# Procedural Audio for Game using GAF

Olivier Veneri[*] CNAM/CEDRIC
Stéphane Gros[†] CNAM/CEDRIC
Stéphane Natkin[‡] CNAM/CEDRIC

## ABSTRACT

Constantly increasing power of gaming platforms now makes it possible for game developers to consider using procedural techniques in making games. These techniques are actually used to create part of graphical assets such as object's textures [9] or to generate character motion [26]. However, sound is still a challenging domain for procedural content creation. This article presents a new software framework designed to support the use of procedural audio for games. This framework is named GAF (*Game Audio Framework*) and is currently developped by CNAM/CEDRIC laboratory in Paris as part of the PLAY ALL platform. In a first part, we will give a quick overview of current framework architectures. In a second part, we will discuss Procedural Audio. In a third and forth part, we will introduce the new framework proposition we make. We end this article with a demonstration of procedural musical capabilities that this framework enables.

## 1. INTRODUCTION

When defining the audio environment of a game, the main role of a sound designer is to build sound objects. Sound objects are the game entities that are responsible to produce the audio responses to some defined game system state changes. They are the basic building blocks of the audio environment. We can split the sound designer work in three principal activities :

- the definition of audio processes (wave player, synthesizer, etc)
- the definition of a musical behavior that control these audio processes
- the association of these resulting musical structures with game states[34]

[*]e-mail: olivier.veneri@cnam.fr
[†]e-mail: stephane.gros@cnam.fr
[‡]e-mail: snatkin@cnam.fr

Current audio tools provide a finite set of audio components that the sound designer can configure to compose sound objects [1]. In audio frameworks dedicated to game, sound designers are mainly provided with some basic types of audio components (wave player, basic generators and some effects) with some possibilities to insert custom ones, but no esay way to control them. In this tools, the paradigm used is this of sequencers, with some functionnalities of musical trackers (e.g. loops, conditionals, triggers or event management fonctionnalities). To define a sound object the sound designer have to produce audio files for each kind of sounds : voice, foley, ambiance and music and then pack them in some containers that possess predifined musical behaviour[17] [4]. Some standard controls available on these containers are :

- Random behavior on parameters of the containers
- Audio components state control (Start, Stop...)
- Audio components parameters manipulation and automation (crossfading, ...)

After this definition phase the created sound objects will be assigned to game entities during level construction. Sound designers have to bind these musical structures with game states that have to be sonified. To achieve this, nowdays framework provide some basic scripting capabilities to change container states when they receive events from the game or when the state of some shared variable was updated [1].

## 2. PROCEDURAL AUDIO

Procedural audio is the algorithmical approach to the creation of audio content, and is opposed to the current game audio production where sound designers have to create and package static audio content. Beneficits of procedural audio can be numerous : interactivity, scalabiblity, variety, cost, storage and bandwidth. But procedural audio must not be seen as a superior way for creating game audio content. Procedural audio also has drawbacks indeed, like high computationnal cost or unpredictable behaviours. Besides, in some situation, traditionnal content creation can be more efficient.
However some game genres would greatly benefit from such techniques. For example, user-driven content games like the forthcoming games *Spore* (figure 1) or *Little Big Planet*, where the user can customize several assets of the game, it become desirable to provide the player with some very flexible audio/musical assets. The production of such a game

is untenable with static wave based audio content because of the huge quantity of audio assets that would have to be produced in order to meet such a flexibility, due to combinatorial explosion.

The problematics that emerge from the use of procedural

**Figure 1: Spore screenshot**

audio in game is strongly related to state of the computer music researches, which gives a good insight on what could be achieved in the context of video gaming.

We can split procedural audio researches in two main categories :

- Procedural Sound (PS)
- Procedural Music (PM).

On one hand we have PS that bring together the synthesis parts of procedural audio. As made in [11], we list here a few of interesting techniques that can be used for game development. This classification is not based on the underlying model but on the sounds they try to mimic :

- Solid Objects [16] [27] [15] [23]
- Water [32]
- Air [10]
- Fire [35]
- Footstep [25]
- Birds [29]

On the other hand we have PM researches which can lead to the creation of highly dynamic musical behaviours. We also list here a few of PM techniques that we think as potentially useful in a video game context, i.e real-time music compostion or adaptation :

- Stochastic music [5]
- Cellular automata [19]
- Constraint-based composition [30]
- algebraic oriented music composition [21] [13]

We will now introduce GAF (Game Audio Framework), our sound system designed to fulfill all existing game audio standard capabilities presented in the first part of this document but also to extend these functionalities to enable the use of procedural audio. The framework is tailored to allow audio creators to tackle the huge set of creative possibilities offered by the game media and also meet our game audio research needs.

## 3. GAF PRESENTATION

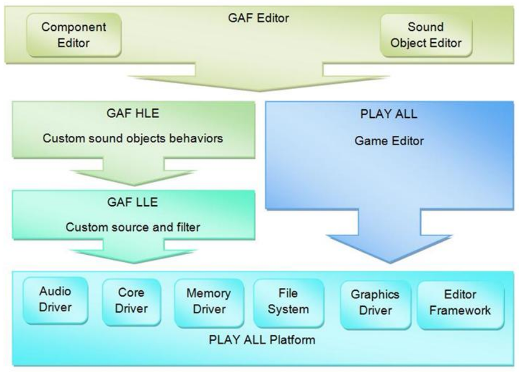

**Figure 2: GAF in the PLAY ALL plateform**

In this section we will describe the engine parts of the framework, that are currently in development for the PLAY ALL platform (figure 2). The engine part of GAF is split in two sub-systems. The first is called the Low Level Engine (LLE) and is focused on the management of all the synchronous computation that occures, i.e synthesis and DSP algorithms. The second sub-system is called the High Level Engine (HLE) and is responsible for handling all the asynchronous computations, i.e event management coming from the game and definition of musical behaviours. In GAF a sound objects is defined as the composition of synchronous and asynchronous computing entities.

### 3.1 GAF Low Level Engine

The Low Level Engine (LLE) has to rely on a standard game audio pipeline [31] with 3D audio capabilities (FMOD, XAudio, OpenAL, MultiStream) supplied here by the PLAY ALL audio driver. GAF LLE focus on the creation of sources and filters to allow the use of custom synthesis and DSP algorithms. GAF users can create custom pipeline objects by definig a typed data flow graph made of connections between processing unit's ports. This data flow graph is called a *Processing Network* in GAF and is similar to the data flow pattern [18] [3] which is intensively used in audio software industry. Current game audio frameworks also have graph capabilities but the main difference lies in the fact that in existent game audio frameworks, graph can only handle one type of data, namely audio data in temporal domain, whereas GAF allows handeling of any data type, as users can define custom types. The introduction of typed port connexion in processing network allows users to create a broader class of synchronous audio processing algorithms that can include frequential domain or any musical data type [3] like midi data for example.

In order to create new processing units, users have to derive the *IProcessingBase* interface and provide this implementation to the framework through the creation of a dynamic library or by statically registering it at the engine initialization. Creating a new processing unit has two uses : the first is obviously to extend the panel of functionalities available in graph creation. The second concerns optimisation. The algorithm described by a graph is splited into isolated processing units, which might not allow fine optimization. Once an audio processing algorithm has been designed, fine-tuned and tested using a graph of existing processing units, it can be useful to set it up as a single processing unit performing the whole algorithm if its opimization matters.

Each *Processing Network* of the LLE can declare global ports that permit state control by the sound objects through the GAFScript language provided by HLE.

## 3.2 GAF High Level Engine

## 3.3 Sound Object behavior

The HLE is used to defined custom sound object behaviours thanks to GAFScript, a scripting language tailored for that purpose.
The development of a domain specific language (DSL) is challenging but this is the only satisfying way to meet our goal i.e. sound designer autonomy and tool flexibility. DSLs exchange genericity for expressiveness in a limited domain, by providing notations and synthax tailored toward a particular application domain, they also offer substantial gains in expressiveness and ease of use compared with general purpose programming languages for the targeted domain.

### 3.3.1 GAFScript Language

GAFScript is a statically typed, stack based scripting language. It is compiled to a bytecode interpreted by the GAFScript virtual machine. The type system is built on the reflection capabilities of the interface *IObject*. It is used to retrieve description of components that implement the *IObject* interface and hence enable their use in GAFScript. This interface is used to implement custom data types that will be usable in the scripting language. This mechanism also meets our high performance intents : thanks to it, GAFScript is only glue code that interface hard-coded C++ objects that perform costly computation.
GAFScript is set up to define custom behaviours for sound objects and especially to allow the use of procedural music. To achieve this, GAFScript brings expressivity of domain specific language by integrating game (like UnrealScript [2]) and music (like Chuck [33]) related semantic [6] :

- Game Communication
- Convenient methods of handling the flow of time
- Concurrency
- Structural organization of musical materials,
- Musical data representation

The communication mechanisms in GAF currently enables a sound object to send events to the game. Existent game system/sound system communication scheme is one way[22], client/server, that is to say that there is no possibility to notify the game system when some specific events occur in sound script. This is not convenient for generative music handeling. Indeed in generative case, we cannot always assert on a precise timing for a definite musical event. For example, if the sound designer wants to use synchresis [7] [1] to strenghten audiovisual relations, it is easier to let the sound object communicate back to the game to achieve synchronizations between the audioviual fluxs.
GAFScript also integrates music related semantic and draws on the [12] statement:

> *Programming languages for music typically go beyond standard general-purpose languages in implementing special musical data types and operations, methods for representing the flow of time, and methods to represent the forms of recursion and parallelism commonly found in music.*

As GAF enables the insertion of custom data type through the *IObject* interface, it can be extended with any suitable musical data type.
Time manipulation is a major novelty of GAFScript in the game audio field. Many authors [24] [28] [33] identify timed based languages for musical applications as a important feature. GAFScript combine the use of a duration type and a special keyword, *next*, to control script execution : We now show how to manipulate time in GAFScript with the definition of a sound object that generate echoed notes. This example is inspired by the discussion in [8] about real-time scheduler.

```
soundobject {

        duration m_EchoDur = 250::ms;
        source m_oSynth = "StringSynth";
        channel m_oChannel;

        event Init()
        {
                m_oChannel.SetSource( m_oSynth );
                m_oChannel.Play();
        }

        event Note1()
        {
                integer iVelocity = 127;
                do
                {
                        m_oSynth.NoteOn( 56, iVelocity );
                        iVelocity -= 1;
                        next = m_EchoDur;
                }while( iVelocity > 0 )
        }

        event Note2()
        {
                integer iVelocity = 127;
                do
                {
                        m_oSynth.NoteOn( 55, iVelocity );
                        iVelocity -= 1;
                        next = m_EchoDur;
```

[1]Synchresis is the forging between something one sees and something one hears

```
            }while( iVelocity > 0 )
        }
}
```

We can see in this example how GAFScript allows the user to control pipeline objects defined in the LLE. In this script the source that is identify by *m_oSynth* is an LLE object.

## 4. SOUND EDITORS

This part gives an insight of the specifications of the tools that are being developed to allow the use of GAF by end-users. The potentials users of the Game Audio Framework Tools are mainly Sound Designers, used to work with some prevalent software, who have to compel with strict time constraints, who have interest in generative, innovative or dynamic sound or music creation.

Making the sound-designers as independent as possible of developers is an aim of the Game Audio Framework, the tools specifications have been set up in this perspective. Once the work of the sound designer has been integrated in a the game environment, on-the-fly editing is much welcome, so sound editors are fully integrated in the game editor framework of the PLAY ALL plateform. GAF is also designed to be used as a tool for research purposes, which presupposes a more heavily iterative process and user-defined data manipulation. In this purposes, the tools have to allow as much modularity as possible and to allow to define new script data types and new processing units.

Two main basic tools are planed for the GAF : one that allows the construction of user-defined fine-tunable processing units, the other that allows the definition of Sound Objects, through the description of both its synchronous and asynchronous behaviours.

### 4.1 Building tunable sound processing units

There are two ways to create a Processing unit : hard coding it as a library or building it as a Processing Network thanks to the Processing Network Editor. The Processing Network Editor allows the user to define a typed data flow graph of formerly defined Processing Units.

The Processing Network Editor is two-sided : one side for Processing Unit definition itself, the other side for interface elaboration. This process is convenient for modularity/reusability and in heavily iterative contexts, as every Processing Unit can provide an adapted and concise access to its parameters that have to be tuned in common usage.

The interface part is detachable : it exists only in editing context and is removed during run-time execution. It is seen as an overlay, that allows to bind inputs and outputs in reading, writing or both between engine entities and interface widgets.

When a parameter (input or output) is bound to a widget for reading, the correspondig data can be read and then displayed by this widget. When in writing, the widget allows to set this value dynamically. With both access, the widget displays and provides control on the parameter value dynamically (i.e. at runtime).

### 4.2 Packing Sound Objects

The sound object editor provides the user with the tools he needs to build a Sound Object from processing networks and script. The sound emitting process is defined by *IChannel*s that are Processing Networks with specific properties (processing networks of *ISource*s and *IFilter*s that are, themeselves, Processing Units with specific properties).

To match current interface paradigm and practice, the *IChannel*s interface looks like a slice and the synchronous behaviour for a sound object is set in a similar way than it would have been on a mixing board. However, the more atypical the Processing Units used, the more complex the interface.

The sonic behaviour of the sound object is defined by a GAFScript. The graphical interface of this sound object is defined by binding the local and global data exposed by the script to some widgets. This features enable the defintion of custom interfaces the same way processing network editor does.

## 5. CASE STUDY : PROCEDURAL MUSIC FOR GAME WITH GAF

In this section we will introduce the use of GAF and especially the GAFScript syntax through an example. In this example we will demonstrate how it is possible in GAF to define generative musical behaviour. We chose to base the musical behaviour on a stochastic approach. The use of stochastic model in music has been studied for a long time now [14] [20] but has almost never really been used in game. Stochastic music can have several advantages. One of the main advantage is that it allows sound designers to insert very flexible and tunable content inside the game. Another advantage of probabilistic approach is that it decreases the feeling of repetition of listener without having to produce a lot of musical data compared to a wave file based approach. Our prototype is based on a simple scene in which the user can move the camera around a character and listen to the musical changes that occurs.

### 5.1 Musical Behaviors

The background music is made of two instruments : a string synthesizer and drum unit. Each one as its own musical processes that consist of a melodic (or drum elements sequence) process and a rythmic process. The melodic part of the string synthesizer is generated by two first order markov matrix and its rythmic part by a set of probability vectors. The markov matrices describe how to get from one melodic note to another and the probability vectors the rythm to apply. The sequence process and rythm process of the drum kit are entirely defined by its probility vectors. We have two sets of data for each instrument : one for a closest camera position and another for a farthest camera position.

The probability vectors are used to define rythmic behaviour by determining the onset of the rythmic event. This means that each note is sustained until the next event onset. The components of the vectors represent the probability for one event (or silence) to be played at each quantification step (sixteenth note) inside a four measure grid. There is one probability vector for each quantification step (1). The vectors related to the string synthesizer have only two components, one for enabling note playing and one for silence. The drum vectors have five components as they are also used to choose which drum element to play. The data for matrices and probability vectors have not been gathered from previous musical analysis but set then adjusted after successive tests and trials of the sound designer.

| quantification step | drum 1 | drum 2 | drum 3 | silence |
|---|---|---|---|---|
| 1/16 | 0.5 | 0.2 | 0.0 | 0.3 |
| 2/16 | 0.5 | 0.0 | 0. | 0.5 |
| 3/16 | 0.0 | 0.0 | 0.0 | 0.0 |
| 4/16 | 0.0 | 0.0 | 0.0 | 0.0 |
| 5/16 | 0.5 | 0.2 | 0.0 | 0.3 |
| 6/16 | 0.5 | 0.0 | 0. | 0.5 |
| 7/16 | 0.0 | 0.0 | 0.0 | 0.0 |
| 8/16 | 0.0 | 0.0 | 0.0 | 0.0 |
| 9/16 | 0.5 | 0.2 | 0.0 | 0.3 |
| 10/16 | 0.5 | 0.0 | 0. | 0.5 |
| 11/16 | 0.0 | 0.0 | 0.0 | 0.0 |
| 12/16 | 0.0 | 0.0 | 0.0 | 0.0 |
| 13/16 | 0.5 | 0.2 | 0.0 | 0.3 |
| 14/16 | 0.5 | 0.0 | 0. | 0.5 |
| 15/16 | 0.0 | 0.0 | 0.0 | 0.0 |
| 16/16 | 0.0 | 0.0 | 0.0 | 0.0 |

**Table 1: Array of probability vector of drum unit for the farthest camera position**

| quantification step | drum 1 | drum 2 | drum 3 | silence |
|---|---|---|---|---|
| 1/16 | 0.125 | 0.525 | 0.25 | 0.0 |
| 2/16 | 0.5 | 0.125 | 0.375 | 0.0 |
| 3/16 | 0.1 | 0.7 | 0.2 | 0.0 |
| 4/16 | 0.3 | 0.2 | 0.5 | 0.0 |
| 5/16 | 0.15 | 0.8 | 0.05 | 0.0 |
| 6/16 | 0.01 | 0.09 | 0.9 | 0.0 |
| 7/16 | 0.1 | 0.1 | 0.8 | 0.0 |
| 8/16 | 0.25 | 0.5 | 0.25 | 0.0 |
| 9/16 | 0.125 | 0.525 | 0.25 | 0.0 |
| 10/16 | 0.5 | 0.125 | 0.375 | 0.0 |
| 11/16 | 0.1 | 0.7 | 0.2 | 0.0 |
| 12/16 | 0.3 | 0.2 | 0.5 | 0.0 |
| 13/16 | 0.15 | 0.8 | 0.05 | 0.0 |
| 14/16 | 0.01 | 0.09 | 0.9 | 0.0 |
| 15/16 | 0.1 | 0.1 | 0.8 | 0.0 |
| 16/16 | 0.25 | 0.5 | 0.25 | 0.0 |

**Table 2: Array of probability vector of of drum unit for the closest camera position**

When the camera moves back and forth toward the character, we hear a continous modification of the musical flux. This modification affect both the melodic and the rythm parts of the two instruments. This is realized through a linear interpolation of the component values of their farthest and closest melodic matrices and their farthest and closest rythmic vectors, with the distance between the camera and the character as the interpolating parameter.

What we want to show here is that it is now possible and interesting to use procedural techniques in the creation of audio environments. Here, we jointly use a probabilitic model for music generation and sound synthesis techniques in order to achieve better player immersion but also avoid too high production cost due to the static creation of highly dynamic content. Indeed, the production of such a flexible scene only by means of wave segments would require a lot of short wave files and then specification of their succession. This process can be long, tedious and costly especially in the very iterative context of game production.

## 5.2 GAFScript Overview

We do not entirely describe the script used in this experiment du to length limitation but we will extract small parts of it to exemplify how it is used and how the GAFScript syntax looks like.

### *5.2.1 Game Communication*

In our example, we use the distance between the camera and the character as the parameter of our musical model, so we need a variable for it. The sharing of variable between the sound object and the game system is defined by prefixing the desired variable with the *shared* keyword.

```
soundobject sandbox08 {

        // Data shared with game
        shared float    m_fDistance     = 1.0;

...

        }
```

**Figure 3: Shared variable definition**

We also need three events :

- The *Init* event for initialisation stuff
- The *StartMusic* for stating the synthesizer process
- The *StartDrum* for starting the drum process

The events are declared simply by defining a method for the sound object inside the script ( figure **??**). This variable and these events can then be used inside the game code using a C++ generated header file created during script complation and that contain all the necessary informations (figure 4)to call events or to set shared variables.

The first thing to be done in GAFScript is the declaration of the various data that will be needed to define the musical behaviour of the currently constructed sound object (figure **??**). There are specific keywords to declare pipeline objects : *source* , *filter* and *channel.*

```
...

        // Pipeline objects
        source m_oSynth       =       "StringSynth.gpo";
        source m_oDrum        =       "Takim.gpo";
        channel m_oChannel;
        channel m_oDrumChannel;

...
```

```
namespace sandbox08 {

        namespace INPUT_EVENT {

        const PlayAll::uint32 Init = 0 ;
        const PlayAll::uint32 StartDrum = 2 ;
        const PlayAll::uint32 StartMusic = 1 ;
        }

        namespace SHARED_VARS {

        const PlayAll::uint32 m_fDistance = 18 ;

        }

}
```

**Figure 4: Generated C++ file.**

Then the musical data and objects can be declared in order to be used in the subsequent parts of the script. Each data type is build by deriving the *IObject* so that each script object makes direct call to C++ code to be as efficient as possible. All the needed memory is allocated at compilation time so that no memory allocation occurs during run-time.

```
        // Data for music generation
        integer                     m_IntResult;
        MarkovInterpolator          m_oMarkovInterp;
        ProbVecInterpolator         m_oVecInterpolator;
        ProbVector                  m_oProbVec1;
        ProbVector                  m_oProbVec2;
        ProbVector                  m_oVecResult;
        MarkovMatrix< integer > m_oMM1;
        MarkovMatrix< integer > m_oMM2;
        MarkovMatrix< integer > m_oMMResult;

...
```

**Figure 5: Musical data declaration**

The definition of the sound object response (figure ??) is made by declaring and defining an *event*. The instructions in the scope of the event *StartMusic* are executed when the sound object receives this event. Each method is a separated thread on the GAFScript virtual machine so they can be used to define concurrent musical processes.

```
...

event StartMusic()
{
        //Set matrix input vector
        m_oMMResult.SetInput(0);

        while(true)
        {
                //Interpolate between markov matrices with cam
                //to character distance as parameter
                m_oMarkovInterp.Interpolate( m_oMM1,
                                             m_oMM2,
                                             m_fDistance,
                                             m_oMMResult);
                m_oVecInterpolator.Interpolate( m_oProbVec1,
                                             m_oProbVec2,
                                             m_fDistance,
                                             m_oVecResult);
                //Choose next midi note accordingly to past events
                m_IntResult = m_oMMResult.ChooseNext() ;
                m_IntResult = m_IntResult 40;
                m_oMidiResult.NoteOn( m_IntResult, 90 );
                m_oSynth.SetMidiIn( m_oMidiResult );
                next = m_oVecResult.ChooseNext();
                m_oMidiResult.NoteOff( m_IntResult, 127 );
                m_oSynth.SetMidiIn( m_oMidiResult );
        }
}

...
```

## 6. CONCLUSION

We demonstrate here that the generative approach can be very interesting when sound designer need some continous mapping between the action that takes place and the musical flux. But this is just a first emprical step and there is a need to conduct further experiments and analysis on player satisfaction. We also think that it will be easier for sound designer to set data of the generating elements in a simpler way. For example by providing a few example by some midi segments that will be used to extract data to feed the matrices. We also intuitively think that the realization of a prototype with more musical processes can achieve better immersion result. Actually, we only genererate and adapt two musical parameters, namely rythm and pitch, with only one model parameter, the camera to character distance. We expect a more interesting result with more complex scene and more complex gameplay. An implemantation of GAF is currently realized as part of the PLAY All middleware project. PLAY ALL use GAF LLE and HLE as part of its engine and the sound object editors as PLAY ALL level editor plugin for enabling sound integration. This project is leaded by a consortium of french game developers (Darkworks, Kylotonn, White Birds Productions et Wizarbox) and french research laboratories such as CNAM/CEDRIC, ENST, LIP6 and LIRIS.
We plan to use this gaming plateform as a base to develop our research in CEDRIC (CNAM Computer Science Laboratory, FRANCE) on sound in video game and other interactive media and also for our game audio teachings in ENJMIN (Graduate School of Games and Interactive Media, FRANCE).

## 7. REFERENCES

[1] Interactive xmf specification. IASIG: The Interactive Audio Special Interest Group, February 2007.

[2] The unreal engine documentation site, 2007.

[3] X. Amatriain. *An Object-Oriented Metamodel for Digital Signal Processing with a focus on Audio and Music*. PhD thesis, Pompeu Fabra University, 2004.

[4] Audiokinetic. *Wwise Sound Engine SDK*, 2006.

[5] J. Beran. *Statistics in Musicology*. Chapman & Hall/CRC, 2003.

[6] S. N. Cecile Le Prado. Listen lisboa: Scripting languages for interactive musical installations. In *SMC07 Proceeding*, 2007.

[7] M. Chion. *L'audio-Vision : Son et image au cinéma*. Armand Colin, 1994.

[8] R. Dannenberg. *Current Research in Computer Music*, chapter Real-Time Scheduling and Computer Accompaniment. MIT Press, 1989.

[9] D. P. K. P. S. W. David S. Ebert, F. Kenton Musgrav. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann, 1998.

[10] D. Y. et al. Real-time rendering of aerodynamic sound using sound textures based on computational fluid dynamics. *ACM TOG 2003*, Volume 22(Issue 3):732–740, 2003.

[11] N. Fournel.

[12] C. A. Gareth Loy. Programming languages for computer music synthesis, performance, and composition. *ACM Computing Surveys*, 17:235–265, 1985.

[13] M. Guerino. *The Topos of Music : Geometric Logic of Concepts, Theory, and Performance*. Birkhäuser Verlag, Basel, 2002.

[14] X. Iannis. Musiques formelles. *La revue musicale*, volumes 253 et 254:232 pp, 1963.

[15] D. L. James, J. Barbič, and D. K. Pai. Precomputed acoustic transfer: Output-sensitive, accurate sound generation for geometrically complex vibration sources. *ACM Transactions on Graphics (SIGGRAPH 2006)*, 25(3), Aug. 2006.

[16] K. P. G. Kees van den Doel and P. D. K. Foley automatic : Physically-based sound effects for interactive simulation and animation. In *SIGGRAPH 2001*, page 7. ACM Press, 2001.

[17] C. Labs. *IPS Tutorial*, 2005.

[18] D. Manolescu. A data flow pattern language, 1997.

[19] E. R. Miranda. *Computer Sound Design : Synthesis Techniques and Programming*. Focal Press, 2002.

[20] J. A. Moorer. Music and computer composition. *Commun. ACM*, 15(2):104–113, 1972.

[21] A. Moreno. *Méthodes algébriques en musique et musicologie du XXe siècle : aspects théoriques, analytiques et compositionnels*. PhD thesis, EHESS, 2003.

[22] C. L. P. M. E. O. Veneri, S. Natkin. A game audio technology overview. In *Proceedings of the Sound and Music Computing Conference*, 2006.

[23] C. M. G. O'Brien James F., Chen SHhen. Synthesizing sounds from rigid-body simulations. pages 175–181, San Antonio, Texas,, 2002. ACM Press.

[24] F. Pachet, G. Ramalho, and J. Carrive. Representing temporal musical objects and reasoning in the MusES system. *Journal of New Music Research*, 5(3):252–275, 1996.

[25] perry R. Cook. Modeling bill's gait: Analysis and parametric synthesis of walking sounds. In *Proceedings Audio Engineering Society 22 Conference on Virtual, Synthetic and Entertainment Audio*, 2002.

[26] C. Reynolds. Big fast crowds on ps3. In *Proceedings of the 2006 Sandbox Symposium*, 2006.

[27] J. E. L. Richard Corbett, Kees van den Doel and W. Heidrich. Timbrefields : 3d interactive sound models for real-time audio. *Presence: Teleoperators and Virtual Environments*, to be published, 2007.

[28] C. Rueda and F. D. Valencia. Formalizing timed musical processes with a temporal concurrent constraint programming calculus.

[29] T. Smyth and J. III. The sounds of the avian syrinx– are the really flute-like? In *Proceedings of DAFX 2002, International Conference on Digital Audio Effects*, 2002.

[30] C. P. Truchet C. Musical constraint satisfaction problems solved with adaptive search. *Soft Computing*, vol 8(nř9), 2004.

[31] N. Tsingos, E. Gallo, and G. Drettakis. Perceptual audio rendering of complex virtual environments, 2003.

[32] K. van den Doel. Physically-based models for liquid sounds. *ACM Transactions on Applied Perception*, 2:534–54, 2005.

[33] G. Wang and P. R. Cook. Chuck: A concurrent, on-the-fly, audio programming language. In *ICMC 2003*, 2003.

[34] Z. Whalen. Play along : An approach to videogame music, november 2004.

[35] T. Y. Yoshinori Dobashi. Synthesizing sound from turbulent field using sound textures for interactive fluid simulation. In *EUROGRAPHICS 2004*, 2004.