A cue can be a member of multiple sections, but how exactly the effects of these sections are applied on a cue? The 2D (volume and panoramic) and 3D mix parameters of a section acts like a macro control on all its cues. When a cue is a member of more than one section (like cue 2 in the figure below) the 2D mix parameters are multiplied, but the 3D mix parameters are overridden by the last section defining them. At this point the cues still have distinct audio channels. Then the "real" mix of these channels occurs when routing the cues to the DSP of the sections. Several DSP inside a section are connected serially. But when a cue is a member of multiple sections, its signal is split and routed to the DSP sets of each section, resulting in a parallel processing (fig. 2).

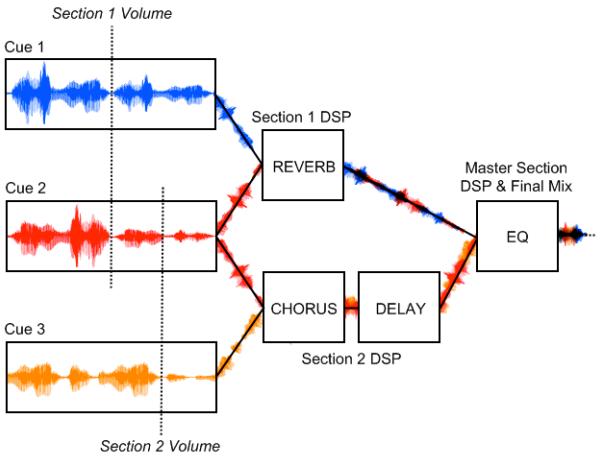

Figure 2 Sections signal processing

Therefore, the flexibility of the sections allows creative and complex audio mix using simple and straightforward definitions. Note that cues which are not members of any section are directly sent to the *master section* for the final mix.

Considering our interactive jungle soundtrack, here is an example configuration for the mix of our audio scene:

```
<masterSection volume="100">
    <dspControl dspName="reverb">
        <parameter name="preset" value="forest"/>
    </dspControl>
</masterSection>

<section id="ambiance" cues="river wind crickets">
    <volumeControl level="60"/>
</section>

<section id="animals" cues="bird monkeys">
    <volumeControl level="100"/>
</section>
```

The configuration in this case is simple, we added a reverb DSP on the master section to add global depth to the soundtrack, and separated the cues in two mix groups: one for the ambient sounds, the other for the animals. However, as the goal is recreating a realistic and immersive soundtrack, what do we need to change in order to add 3D spatialization?

```
<masterSection volume="100">
    <auditor>
        <locationControl location="0 0 0"/>
        <orientationControl orientation="0 0 1"/>
    </auditor>
    [...]
</masterSection>

<section id="river_3D" cues="river">
    <mix3D>
        <locationControl location="-10 0 50"/>
        <macroscopicControl size="2000 100 10000"/>
    </mix3D>
</section>
```

We specified the auditor's location and orientation in the master section, as we want him to be at the center of the scene, looking forward. Then we added a section to specify the 3D mix parameters of the river – its size and location – in our virtual environment. The same thing can be done for the fly cue with an additional *Doppler effect* to increase the perception of movement of the fly. Obviously, it means that the location of our virtual fly needs animation, but the solution for this problem will be seen later.

3.2. Controlling the sound parameters

As you may have noticed, in order to adjust the multiple parameters of the audio sources, DSP and mixers we have defined special elements that we call *audio controls*. For each parameterized source, we have a distinct control element that regroups all the related parameters. For example, the *tempoControl* element is used to adjust the tempo synchronization base of an audio source, the *locationControl* allows to specify the location of the auditor or a sound source in a 3D environment, etc. Audio controls are directly contained by the audio elements – cues, chunks, sounds, sections and master section – on which the effect of those controls is applied.

3.2.1. Audio controls

For our jungle scene, we have only specified the overall audio structure and mix. We will now be interested in the methods to add more variations to the soundtrack, in order to have a more dynamic and somewhat unpredictable result. First, to ease the static structure of the cue/chunk/sound hierarchy a little, we can use the *triggerControl*:

```
<cue id="wind" loopCount="-1">
    <chunk pick="random" fadeOutType="crossfade"
fadeOutDur="0.5s">
        <sound src="/wind_1-1.wav"/>
        <sound src="/wind_1-2.wav"/>
        <sound src="/wind_1-3.wav"/>
        <sound src="/wind_1-4.wav"/>
    </chunk>
    <chunk pick="exclusiveRandom"
fadeOutType="crossfade" fadeOutDur="0.5s">
        <sound src="/wind_strong_1.wav"/>
        <sound src="/wind_strong_2.wav"/>
        <triggerControl chance="10"/>
    </chunk>
</cue>
```

The second chunk has here a 10% chance of being played after the first one, and is used to produce sometimes a stronger wind effect. Moreover, the trigger control can also be used to start automatically and in a regular manner the cues. In that case, it can be useful to also use the *spawnControl* to define a maximum number of instances allowed to play at the same time:

```
<cue id="fly" restartOnRetrigger="never">
    <chunk pick="random" dur="rand(5, 20)s"
fadeInType="simplefade" fadeInDur="2s"
fadeOutType="simplefade" fadeOutDur="2s">
        <sound src="/fly_1.wav" pickPriority="2"
loopCount="-1"/>
        <sound src="/fly_2.wav" pickPriority="1"
loopCount="-1"/>
        <sound src="/fly_3.wav" pickPriority="1"
loopCount="-1"/>
    </chunk>
    <spawnControl maxInstances="1"/>
    <triggerControl chance="10" autoStart="rand(5,
20)s"/>
</cue>
```

The fly cue is here set to be randomly triggered every 5 to 20 seconds interval and a 10% chance to be played each time it is triggered. Only one fly can be heard at a time as set by the spawn control. You can notice the additional *restartOnRetrigger* parameter on the cue that specifies its behavior when a new instance is required to play: with the current value "*never*", a play request on the cue will be effective only if the maximum instances

number has not be reached, else nothing will happen. With the value "*always*", the cue will play from the begin each time it is requested, stopping the currently playing instance if there is one hence only one instance will be used at anytime. Finally, the value "*auto*" will cause the oldest instance of the cue to restart only when the system is running out of available instances for the cue. Spawn control can be defined at every level of the hierarchy, from the sound to the master section, where it allows to define the global maximum number of instances allowed.

3.2.2. Animations

There are many different control elements available in our language allowing a broad range of action, but they also permit more than just defining static values, as every control can be fully animated. Such as some others languages like SMIL or *Scalable Vector Graphics* (SVG) [7], AMML has configurable *animate* elements, allowing control parameters animation. These elements are directly contained inside the audio control element they refer to, and can be used to modify any parameter of this control. Like SVG animations, simple linear transitions or complex animation paths can be defined, leaving a great range of creativity in the hands of the sound designer. It also removes the need to rely on the programmer to animate the values of the audio elements.

Back to our jungle soundtrack, we would like the birds and monkeys sounds to become quiet and more sparse when the player fires a gunshot, then progressively regain their initial volume and frequency after some time. This is a perfect example to illustrate our animation system:

```
<section id="animals" cues="bird monkeys">
    <volumeControl level="100">
        <animate id="quiet_animals_anim"
attribute="level" from="rand(25, 40)" to="100"
dur="rand(10, 20)" fill="remove"/>
    </volumeControl>
</section>
```

The animation added here will alter the mix of the section, by cutting the volume down to value between 25 and 40, and progressively return back to 100 with duration between 10 and 20 seconds. We also need to animate the trigger control of the bird and monkeys cues so they become more sparse:

```
<cue id="monkeys" restartOnRetrigger="never">
    <chunk pick="exclusiveRandom"
fadeOutType="simplefade" fadeOutDur="1s">
        <sound src="/monkeys_1.wav"/>
        <sound src="/monkeys_2.wav"/>
        <sound src="/monkeys_3.wav"/>
        <sound src="/monkeys_4.wav"/>
    </chunk>
    <spawnControl maxInstances="2"/>
    <triggerControl chance="15"
autoStart="rand(0, 10)s">
        <animate id="more_sparse_monkeys_anim"
attribute="chance" dur="rand(10, 20)s"
from="rand(0, 3)" by="rand(0, 5)" fill="remove"/>
    </triggerControl>
</cue>
```

A similar animation is also defined for the bird cue. Using animation is an easy way to create more variations in the soundtrack and adapt it to the current scene situation, by changing the reverb parameters for example. Subsequently, once the whole audio soundtrack has been defined, we need to take care of the synchronization of the directly playable elements, cues and animations.

3.3. Synchronization

When composing an interactive audio soundtrack, a very important step in the design process is to synchronize the audio elements to the visual scene and user input. There is many ways to do this synchronization: time triggers, callbacks, user events… In audio interactive formats like the FMOD Designer [8] format or iXMF, a mix of these methods are used, but agreements between sound designers and programmers are often needed – especially when using callbacks – and little changes in the soundtrack usually need some adaptation on both sides. As we sought after a clear separation between these tasks – programming and sound design – we aimed our language towards a simplified synchronization system for both sides.

3.3.1. Time, tempo and event unification

To reach this goal, we use a system very similar to SMIL, with some additional features. Like SMIL, synchronizable audio elements have *begin* and *end* attributes to specify when they start and stop. These attributes can use a unified time, tempo and event based synchronization, and random expressions can be also be used. By unified, we mean that any of these three synchronization sources can be mixed indifferently by using for example time or tempo based offsets on events, or timed and event based triggers at the same time. In addition, a *quantization* attribute allows restricting determined or undetermined triggers on specified time or tempo interval, so they can always be in sync as wanted by the sound designer. In a case such as an indeterminate music soundtrack being dynamically built, a cue playing a particular melody set

to be started on a player event – so that may happen anytime – can be restricted to start only on the begin of a 4 beats timing, to be in sync with the rest of the music. In our jungle soundtrack, there is no need for complex synchronization, animals and insects cues are automatically triggered so we only need to start the ambience cues at the start of our scene (0 seconds) like that:

```
<cue id="river" begin="0s" ...>
[...]
```

The more interesting part is related to the management of events, as we chose to focus on them in AMML to deal with all sorts of interactions, and to make the bridge between the sound designers and programmers.

3.3.2. An event-driven system

Events are already present in SMIL, in two forms: internal events automatically generated when the state of the multimedia elements changes, and external events. The latter is though limited to some predefined events, like keyboard or mouse input. On the contrary, the FMOD Designer format and iXMF only support custom user events, with for iXMF a static binding to cue events, the different actions occurring then being scripted in these cues. In AMML, we use a mix of all those concepts with some SMIL-like attributes, but with user definable events and a lot more flexibility.
First, external events can be freely defined by the sound designer, just by using them – by their name – explicitly in the document. The programmer's task will then only consist in sending the events to the AMML API which will dispatch the events, with unrecognized events simply being ignored. This the only part of the soundtrack design process that needs interaction with the programmers, as they need to know which events are required to be sent. But the opposite also works: programmers define and throw events throughout the application, and the sound designer picks up only events he needs, ignoring the others. In the jungle example, an external event is sent when the player decides to fire a gunshot:

```
<cue id="shot" begin="player.shot"
restartOnRetrigger="always">
    <chunk pick="random" fadeOutType="simplefade"
fadeOutDur="1s">
        <sound src="/shot_1.wav"/>
        <sound src="/shot_2.wav"/>
        <sound src="/shot_3.wav"/>
    </chunk>
</cue>
```

The cue corresponding to the gunshot will then automatically start when the *player.shot* event is received. Next, the internal events – like *start*, *stop* or *repeat* events – are generated automatically by cues, animations and chunks, using a simple dotted notation based on their ID name. We can also distinguish between global events for an object (any instance of the object will generate the event) and instance events by using the *this* prefix. Using internal events is useful for adding interaction between the audio objects. For example in our jungle scene, we want the animals to become quiet when a shot is fired. We added previously an animation to reduce the volume of their section, but we would also like all animal cues to be stopped when the shot cue is triggered:

```
<cue id="bird" end="shot.start" ...>
[...]
<cue id="monkeys" end="shot.start" ...>
[...]
```

Moreover, event usage is not limited to triggering cues or animations. They can be used as well for many other actions, like selecting the current sound of a chunk, disabling a chunk, etc. As data can be passed along events, animations are also able to directly use this data as animation values:

```
<section id="fly_3D" cues="fly">
    <mix3D>
        <locationControl
enableAutomaticDoppler="true">
            <animate id="fly_3D_anim"
attribute="location" eventValue="fly.move.default"/>
        </locationControl>
    </mix3D>
</section>
```

We added here a new section to add 3D spatialization to the fly cue. Its location is animated from the special data structure – a set of named values – received along the *fly.move* event. Each time this event is received the location of the fly is updated, and an automatic Doppler effect calculated from its movement is applied.

Finally, we chose to use events for managing all interactions in our language, but the spread of events is not unilateral from the application to the soundtrack, as the internal events generated by our system can also be retrieved by the application for both ways interactions.

3.4. Dynamic audio adaptation

Even though we integrated many features to control various aspects of the soundtrack dynamically, everything is still based on a predefined audio object hierarchy. Although it may be flexible enough for a broad range of usages, there are still limitations inherent of this kind of static structure. When composing

complex audio scenes, there is sometimes a need to break the strict logic of the defined hierarchy, depending of certain conditions. That is where you would need scripts in other formats, but instead of pulling and checking data to see if those conditions are met, we chose to use a different approach in our language: you can simply describe what happens in the audio flow when a condition is met, as an event being pushed. This straightforward approach simplifies the task of the audio designer and prevents the need for him to think like a programmer.

3.4.1. Event graphs

As events are used everywhere in our synchronization system and are good indicators of the status of the diverse soundtrack elements, it was a natural extension to use them to replace conditional scripting. Traditionally when using audio scripts, the sound designer has to check the application status or use some variables to determine the changes needed in the soundtrack. But this task should be done by the programmer, not the sound designer. As XML is a declarative language we wanted a declarative way of handling dynamic soundtrack adaptation to the context. Thus, sequences of actions on the soundtrack can then be declared to occur on particular conditions – without worrying about how to check for those conditions – and connected using our graph description system entirely based on events (fig. 3).

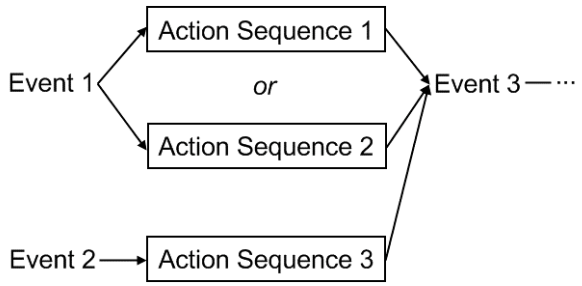

Figure 3 Simple event graph example

Multiple action sequences can be assigned on the same event though only one is picked at a time, in the same manner as sounds are picked inside a chunk: sequentially, randomly or manually (the sequence is set as active by an event). Defining event graphs is very simple, as action sequences allow raising custom events. The remaining question is still: what can we do inside those action sequences? We wanted to avoid adding a further level in the audio hierarchy as the main objective was to relax the existing one, so we added a sort of side level to our audio hierarchy, focused on controlling the audio flow of the soundtrack by modifying the way audio elements should be normally played.

3.4.2. Altering the audio flow

The regular audio flow is made up by the soundtrack manager playing the defined audio elements the way they are supposed to be played, as described by the audio hierarchy. Our idea was to add a way to alter the regular audio flow, by including a sort of second parallel soundtrack manager, dedicated only to do modifications on the arrangement produced. This is where we introduce the *flowControl* elements: triggered the same way as cues or animations, they contain one or many action sequences, only one being picked at a time as multiple sounds inside a chunk. Actions are simple commands aiming at the manipulation of the audio flow: you can mute and unmute cues or sections, start and stop a cue, an animation or another flow control, raise an event, and queue a playing cue to another one. Queuing cue elements is a convenient way to use in a different way the audio hierarchy: the chunk for a playing cue is followed with a chunk of another specified cue like if it was the one supposed to play after – any transition set applying – allowing new degrees of freedom for soundtrack composition (fig. 4).

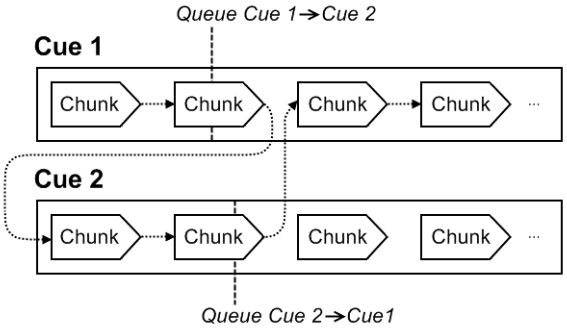

Figure 4 Effect of Queue actions

This action is very useful for example for producing continuous background music that changes depending of the game situation: cues contains the distinct music moods, and flow control elements manages them so only one is playing at a time while they seamlessly fade from one to another in regards of the situation. A simple demonstration for this can be defined in our jungle example, in order to manage the climate ambient sound. In addition to our wind cue, we add a simple cue for simulating a rainy weather:

```
<cue id="rain" loopCount="-1">
    <chunk pick="random" fadeOutType="crossfade"
fadeOutDur="1s">
        <sound src="/rain_1-1.wav"/>
        <sound src="/rain_1-2.wav"/>
        <sound src="/rain_1-3.wav"/>
    </chunk>
</cue>
```

Then we use a flow control to manage the climatic changes:

```
<flowControl id="weather" begin="weather.change">
    <choice pick="ordered">
        <sequence>
            <queue source="rain" target="wind"/>
        </sequence>
        <sequence>
            <queue source="wind" target="rain"/>
        </sequence>
    </choice>
</flowControl>
```

The event *weather.change* sent by the game engine will now trigger each time a change in the climatic ambiance.

Finally, flow controls are simple and creative elements, effective in many applications. However, for the case where scripting is essential and cannot be easily replaced only by a simple flow control – manipulation of DSP parameters based on conditional calculation for example – there is always the solution where the programmer does the scripting job internally in the application and send an event with the needed data, while the sound designer use the event data in an animation (for parameter animation) or just as a trigger.

4. APPLICATIONS

We are actually developing a mobile application for indoor-outdoor guidance of blind people using *Audio Augmented Reality* (Hear-Through AR), which delivers additional sounds through bone conduction, while environmental sounds are still heard through the ear canals. The use of compound documents (OSM and AMML) is of a great help for authoring the audio-geographical database and in its updating through a wiki-web style mechanism. 3D rendering of speech and non-speech audio is of a primary importance as is the possibility to interrupt the audio system through our event mechanism to ask for more information.

5. FUTURE WORK

The use of XML makes evolutions of our format easy, as new features can be added without breaking the compatibility with old versions. We are currently working on a solution to allow AMML data manipulation inside flow controls, to add even more dynamic audio adapatation possibilities. Using more extensively the data attached to events is also under examination. Also, we plan to support additional event triggering using markers inside audio files, like iXMF already does.

Concerning events, the versatility of their usage – especially when using flow control elements – can potentially leads to unwanted behavior like infinite loops if they are not used with care. Cascading event triggers, while allowing complex interactions between internal and external elements, can also easily make a sound designer lose the control of what is occurring on the soundtrack. Unfortunately there is no simple solution for this kind of problems as it is a downside of leaving more freedom in hands of sound designers.

Finally, most efforts regarding the evolution of our language are oriented towards extensive usage of granulation. Currently, only macro grains (audio samples with duration >100ms) are really usable with our soundtrack manager. Though, we would like to support usage of micro grains (duration from 1 to 100ms) and provide functions to make dynamic audio reconstruction more efficient in this particular granulation usage.

6. REFERENCES

[1] Interactive 3D Audio Rendering Guideline Level 2.0, Interactive Audio Special Interest Group, http://www.iasig.org

[2] Synchronized Multimedia Integration Language (SMIL 2.1), W3C, http://www.w3.org/TR/SMIL2/

[3] Interactive XMF, Interactive Audio Special Interest Group, http://www.iasig.org./wg/ixwg/

[4] Keyhole Markup Language (KML), Google, http://code.google.com/intl/en/apis/kml/

[5] OpenStreetMap, http://www.openstreetmap.org

[6] Leonard J. Paul, An introduction to granular synthesis in video games, *From Pac-Man to Pop Music*, Karen Collins (2008).

[7] Scalable Vector Graphics (SVG), W3C, http://www.w3.org/Graphics/SVG/

[8] FMOD, Firelight Technologies Pty, Ltd. http://www.fmod.org