The SpatDIF library – Concepts and Practical Applications in Audio Software

Jan C. Schacher
Zurich University of the Arts
Institute for Computer Music
and Sound Technology ICST
jan.schacher@zhdk.ch

Chikashi Miyama
University of Music, Cologne
Studio for Electronic Music
me@chikashi.net

Trond Lossius
Bergen Center for Electronic Arts BEK
trond.lossius@bek.no

ABSTRACT

The development of SpatDIF, the Spatial Sound Description Interchange Format, continues with the implementation of concrete software tools. In order to make SpatDIF usable in audio workflows, two types of code implementations are developed. The first is the C/C++ software library 'libspatdif', whose purpose is to provide a reference implementation of SpatDIF. The class structure of this library and its main components embodies the principles derived from the concepts and specification of SpatDIF. The second type of tool are specific implementations in audio programming environments, which demonstrate the methods and best-use practices for working with SpatDIF. Two practical scenarios demonstrates the use of an external in MaxMSP and Pure Data as well as the implementation of the same example in a C++ environment. A short-term goal is the complete implementation of the existing specification within the library. A long-term perspective is to develop additional extensions that will further increase the utility of the SpatDIF format.

1 Introduction

The Spatial Sound Description Interchange Format (SpatDIF) presents a structured syntax for describing spatial audio information, addressing the different tasks involved in creating and performing spatial sound scenes. The goal of this approach is to simplify and enhance the methods of creating spatial sound content and to enable the exchange of scene descriptions between otherwise incompatible software. SpatDIF proposes a simple and extensible format as well as best-practice examples for storing and transmitting information about spatial sound scenes. It encourages portability and the exchange of compositions between venues with different surround audio infrastructures. SpatDIF also fosters collaboration between artists such as composers, musicians, sound installation artists as well as researchers in the fields of acoustics, musicology, and sound engineering. SpatDIF was developed in a collaborative effort and has evolved over a number of years.

The completion of a first usable version of the specification [9] defining the core descriptors and a few indispensable additional descriptors was achieved in 2012 and is published in the Computer Music Journal [8]. The community pages as well as all the related information can be found at: http://www.spatdif.org. [1]

The SpatDIF specification was informally presented to the spatial sound community at the ICMC in Huddersfield in August 2011 and at a workshop at the TU-Berlin in September 2011. The responses in these meetings suggested the urgent need for a lightweight and easy to implement spatial sound scene standard, which could contrast the complex MPEG-4 scene description specification [12]. In addition, several functions necessary to make this lightweight standard become functional, such as the capability of dealing with temporal interpolation of scene descriptors as described, were introduced. For a complete overview of the state-of-the art in audio spatialisation tools, please refer to the 2013 article in Computer Music Journal [8], which also functions as a sort of white paper for the specifications 0.3 [9].

Since then, one mayor development in surround audio workflows has been the introduction of the proprietary Dolby Atmos format, which mixes concepts such as sound-beds and channel-based traditional panning with object based real-time panning. Dolby Atmos authoring is achieved using ProTools and the Dolby Rendering and Mastering Unit (RMU). RMU provides the rendering engine for the mix stage, and integrates with Pro Tools through the Dolby Atmos Panner plug-in over Ethernet for metadata communication and monitoring. The metadata is stored in the Pro Tools session as plug-in automation [2]. Dolby Atmos was initially developed for cinema, and more recently consumer appliances have been announced as well.

Finally, one toolset deserves mention because it resembles in many ways what the development process described in this paper is aiming at. The SoundScape Renderer by Geier et al. [4] and its XML-based storage format ASDF [3] were developed in the opposite direction, going from concrete software-implementations to format definitions. This has as a consequence that some of the ASDF-descriptors are implementation-driven, which makes it less portable than SpatDIF aspires to be.

1.1 SpatDIF Basics

Since SpatDIF is a syntax rather than a programming interface or file-format, it may be represented in any of the cur-

Copyright: ©2014 Jan C. Schacher et al. This is an open-access article distributed under the terms of the Creative Commons Attribution 3.0 Unported License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

[1] All URIs in this article were last accessed in April 2014.

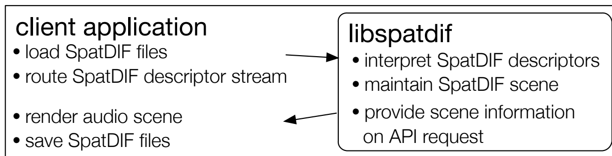

Figure 1: Tasks and Data flow between client application and the SpatDIF library

rent or future structured mark-up languages or messaging systems. It describes the aspects required for the storage and transmission of *scene descriptions*. Because a complete work typically contains aspects that are outside the realm of spatial sound scenes, SpatDIF provides descriptors to link these aspects to the spatial dimensions.

A central principle for SpatDIF is the separation of authoring and rendering of spatial sound scenes or pieces. These processes may use the same or different infrastructure. They may occur at separate times or at separate places. They may be executed either simultaneously or with a long time between the two, and the may finally combine all of these factors in a specific way. The exact modality of these processes should not have to be determined at the outset.

In addition, two principal use-cases can be distinguished. The first scenario is focusing on storing spatial audio scene descriptions for future playback. The second scenario deals with streamed audio content and scene description information in real-time and quasi real-time. For these applications SpatDIF formulates a concise semantic structure that is capable of carrying all the information relevant for preserving a sound scene, without being tied to a specific implementation or technical method.

2 Library Concepts

After establishing a coherent specification with example use-cases in textual form only, the next development step is the implementation of software, which embodies the specified concepts and should serve as a reference for future work.

In this article we present the development and implementation of software tools aimed at easy integration of SpatDIF into existing software and workflows. The concepts and guidelines laid down in the SpatDIF specification are implemented in a platform-independent software library written in C/C++ [5]. The library is in charge of holding one or more spatial audio scenes and provides ways to read and write elements to and from this scenes, either directly from native code or via OSC-formatted messages, that may originate from within the application or arrive from an external source via the network. By providing a software library rather than a complete software application, implementations in many different software environments are facilitated, which is one of the goals of the project.

In section 3.1 the application of the library will be demonstrated in an external for MaxMSP [2] and PureData [3] as well as in an application written entirely in C++ in OpenFrameworks. [4]

2.1 Library Tasks

In order to facilitate implementations in many different environments without making any assumptions about their capabilities, the types of tasks given to the library are carefully selected. There is a deliberate division of labour between the library and the client application (see Fig. 1).

On the one hand, the library builds and maintains in memory the SpatDIF scene, either obtained from an already existing description stored in a file, or on-the-fly in real time from elements received via OSC-formatted commands or native code calls. It provides an application programming interface (API) for accessing the scene that hides most of the complexity of handling the scene data. Through this interface all the information is queried or written.

The client application, on the other hand, is in charge of connecting to the file-system, managing the networking interfaces as well as running the audio-system. All time-based operations are done in the client-application, since they may be driven by audio-rate, a control-rate scheduler or even an external sync source. The client application deals with all audio-related processes, such as loading audio files, playing them back, configuring the audio-system, and rendering the audio to generate an immersive experience.

2.2 Library Class Structure

The class diagram of the SpatDIF software library (see Fig. 2) illustrates the relationship between the scene and its contents, as well as their hierarchical dependencies. An instance of *sdScene* class represents a SpatDIF scene and maintains instances of *sdEntityCore*. Core classes cover the elements mandated by SpatDF while extension classes introduce additional and optional descriptions. The functionalities of *sdEntityCore* may thus be extended by the descendants of *sdEntityExtension*. The activation and deactivation of the extensions is managed globally within a scene, therefore *sdScene* is responsible for all extension handling. Each instance of *sdEntityCore* maintains instances of *sdEvent*, which represent all the events of the entity as they unfold within the scene over time.

The most important classes are described in the following section.

sdScene

An instance of *sdScene* maintains all data associated with a SpatDIF scene. This class offers clients the addition, deletion and modification of *entities* in the scene, the addition and modification of the *meta data* associated with the scene, and finally the activation and deactivation of *extensions* in the scene.

[2] www.cycling74.com
[3] puredata.info
[4] www.openframeworks.cc

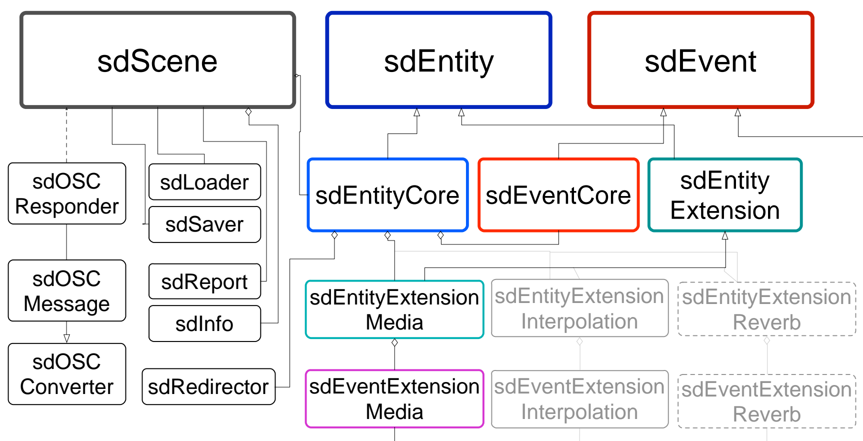

Figure 2: Simplified class hierarchy of the SpatDIF software library. The top row shows the three main classes, below are the derived subclasses coloured according to their parents. The interfacing and utility classes are located on the left and future extensions on the right are marked in grey.

Once the client activates an extension in a scene, *sdScene* automatically adds extended functionalities and allocates extra memory to all existing and newly created instances of *sdEntityCore*. Symmetrically, when deactivating an extension, *sdScene* removes all extended functionalities and previously allocated memory from all existing *sdEntitieCores*, leading to the deletion of all data stored in the extensions.

sdEntity

This class defines entities in SpatDIF scenes as a pure abstract class. It implements basic functionalities, such as addition, deletion, and modification of events.

sdEvent

This is a pure abstract class holding events, storing the absolute *time* of the event, a *descriptor* of the type of event, and the actual data as a *value*.

sdEntityCore

An instance of *sdEntityCore* maintains all events belonging to that entity and a vector storing instances of SpatDIF extensions. This class replies to queries from the client about events. The client is able to query about multiple events within a certain time frame and filter events by descriptors.

sdEventCore

Each instance of *sdEventCore* maintains one SpatDIF core event, consisting of the time of the event, a SpatDIF core descriptor, and associated value(s). Any event in the scene that is tied to a core descriptor is stored in an *sdEntityCore*.

sdEntityExtension

This is a pure abstract class of extensions, and the descendants of this class. e.g., *sdEntityExtensionMedia*, handle the events with extended descriptors. If a client activates an extension in a scene, each existing instance of *sdEntityCore* instantiates the designated subclass of *sdEntityExtension* and registers it.

sdLoader/sdSaver

These two utility classes enable clients to create or store an instance of *sdScene* to or from a XML string. In order to maintain platform independence and to achieve maximum flexibility, the library does not handle files directly, the client software is responsible for the file management. At the time of this writing the functions use the external library TinyXML-2 [5] for parsing of markup formatted strings.

3 Practical Implementations

The SpatDIF syntax is an implementation-independent specification. However, the actual value of using it only becomes evident in real applications. Although SpatDIF was developed with a number of different scenarios in mind, the use-case most closely associated with the authors' practices are electro-acoustic surround audio compositions for concerts and installations or real-time spatialisation in computer music performances. Therefore, the first code implementations of the SpatDIF library are made with tools for real-time audio software.

In order to explore the methods and actual handling of the 'libspatdif' in a real situation, a dual testbed was implemented as an external for both MaxMSP and Pure Data, named 'spatdif'. Apart from small differences in the two environments the two implementations are identical. In addition, an example application with a limited feature-set was written in openFrameworks, in order to establish and test a workflow done entirely in the C++ language.

There are a number of concepts that need to be taken into account when using the library, informing the design of the implementations shown here. The library serves as a data-storage for audio scenes that needs to be queried for its information in specific ways. It does not provide a scheduling mechanism of its own, rather, the client application is responsible for executing all time-related functions. This design choice is explicitly geared towards temporal flexi-

[5] www.grinninglizard.com/tinyxml

bility, e.g., slowing down or speeding up playback, jumping and cueing, which are features that can only properly be implemented by the client application.

There are two main interfaces for the library, the native command and the OSC-command, as will be discussed in section 3.3. The native commands call functions of the library within C/C++ code whereas the OSC-commands get handed to the library as messages conforming to the OSC syntax. In addition there is a wrapper in development for embedding the library a pure C library. This wrapper reflects the API of the OSC interface, but adds a few language-specific elements, such as hierarchical data-structures. The library does not implement network socket handling functionalities itself, this is the responsibility of the client environment. In order to input and output information directly to and from the scene, the name-space is described with hierarchical addresses that must conform to the SpatDIF specification. In the implementation of the external, the input of elements into the scene adhere to the OSC-style with a slash delimited format, whereas for the outputs from the external, the addresses are converted to a space-delimited format in order to avoid the dependency on an additional OSC-parser.

Additional commands that directly address functions of the library have a different syntax, which is not part of the SpatDIF specification, but instead are specific to the implementation of the library. These */spatdifcmd* messages concern the querying of information from the library and the setting and getting of variables necessary for the executing the queries.

File operations are not functions of the library itself. The external in MaxMSP and PureData implements the required file loading and saving methods, which are specific to their own environment.

3.1 Example Scene

For the following examples we use the canonical piece 'Turenas' by John Chowning [1] (for a detailed discussion of this piece in context, see also [8]). The beginning of the SpatDIF scene, including only the 'insect' trajectory at second 0:44, contains the following elements in an XML file format:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<spatdif version="0.3">
  <meta>
    <info>
      <annotation>turenas insect trajectory</annotation>
    </info>
    <extensions>media</extensions>
    <ordering>time</ordering>
  </meta>
  <time>44.0</time>
    <source>
      <name>insect</name>
      <position>0.0 8.0 0.0</position>
      <media>
        <type>file</type>
        <location>/sound/insect.wav</location>
      </media>
    </source>
  <time>44.078</time>
    <source>
      <name>insect</name>
      <position>1.359056 7.757522 0.0</position>
    </source>
```

The corresponding sound files have to be stored and transported alongside the SpatDIF file. It is therefore important to think in terms of SpatDIF bundles or projects rather than single files. We deliberately choose not to propose a container that combines sound files and scene descriptors in a binary format, since human-readability without additional software tools would be lost.

3.2 Playback

The first example deals with file-handling and the playback of a SpatDIF scene in a multichannel loudspeaker setup. Figure 3 shows a simple MaxMSP patch where a monophonic audio file is spatialised to eight loudspeakers via the ICST Ambipanning external [6]. This workflow makes a few assumptions which are not limitations of 'libspatdif' as such, but reductions that help to clarify the concepts. The program demonstrates the rendering of the 'Turenas' scene excerpt. The scene is stored on disk in a SpatDIF-formatted XML-file together with the audio content as a sound file. After reading the scene from disk, the meta section can be parsed in order to obtain annotation information, as shown in the lower right.

In a fully dynamic system, additional information is required to set up the rendering algorithm. For this purpose queries are made to the library to gather information about the number of entities present in the scene and the names of the entities as well as the extensions that are present. [6] This allows to determine the number of playback voices needed, so that hierarchical message routing can be set up according to the names of entities. In the example this step is omitted and only one playback voice is implemented with a hard-coded message-routing set to the entity-type of *source* and the entity-name called *insect*. Subsequently, the messages are routed to obtain the *position* core-descriptor required for the spatialisation process as well as the *media* extension with the *location* descriptor necessary to load files for playback. The sound-file player, visualisation, and spatialisation algorithms [11] shown here represent the minimal case and would normally be more fully implemented.

As mentioned earlier, the scheduling of events in time is a task of the client application. In the present example this functionality becomes necessary and therefore a method for time-based playback is demonstrated. The *spatdifcmds* necessary to run iteratively through the scene can be seen in the right half of the example patch. The basic action is to ask with *getDeltaTimeToNextEvent* for the delta times between subsequent events. Since a scene can contain sparse data at no fixed intervals, it is crucial to have a dynamic timing mechanism for playback. The command *setQueryTimeToNextEvent* sets the query-time variable to the next event, then the library gets queried for all the events at that point in time with *getEventSetsFromAllEntities*, and finally the time to wait until the next event is retrieved again. These commands form a loop that steps through the scene, something which is visualised through the orange connection going back to the 'spatdif' external. The tim-

[6] For more specific information about the concept of extensions in SpatDIF, please refer to [7, 8, 9].

Figure 3: Implementation of SpatDIF in a MaxMSP external (marked in yellow), demonstrating the playback of the Lissajous trajectory from John Chowning's 'Turenas'.

ing is executed by a delay which waits for the appropriate amount of time to the next event before re-triggering the same sequence. These commands are global to the scene, so that all events associated with any entity in the scene are retrieved. If only events from selected entities are desired, this can be achieved by filtering in the message routing system, or as will be shown below with more specific commands to the library.

3.3 Recording

The example shown in Figure 4, records spatialisation information originating from real-time input via a physical controller into a SpatDIF scene. Four joysticks are set up to control the playback and spatialisation of four monophonic point-source entities in a scene. As in the previous case, this example is a simplification of a real application, yet still represents a fully functional implementation. The patch is divided into two processes that run in parallel, the recording on the right side depending on the realtime processing on the left.

In the left half of the Figure 4, the real-time process starts from the controller-input and leads to sound spatialisation and multichannel output. The controller-input at the top feeds into a visualisation-tool before reaching directly the spatialisation-module. Underneath, keyboard-commanded start/stop switches and file-selection menus control four sound file playback modules.

On the right hand side of Figure 4 are the parts necessary to record the key-events into a SpatDIF-scene. The large message-box in the right shows the initialisations necessary to set up the scene.

In this example, each voice independently activates the recording of position information in synchrony with its file-playback. This mechanism is shown in the 'p voice' sub-window. Here, the *setPosition* 'spatdif' command is formatted with the correct entity-name and combined with the real-time position data arriving from the visualisation tool. Further 'spatdif' commands to set media events are */media/setType* and */media/setLocation*. These commands are tied to specific entities, and therefore need to be formatted with the entity-name and combined with the file-type and file-path of the media resources.

Once the media- and position-commands are formatted, they are sent directly to the 'spatdif' external for storage with a time-stamp obtained from the system. This time-stamp is calculated as relative time since the beginning of the recording and is represented in seconds.

The *setWriteTime* commands sets the writing 'cursor' in the scene, which will apply to all messages that arrive until a new value for the 'writeTime' variable is set. Grouping all incoming events in this way may be regarded as a type of 'frame-based' time-stamping and is defined in the SpatDIF-specification as a scene ordered according to time.

A 'track-based' ordering of the events is also possible with this method, as is demonstrated in this example. All events belonging to each entity may be recorded separately, and their time can be reset by setting the 'writeTime' variable to zero when starting the recording of a new entity's events. This 'overdubbing' method works without prob-

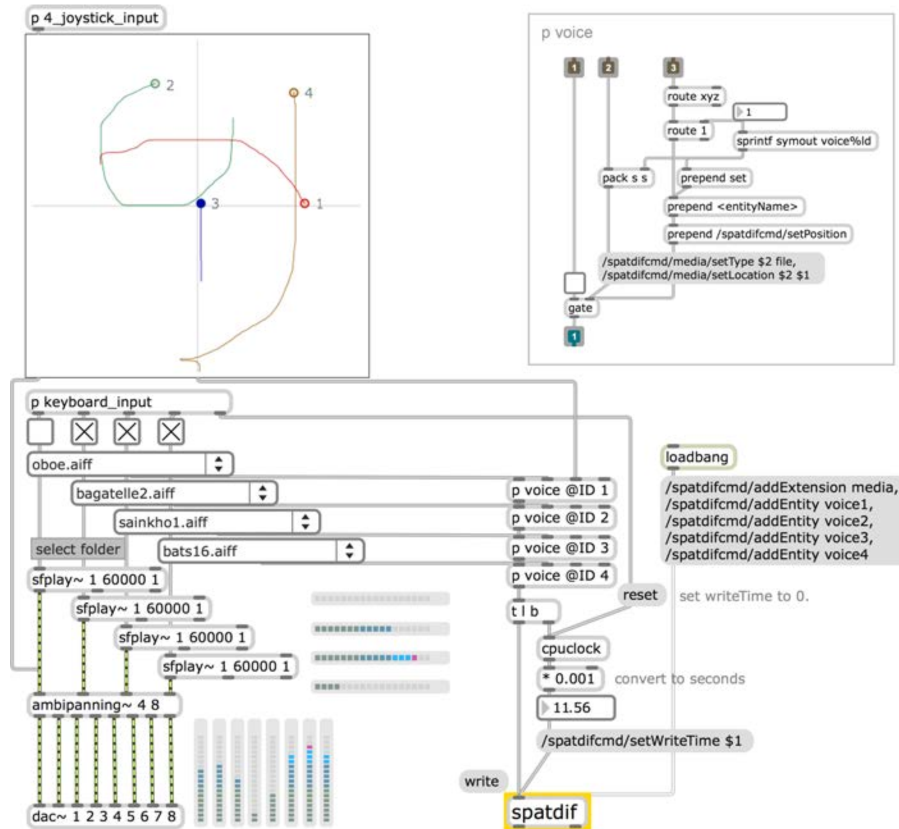

Figure 4: MaxMSP implementation, demonstrating the recording of four manually generated trajectories obtained from joysticks. The upper right shows the 'voice' section responsible for formatting the media and position inputs.

lems when different entities are concerned (entities here are synonymous for voices or tracks). When overdubbing events of the same entity, however, is it only possible to directly overwrite events if the time-stamps correspond exactly to the ones already stored, as could be the case for example for points in time generated by an algorithmic processes. In a real-time case this is difficult, if not impossible, to guarantee, therefore it is advisable to clear an entity's entire content with a call to the commands */removeEntity* followed by */addEntity* before re-recording events.

In general, all interaction with 'libspatdif' occurs through the *spatdifcmds*-syntax. In a future version, input of pure SpatDIF-formatted OSC-messages will be implemented, eliminating the need to reformat the information to the *spatdifcmd* syntax.

3.4 C++ Implementation

The C++ example application implements the entire workflow for the playback of a SpatDIF scene. The application is called a 'renderer' in analogy to visual tools, because it renders audible, in a surround setup, the information contained in a SpatDIF 'bundle'.

The implementation has to solve all the tasks relating to file-handling, handling OSC-streams, instantiating the voices of the playback engine, panning, distance cues, and handling other descriptors present in the SpatDIF specification version 0.3 [9].

In order to provide a relevant example for the application of the 'libspatdif', the scope of the application has been limited deliberately. Again, the panning algorithm used is the spatial windowing algorithms named 'ambipanning' [6] that is highly flexible, easy to implement, not tied to a specific number of speakers and usable without modification both in two and three dimensional spatialisation situations. The application provides a stand-alone implementation, with a basic 3D visualisation of the scene, and the possibility to play the scene in a stereo speaker setup. It allows to load a SpatDIF file with associated sound files and play it through a few predefined multichannel speaker layouts.

This application is implemented in OpenFrameworks, which provides a powerful C++ toolset and has a thriving and helpful community. It produces both a sonic and visual rendering of the scene. In analogy to the external for MaxMSP and Pure Data, this implementation encapsulates all the functionalities concerning calls to the library in its own class or 'addon' named 'ofxSpatDIF'. This 'addon' reflects the interface found in the external.

Since OpenFrameworks is not particularly oriented towards audio, the classes provided for sound processing are somewhat rudimentary. However, and that is its strength, many extensions exist and it is simple to add new functionalities and tie in external libraries. 'libsndfile' [7] is such an external library. It provides a powerful audio file handling toolset and is linked in as a dynamic library, as is stipulated by its license.

[7] `http://www.mega-nerd.com/libsndfile`

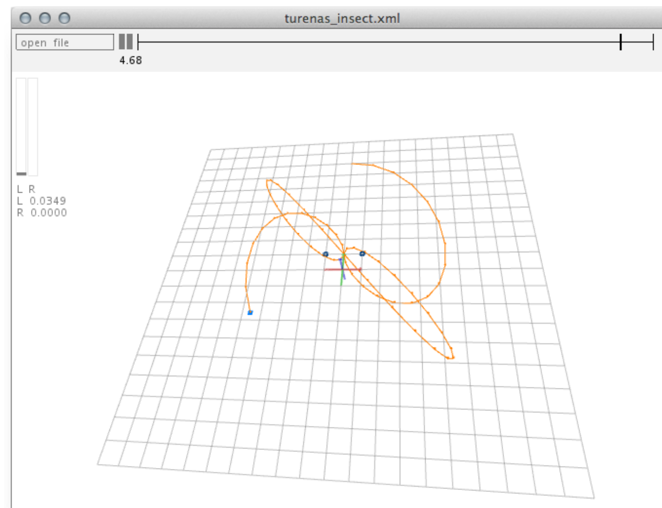

Figure 5: OpenFrameworks implementation of SpatDIF, demonstrating again the playback of the 'Insect' trajectory from 'Turenas'

The visual representation is a bare-bones wireframe drawing of the scene in OpenGL as seen in Figure 5. The sound playback is processed through the sound-stream interface provided by the environment. The process is straightforward, even if its implementation is a bit delicate.

The audio samples in OpenFrameworks are calculated in blocks, as in most sound processing applications. After retrieving samples from the sound files via 'libsndfile', the panning and distance corrections that were calculated from the current scene information for each speaker position are applied to the sound signals before each block of samples is output. The signal processing chain for this example is deliberately kept simple, to provide a clearer view of the implementation of such a process.

4 Conclusions and Outlook

In this article we show concrete implementations of SpatDIF in two types of software. The software library's purpose is to provide a reference implementation of the specification. It also to serves as a pre-built and tested tool that facilitates the use of SpatDIF in many environments. The external for MaxMSP and Pure Data represents the first application of this idea, while the C++ implementation in OpenFrameworks provides an additional point of reference.

Both the software library and the external and C++ application are work-in-progress and will be developed and enhanced further as the project progresses. It is important to keep in mind that the examples presented in this article only demonstrate limited use-cases, which need to be worked out more fully for a real-life applications. By providing software components both on a low and an intermediated level, the intention is to make available an accessible method for working with SpatDIF in audio and processing environments, which should be flexible enough to handle all layers of a spatial audio workflow [10]. A greater challenge in the future will be the application of these tools in commercial hosts, in particular within DAWs that only expose a small part of their structure to external access, for example by plug-ins.

A short-term goal of this project is to finish the implementation of the existing specification version 0.3 [9] within the library. One such implementation that is currently in the works is the 'Interpolation' extension, which permits to query the scene at arbitrary points in time and returns interpolated values from descriptors, where this makes sense. In addition, the methods for purely OSC-driven input and output have to be completed as well as loading and saving of scenes in other markup languages such as JSON. A long-term perspective is to develop additional extensions that will further increase the utility of the SpatDIF format. In addition, the introduction of new entities and extensions should extend the palette of spatial audio scene descriptors. For instance a 'room' entity type would enable the description of room acoustics in a *reverb* extension.

The next iteration of the specification will be shaped by the experiences gathered while implementing SpatDIF into the presented tools, and will be addressing mostly technical issues that have become apparent. This will take less effort to integrate into the library, external and 'addon' thanks to the consistent and open design of the library.

The source code for 'libspatdif' is licensed under the 'FreeBSD' license [8] and can be obtained through:
`git clone http://code.zhdk.ch/git/spatdiflib.git`
The 'spatdif' external's source code and the SpatDIFRenderer's C++ implementation are under the 'FreeBSD' license and can be obtained through: `git clone http://code.zhdk.ch/git/spatdifrenderer.git`
A preliminary version of the external and help file for Pure Data and MaxMSP can be downloaded here: `http://www.icst.net/research/downloads/spatdif-external/`

[8] `http://opensource.org/licenses/BSD-2-Clause`

Acknowledgments

These software developments for the SpatDIF project are funded by the Institute for Computer Music and Sound Technology of the Zurich University of the Arts.

References

[1] J. Chowning. "Turenas: the realization of a dream". In: *Proc. of the 17es Journées d'Informatique Musicale*. Saint-Etienne, France, 2011.

[2] Dolby. *Dolby Atmos: Next-Generation Audio for Cinema*. white paper. Dolby Laboratories, Inc., 2012.

[3] M. Geier and S. Spors. "ASDF: Audio Scene Description Format." In: *Proceedings of the International Computer Music Conference*. 2008.

[4] Matthias Geier and Sascha Spors. "Spatial Audio with the SoundScape Renderer". In: *27th TONMEISTERTAGUNG – VDT INTERNATIONAL CONVENTION*. Nov. 2012.

[5] Chikashi Miyama, Jan C. Schacher, and Nils Peters. "Spatdif Library – Implementing the Spatial Sound Descriptor Interchange Format". In: *Journal of the Japanese Society for Sonic Arts* 5.3 (2013), pp. 1–5.

[6] Martin Neukom and Jan C. Schacher. "Ambisonics equivalent panning". In: *Proc. of the International Computer Music Conference*. Belfast, UK, 2008, pp. 592–595.

[7] Nils Peters, Trond Lossius, and Jan C. Schacher. "SpatDIF: Principles, Specification, and Examples". In: *Proc. of the 9th Sound and Music Computing Conference*. Copenhagen, DK, 2012, pp. 500–505.

[8] Nils Peters, Trond Lossius, and Jan C. Schacher. "The Spatial Sound Description Interchange Format: Principles, Specification, and Examples". In: *Computer Music Journal* 37.1 (2013), pp. 11–22.

[9] Nils Peters, Jan C. Schacher, and Trond Lossius. "SpatDIF specification Version 0.3, draft version". `http://redmine.spatdif.org/projects/spatdif/files`, last accessed Oct. 2012. 2010–2012.

[10] Nils Peters et al. "A stratified approach for sound spatialization". In: *Proc. of the 6th Sound and Music Computing Conference*. Porto, PT, 2009, pp. 219–224.

[11] Jan C. Schacher and Phillippe Kocher. "Ambisonics Spatialization Tools for Max/MSP". In: *Proc. of the International Computer Music Conference*. New Orleans, USA, 2006, pp. 274–277.

[12] E.D. Scheirer, R. Vaananen, and J. Huopaniemi. "AudioBIFS: Describing audio scenes with the MPEG-4 multimedia standard". In: *IEEE Transactions on Multimedia* 1.3 (1999), pp. 237–250.