AudioFX

Embedded Software Engineering Course
Winter 2009 / 2010

Christian Ebner
Daniel Sürth

Table of Contents

1 Introduction

The goal of our project AudioFX was to process an audio input signal in real-time. Before the audio device sends the audio signal to the speakers the signal is modified by an audio-effect-module.

2 Sound Card, Drivers and different Operating Systems

The sound card processes the input and output signals.

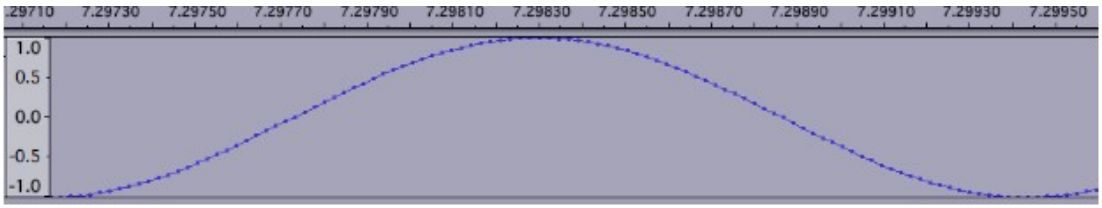
On windows systems proprietary audio drivers are provided from manufacturers. Most UNIX systems use OSS (Open Sound System) as a sound driver. The MacBook Pro we used has an Intel high definition audio driver.

Since there exist so many different drivers for different operating systems we looked for a library which can be used on different platforms and we found OpenAL.

3 Audio Signal Processing

When the analog audio signal is transformed to a digital one, the audio card splits the signal into audio samples.
The information of one sample is coded into a number of bits so the bit-rate is defined as the number of bits per samples. A CD for example has a bit-rate of 16 bits/sample.

The sample rate is defined in Hz. 1 Hz is one oscillation per second.
If we take a sample rate of 44100 Hz and a bit-rate of 16 bits/sample, then 44100 samples have to be processed within a second, that are about 86 KB a second.



Mono/Stereo:
Mono or monophonic describes a system where all the audio signals are mixed together and routed through a single audio channel whereas stereophonic systems have two independent audio signal channels, one for the left and one for the right side.
In a stereo buffer stream the samples of the left and the right channels are stored in turns.

Audio Effects
The effects we implemented are echo/delay and pitch.
An echo is defined as a repetition of the original signal within 50ms. One or more delayed signals are added to the original signal. The delayed signals have to last 50ms or longer.
With the technique of pitching it is possible to shift the original signal up or down in pitch.

4 OpenAL

4.1 Description

OpenAL is a cross-platform audio API developed by Creative Labs and written in the programming language C.

4.2 Installation

OpenAL may be used within different operating systems. We chose Windows (XP, Windows 7) as well as a Linux version for enabling the scheduling of existing task.

For using OpenAL on Windows, the dynamic-linked library OpenAL32.dll has to be installed. This will be done by an installer provided by the OpenAL team. For using the API while programming AudioFX, needed c header and the corresponding OpenAL32.lib library were taken from the OpenAL 1.1 Core SDK.

For using OpenAL on Linux, we downloaded the source code and compiled it using cmake and make. The header files were copied to /usr/include/AL and the library to /usr/lib/

4.3 OpenAL model

Working with OpenAL includes working with at least one device. This device includes different basic items which are used within OpenAL: buffers, sources and listeners. If the device is opened, a context will be created which includes a listeners object. Nevertheless, we are not concentrating on the listener object in our project, since this feature is more important for adding 3D audio effects and we are more interested in basic audio effects instead of adding 3D effects.

As demonstrated in fig 1, a device may have multiple sources and buffers. Captured audio signals have to be stored in those buffers and the source will continue playing as long as buffers are filled.

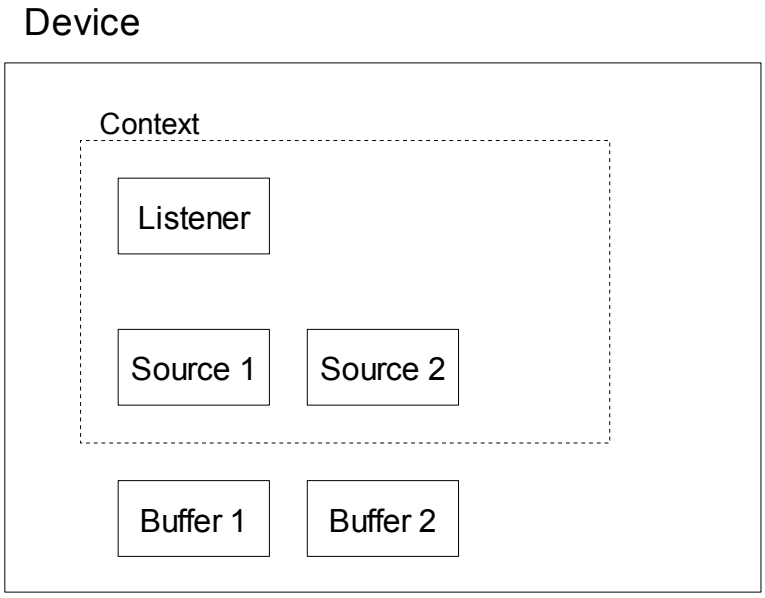


Figure 1. OpenAL objects

5 Threading

In the AudioFX project, two additional threads are created by the OpenAL API. The actual called functions are depending on the used audio drivers. In the following example, files are analyzed which are related to the Open Sound System (which is used on Linux).

5.1 Task Model

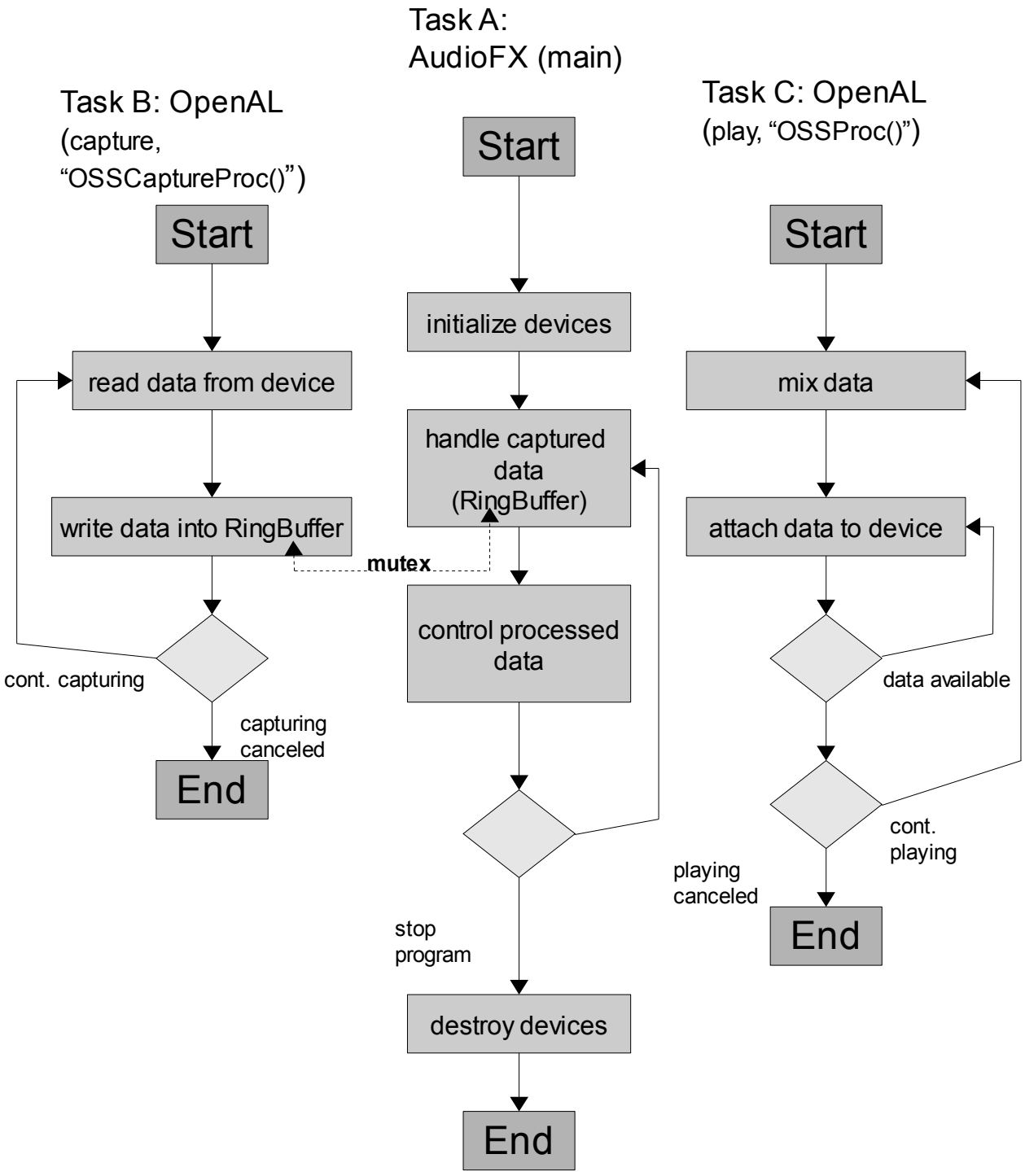


Figure 2. Task model

In fig 2, three tasks are shown:

(1) Task A represents the main program of the AudioFX project. Running this tasks includes initializing all used OpenAL structures, as devices, sources, buffers, etc. Errors within this phase are crucial and means for example that a playing device could not be found.

Afterwards the program starts to manage the capturing and playing process. This is sequentially done in the two functions `capture()` and `play()` which are called rotationally The capturing function checks, how many audio signals have been written to an internal ring buffer since the last call of the function. If there are enough signals available for filling an audio source-related buffer, the data will be retrieved from the ring buffer and saved to one of available audio source-related buffers.

In the play function, filled buffers will be enqueued to the specific audio source. Furthermore, buffers which are already processed (what means that the audio signals have already been played), are recognized and the buffers are marked to be reused.

Due to the programmers manual it may happen that an audio source stops playing if there are no further buffers attached to it. Although this never happens in our tests, we are checking if the source is actually playing or should be restarted.

(2) Task B is part of the OpenAL API and runs a function called OSS`CaptureProc`. It is started when an new capturing device (e.g., a microphone) is opened in the main program of the project. The task will end only, if the capturing process is stopped. During the runtime, the captured data is retrieved by using the OSS driver to access the audio device. The audio signals are written to an internal ring buffer which may be read using OpenAL commands (`alcCaptureSamples()`). Reading form or writing to the internal ring buffer is constructed as a critical section, which is secured by a mutex.

(3) Task C is started, after a new context has been created. As described in chapter 4.2, a device has at least one context, so this thread starts immediately after OpenAL has been initialized. The task is part of the OpenAL API and executes a function called OSSProc. As long as the initialized device is not destroyed, the stored data is mixed what means that the data is ordered according to the audio format (e.g., mono or stereo). Afterwards, the signals are written to the audio device using the Open Sound System driver and the signals are played.

6 Scheduling

6.1 Default scheduling

OpenAL creates new threads without defining the priority or the algorithm of a the thread. Therefore, the threads are scheduled using Round Robin.

6.2 Enabling access to threads for scheduling

We were interested how to access all threads which are created in AudioFX, either in our own code or in the code provided by OpenAL. If we are able to access them, we can change their behavior including their priorities.

Therefore, we decided to modify the OpenAL API code, which is open source. Every thread is created in a function called `StartThread()` in the file `alcThread.c` which belongs to OpenAL. That means, we have to change only very view spots of the external source code, although it took some time to detect these spots.

It is possible to recognize all started threads within this function as may be seen in table 1. The bold lines in this table has been added by us. The created threads are saved in an array which is accessible via a getter function

```
ALvoid *StartThread(ALuint (*func)(ALvoid*), ALvoid *ptr)
{
 ThreadInfo *inf = malloc(sizeof(ThreadInfo));

 if(!inf) return NULL;

 inf->func = func;
 inf->ptr = ptr;


 if(pthread_create(&inf->thread, NULL, StarterFunc, inf) != 0)
 {
 free(inf);
 return NULL;
 }
 // we are actually using only two threads of the OpenAL API in
 // the AudioFX project.
 if (AUDIOFX_threadPos < 2)
 {
   AUDIOFX_allThreads[AUDIOFX_threadPos++] = inf->thread;
 }

 return inf;
}
```

Table 1. Code snippet from alcThread.c

The getter function getAUDIOFX_Threads() has been added to the same file and is part of the OpenAL header called al.h, to be accessible. Now, all threads, either created by our code, or by OpenAL, may be controlled.

```
pthread_t *threads = getAUDIOFX_Threads();
```

Table 2. retrieving OpenAL threads within the AudioFX project

The modifications of the OpenAL source code are only valid for operating systems which are not based on MS Windows. Therefore, this OpenAL code might not compile when used on Windows.

6.3 Changing priorities

Having the information of the previous two paragraphs, we are now able to change the behavior of every thread within the project using POSIX standard commands (see table 3).

```
pthread_t thr;
pthread_attr_t pta;

pthread_attr_init(&pta);

struct sched_param param;
param.sched_priority = 10; // 10 has to be considered as example

pthread_attr_setschedpolicy(&pta, SCHED_FIFO);
pthread_attr_setschedparam(&pta, &param);

pthread_create(&thr, &pta, thread_func, NULL);
```

Table 3. Setting thread behavior

6.4 Future work

The current version of AudioFX is programmed for a MS Windows operating system. The concepts of this chapter are meant for a Linux version, maybe even with an installed real-time patch to ensure the aimed behavior and control. Due to the deadline set for the Audio FX project, the behavior of a Linux including a real time path could not be tested.