An Open-Source C++ Framework for Multithreaded Realtime Multichannel Audio Applications

Matthias GEIER[1], **Torben HOHN**[2] and **Sascha SPORS**[1]

[1]Quality & Usability Lab, TU Berlin
Ernst-Reuter-Platz 7
10587 Berlin, Germany
matthias.geier@tu-berlin.de
sascha.spors@tu-berlin.de

[2]Linutronix GmbH
Auf dem Berg 3
88690 Uhldingen, Germany
torbenh@linutronix.de

Abstract

An open-source C++ framework is introduced which facilitates the implementation of multithreaded realtime audio applications, especially ones with many input and output channels. Block-based audio processing is used.

The framework is platform-independent and different low-level audio backends can be used for both realtime and non-realtime operation. Support for further backends can be added easily.

Keywords

MIMO, realtime, multithreading, C++

1 Introduction

Most realtime audio processing applications which have to handle a high number of input and output channels are bound to a specific realtime audio framework. This is not a problem in itself, because there are several very good frameworks available. Some of them can be used on several platforms and some of them even have the possibility to switch between different audio backends.

Such frameworks are perfectly suitable to create stand-alone applications. But what if you want to compare the audio output of your program with the prototype algorithm you realized in your favorite software for numerical processing? Wouldn't it be nice to create a shared library for this software directly from your realtime C++ code? And while we are at it, wouldn't it be nice to use the same source code to create a plugin for your favorite graphical patching software?

Multichannel applications often have an embarrassingly high potential for parallel computation on multi-processor/multi-core computers, especially if they have many output channels. Wouldn't it be nice if parallel processing would be automatically provided in all the aforementioned scenarios?

To tackle these challenges, this paper presents a new C++ framework for interactive multiple-input/multiple-output (MIMO) audio applications. It is part of the *Audio Processing Framework* (APF)[1], which is free software released under the GNU General Public License (GPL)[2].

Applications created with this framework are automatically capable of multithreading. The number of audio threads is chosen by the user and does not change during runtime. The scheduling mechanism is simple and static, yet effective (see section 5). Applications are not bound to a specific audio driver. Audio backends for both realtime and non-realtime operation can be switched easily (at compile time) and new backends can be added by the user. This way, it is possible to implement audio algorithms for a realtime application, but the algorithms can nevertheless be evaluated block-by-block in a non-realtime manner. For more information about the audio backends and their usage see section 4.6.

2 Target Applications

The presented framework can be used for any block-based audio application with many input and output channels. The overall topology of the audio processing graph should not change too much during runtime, but the number of channels and other system parameters may change dynamically. Typical applications are sound field synthesis, multichannel echo cancelling and loudspeaker/microphone beamforming. The target systems range from stereo processing on a simple laptop to dedicated computer systems driv-

[1]http://tu-berlin.de/?id=apf
[2]http://gnu.org/copyleft/gpl.html

ing tens or even hundreds of loudspeaker channels with as many input channels. The first application using the framework is the *SoundScape Renderer* (SSR)[3], a tool for object-based spatial audio reproduction providing a variety of rendering algorithms [Geier et al., 2008].

3 Realtime & Non-Realtime Threads

An application based on the presented framework uses two different kinds of threads. The actual audio processing is mostly done in a callback function which is called successively – for every audio block – from the audio backend. The thread running this callback function is called *realtime thread* because only a given time period is available for the computation of each audio block. Therefore, only a certain fixed amount of calculations may be done, depending on block size, sampling rate, processor speed and other factors. Additionally, no blocking functions may be called in the realtime thread. Especially operations like allocating memory, reading and writing of files and sockets, creating and joining threads and waiting for mutexes have to be avoided. If processing of an audio block is not finished at the required time, the output data is not ready on time and has to be discarded, leading to errors in the output signal.

The other kind of thread is simply called *non-realtime thread*. Threads of this kind handle input from the user interface, read and write files, reserve and free memory, communicate via network and do anything else which is not related to audio processing.

The timing of the audio callback function is normally bound to the soundcard, which imposes the mentioned realtime constraints. However, – as described in section 4.6 – there is a special audio backend which can be used for offline processing. In this case, realtime-safety would not be needed anymore. The thread running the audio callback function should nevertheless be considered as realtime thread. Audio algorithms should be backend-agnostic and assume to be potentially used in a realtime context.

In interactive applications, information must be transferred from the non-realtime thread to the realtime thread. This should not be done by writing to and reading from the same memory location in both threads, respectively, because this can lead to data corruption. A mutex cannot be used either, because this would be realtime-unsafe. Instead, a lock-free queue is used (see section 4.3).

If more memory is needed for audio processing or if unused memory shall be freed, this may not be done in the realtime thread. Therefore, if new memory is needed, it is allocated and initialized in the non-realtime thread and then a pointer to the new data is transferred to the realtime thread in a realtime-safe way. This is realized by means of the realtime-safe list described in section 4.4.

4 Components

The framework is implemented in C++ using the Standard Template Library (STL). Many structural decisions are made at compile time to avoid unnecessary runtime overhead. In most cases, generic programming is preferred over classic object-oriented programming. Dynamic polymorphism is only used where actually needed.

The framework does not define a special data type for audio data, only the audio backends (see section 4.6) define a sample type (in most cases – but not necessarily – `float`). Within the audio algorithms themselves, arbitrary data types can be used.

The following sections describe the main components of the framework.

4.1 Lock-free Ringbuffer

The key component to make the framework both realtime-safe and thread-safe is a lock-free ringbuffer. It is available as the class `LockFreeFifo<Command*>` and it is only thread-safe for single-reader/single-writer access. This means that only one single thread is allowed to use the `push()` function to write data to the ringbuffer and only one other thread is allowed to use the `pop()` function to retrieve data from the ringbuffer. It has to be ensured by the programmer that each function is only used in the appropriate thread.

4.2 Command Queue

Two instances of the lock-free ringbuffer are used in the `CommandQueue`, a lock-free queue for arbitrary user-defined commands which can be defined by implementing the member functions `execute()` and `cleanup()`.

[3]`http://tu-berlin.de/?id=ssr`

Every command object is created and initialized in a non-realtime thread, all necessary memory is allocated at this time. A pointer to the fully constructed command object is pushed to an instance of `LockFreeFifo<Command*>`. The main realtime thread periodically (usually once per audio block) processes all items from this queue with the function `process_commands()`, which in turn invokes the virtual `execute()` function on each command object. Of course, the `execute()` function is not allowed to use any realtime-unsafe functions. After execution, it is not safe to deallocate the memory used by the command object within the realtime thread. Therefore, the command object is pushed into another instance of `LockFreeFifo<Command*>` to be deallocated in the non-realtime thread. Before that, the virtual `cleanup()` function is called from the non-realtime thread.

To add a command to the `CommandQueue`, the member function `push()` is used, the function `wait()` can be used to wait until the command is actually executed and cleaned up. No actual commands are defined by the `CommandQueue`, only the abstract base class `Command` from which arbitrary commands can be derived as long as they provide an implementation for the `execute()` and `cleanup()` member functions.

The transport of information from the realtime thread to the non-realtime thread also has to be triggered by the latter. For that, commands can be defined which query the information in the `execute()` function and provide it to the non-realtime thread in the `cleanup()` function.

Typically, an application holds exactly one instance of `CommandQueue` which is responsible for all the communication to and from the realtime thread. If there are several non-realtime threads – for example for a graphical user interface and a network interface which work in parallel – access to the `CommandQueue` has to be locked with a mutex. This is not done automatically.

4.3 Thread-safe Data Access

To avoid parallel writing and reading at the same memory location from a realtime and a non-realtime thread, respectively, the class `SharedData<T>` can be used. It wraps a single variable of any standard or user-defined type. Internally, it uses a reference to a `CommandQueue` to push a custom `Command` which holds a copy of the value that is assigned to the actual variable in the `execute()` function.

4.4 Realtime-safe List

To handle a dynamic number of audio channels, the realtime-safe data structure `RtList<Item*>` is used. In the non-realtime thread, list elements are created, initialized and added to the list with the `add()` function and elements are removed with the `rem()` function, whereby the deallocation of any memory also happens in the non-realtime thread. Communication between threads is again handled by means of the `CommandQueue`. In the realtime thread, the relevant functions of a `std::list` can be used, namely `begin()`, `end()`, `empty()` and `size()`.

When adding items to the `RtList`, ownership is passed to the list. That means the responsibility to destroy the object at an appropriate time is transferred to the list and user code may not call `delete` on an object owned by an `RtList`.

4.5 MIMO Processor

The class `MimoProcessor` is the base class for the signal processing part of an application. When deriving from `MimoProcessor`, several template arguments have to be used. The first one is the deriving class itself. This C++ idiom is called Curiously Recurring Template Pattern (CRTP) [Coplien, 1995] and it is used to achieve compile-time polymorphism. The rest of the template arguments are so-called policy classes. The user can choose from a given set of policies and even implement new ones. This programming strategy is called policy-based class design [Alexandrescu, 2001]. One of the policies selects the audio backend which is described in section 4.6.

The `MimoProcessor` incorporates one instance of `CommandQueue` for all realtime-safe and thread-safe operations. Instances of `RtList` and `SharedData` can be used with this queue.

4.6 Audio Backends

The `MimoProcessor` base class is not limited to a specific audio backend. By means of policy-based design, the audio backend can be specified as template argument at compile time.

Currently, the JACK Audio Connection Kit

(JACK)[4] is supported (`jack_policy`), support for PortAudio[5] is underway and further audio backends can be added easily.

A special policy class is the `pointer_policy`, which can be used with any C or C++ program; in this case, the audio callback function has to be called explicitly. This can be used in a realtime context, for example in an External for *Pure Data*[6] and *Max*[7] (utilizing the *flext* library). However, it can also be used in a non-realtime context for offline processing, for example in a MEX-file (shared library) for *Matlab*[8] and *GNU octave*[9] or in an application which reads all inputs from a multichannel audio file and writes all outputs to another multichannel file. Example code is available for all mentioned applications.

4.7 Crossfade

When doing block-based processing, the input signal is divided into consecutive blocks of audio data and it is assumed that all parameters of the algorithm are constant during one audio block.

Parameter changes only happen between blocks and this can very often lead to audible artifacts due to discontinuities in the resulting output signal. One way to reduce these errors is to calculate each block two times – once with the parameters of the previous block and once with the current parameters – and make a crossfade between the two resulting blocks. With large parameter changes and small blocksizes there may still be audible artifacts, but in most cases the transitions become inaudible.

The described crossfade functionality is built into the `MimoProcessor`. Only if parameters change – as noticed by the `CommandQueue` – the specified audio algorithm is processed twice and a crossfade is done automatically. In static scenarios the overhead of crossfading may not be wanted. Therefore, the whole functionality can be switched off at compile time.

5 Parallel Processing

The desired audio processing algorithm has to be organized in several lists of type `RtList<Item*>`. These lists hold polymorphic base class pointers and allow the execution of the virtual function `Item::process()` on each list item. To do the actual signal processing, item classes have to be defined – derived from the class `Item` – and they have to implement their signal processing algorithms in the `process()` function. By implementing the list items and assembling them into lists, the programmer can establish the overall audio processing graph. The different lists can be seen as *stages* of the algorithm. Lists are processed one after each other, items within one list are processed in parallel by several threads.

The `process()` function is called from a realtime thread and has to fulfill the following constraints. It may write data only to places where the item has exclusive access, either in member variables of the object itself or in dedicated memory areas elsewhere. It may read data from any of the `RtLists` except the list where its object belongs to, because other list elements could be written to at the same time by another realtime thread. The amount of memory used by a list item may not change during its lifetime. It must be allocated in the non-realtime thread and when the item is removed, its memory must also be deallocated in the non-realtime thread. This happens automatically if the `add()` and `rem()` functions of the `RtList` are called from the non-realtime thread.

Especially in applications with a dynamic number of channels, `RtLists` can be iterated over by means of the `begin()` and `end()` functions which reflect possible changes in the length of the list. In less dynamic scenarios, references to other objects – as long as they will not be removed during runtime – can also be specified at initialization of the list item.

Parallel processing is achieved by automatically executing different `process()` functions in different realtime threads which can run in parallel on multi-processor/multi-core computers. The number of realtime threads can be specified by the user, according to the available resources, and this number does not change during runtime. If N threads are requested, $N-1$ *worker threads* are created. The *main audio thread* – the one where the audio callback function is called from – is normally created and controlled by the audio backend.

[4]`http://jackaudio.org`
[5]`http://portaudio.com`
[6]`http://puredata.info`
[7]`http://cycling74.com/products/max`
[8]`http://mathworks.com/products/matlab`
[9]`http://gnu.org/software/octave`

The distribution of list elements to the audio threads is very simple. The main audio thread gets every N-th item starting with the first, the first worker thread gets every N-th item starting with the second and so on until the $(N-1)$th worker thread, which gets every N-th item starting with the N-th. This very basic scheduling mechanism may not be the best choice for arbitrary audio processing graphs, but it works quite well for the targeted application areas. It works best if the length of the lists is much bigger than the number of threads and if the amount of work done in the `process()` function is not too different in all items of a given list.

The main audio thread and the worker threads are synchronized with semaphores. Each worker thread does a very simple cycle of operations repeatedly. It waits until signaled from the main audio thread by a semaphore and then processes all items of the current list which are chosen by the aforementioned scheduling algorithm. When finished, it signals to the main audio thread and waits again until the next iteration. The cycle of the main audio thread can be defined by the user. Typically, several lists are processed one after each other with the function `_process_list()`. Within this function, the specified list is set as current list, the worker threads are signaled via their semaphores to start processing and the main audio thread itself then also processes the appropriate items of the current list. Afterwards, the main audio thread waits until all worker threads have signaled that they are done with their parts of the list.

6 Additional Components

As mentioned in the introduction, all presented components are part of the Audio Processing Framework (APF). Additionally, several other components are included in the APF, among them a delay line, a partitioned convolution engine (using the FFTW library[10]), several specialized iterator classes, math operations and many helper functions. For a full list of features and more detailed information see the online documentation[11].

[10]`http://fftw.org`

[11]`http://dev.qu.tu-berlin.de/projects/apf`

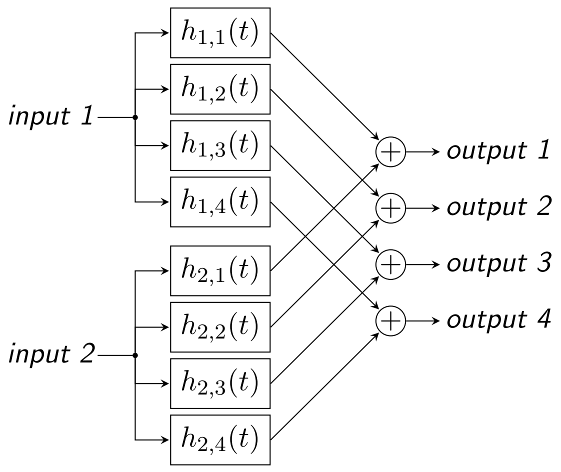

Figure 1: Example MIMO system with 2 input channels and 4 output channels. In the general case of N inputs and M outputs, $N \times M$ filters are needed.

7 An Example

Figure 1 shows a generic MIMO system with a filter between each input and each output. The main objective of the programmer is to identify which parts of the algorithm shall form separate objects, how they depend on other objects and which of these objects can be processed in parallel. In the example, three kinds of objects come to mind – the input channels, the filters and the additions (which can be combined with the output channels).

To implement the filter class, the convolver which was mentioned in the previous section can be used. A filter object is always bound to one specific input channel, so a reference can be specified at initialization and stored within the filter object. Because of that, the `process()` function of the filter object does not need to iterate over the whole list of inputs but can rather get the input data directly via the internal reference to the corresponding input channel. Input data is processed by the convolver and the result is written to a member variable inside the filter object. Apart from these processing instructions, the filter object must also provide means for the following stages of the algorithm to read the output data of the filter. This is normally done by providing (read-only) `begin()` and `end()` functions which return iterators to the data. Of course, the filter class has to be derived from the class `Item` so that all filter objects can be added to a list of type `RtList<Item*>`. Having a dynamic number of inputs is no problem. Whenever an input is added, a whole set of filter objects – one for each output channel – has to be created an added to the list at once. To safely switch filter coefficients during runtime, the `CommandQueue` is used.

The other class which has to be defined in this example is for the addition, which can be combined with the output channels. Each addition corresponds to one unique output channel. The `process()` function of the addition object must iterate over all filter objects and find out which of the filter outputs must be added. Therefore, each filter object must contain a flag of some kind to be associated with a certain output channel. If the number of inputs is supposed to change dynamically, this information cannot be stored in the addition object itself because this would require dynamic memory allocation which is not allowed in a realtime context. Once all relevant filter data is added, the result can be written directly to the corresponding output.

Most of the work is done in the `process()` functions. The only thing left to do is to call `_process_list()` for each of the defined lists in the main audio callback function.

8 Acknowledgements

Thanks to Till Rettberg for many helpful suggestions and discussions. This work was partly funded by German Research Foundation (DFG) grant FOR 1557.

References

Andrei Alexandrescu. 2001. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley.

James O. Coplien. 1995. The column without a name: Curiously recurring template patterns. *C++ Report*, 7(2):24–27.

Matthias Geier, Jens Ahrens, and Sascha Spors. 2008. The SoundScape Renderer: A unified spatial audio reproduction framework for arbitrary rendering methods. In *124th Convention of the Audio Engineering Society*.