1. Audio programming using C++ and Pure Data

Jonathan Adam (jadam@kth.se), Marcus Groth (magroth@kth.se); Nicolas Pignier (pignier@kth.se)

This report demonstrates two approaches to program audio applications: a low-level approach using C++, the PortAudio API and the Libsndfile library, and a high-level approach using Pure Data and the OSC protocol. For the first part of the task, the low-level approach, we have implemented a record-and-playback program. For the second part of the task, the high-level approach, we have designed a mixer mixing four pre-recorded .wav files and a live input, with individual controls such as low- and high-pass filters, volume and reverb.

1.1 Introduction

Audio programming comes with its own set of challenges. Working from the system level up, programs dealing with audio must conform to strict timing requirements – sound cannot simply wait for another process to finish. Working on a more abstract high level, programmers nowadays can avoid dealing with low-level concerns but have to figure out how to translate their creative ideas into code. In this project we created two programs to acquaint ourselves with some of the challenges, pitfalls and rewards of various programming approaches to audio.

1.1.1 Implementing a low-level player and recorder

We created a basic-functionality audio player and recorder in C++ using the Portaudio library. C++ was appropriate for this project as it exposes the lower levels of abstraction needed to fully grasp what is going on under the hood. It was also a language that some members in our group already had familiarity with, ensuring we could focus primarily on audio paradigms as opposed to dealing with learning a new language. Portaudio was the audio library of choice because it provides an interface to audio programming that works across operating systems, allowing us to work on one operating system in the safety that the audio programming aspects will continue to work on other platforms.

1.1.2 Visual programming of a high-level mixer with remote control of a live input

For this part of the work, we designed a mixer controlling five channels of sound, four of which being instrumental tracks read from .wav files and the remaining one consisting of a live input. Pure Data (Pd) was the visual programming software of choice for this task because it allows a high-level control of algorithms through a dispatching of blocks linked together by connectors. An important feature of Pd is that it is possible to transfer external information via protocols such as MIDI or OSC. High-level audio programming leaves a large freedom for creative ideas, managing for the user the issues of low-level audio processing.

1.1.3 Group workflow

Working as three people with different backgrounds meant allocating various tasks to different group members. None of us had much experience with system programming, but as Jonathan at least had some knowledge of C++ he took the first stab at the problem in order to get some of the boilerplate code in place. Meanwhile, Marcus and Nicolas began to get acquainted with Pure Data. As the project progressed, we used group meetings not just to track progress but to teach each other what we had learned, so that by the final weeks the two programs were a collaborative effort. Working separately at first and growing towards each other allowed us to take advantage of each others' skills to learn more ourselves, and was more enriching than a simple divide-and-conquer would have been, or a project where we felt stuck with a technology we could not understand at all.

1.2 Record and playback of audio using C++

1.2.1 Overview of the main problem

In low-level audio programming, timing issues are paramount. In order to ensure smooth recording or playback without glitches, the program must deal with the audio device (be it microphone or speaker) with total regularity to avoid any breaks in the flow of samples. This means careful programming needs to avoid audio having to wait for a process to end, or for a resource to be freed by another thread. In the case of short segments of audio, a program can get away with storing audio in a buffer in memory, allowing for rapid access. This is not feasible as sound files grow longer, if one wants to access files on disk, or if the amount of memory is not yet determined (eg. when recording the live input of a microphone for an unspecified amount of time.)

Our program has to deal with two main issues. Firstly, we need to ensure that audio is served to or from the audio device within the appropriate time. Secondly, we need to interact with the filesystem to read from and write to disk, without interfering with the timing of the audio. We do both of these things by using the architecture Portaudio provides us with, as well as threading our program in a safe way.

1.2.2 Structure of the program

Instead of forcing the problem into objects and classes, we approached this program from an imperative paradigm and divided various components of the problem into modules.

1.2.2.1 AudioWorker

The AudioWorker module functions as the main audio processing unit of our program. It is here that the most top-level functions are called, being *record_file(char filename, SF_INFO soundinfo)* or *play_file(char filename, double startingpoint)*. By default, *record_file* will record to a 16-bit mono PCM Wav file at 44.1 kHz, but these parameters can be changed by adapting the *SF_info* struct (which comes from the libsndfile library). By default, *play_file* will start playing from the beginning of the sound file, but given a *startingpoint* will seek in the file and begin playing at that moment. Another function exposed in AudioWorker's header file is *record_short*, which was simply intended to record a five-second buffer for testing purposes.

While the interface of AudioWorker is intentionally simple, this module is where the main usage of Portaudio lies. In order to get through audio with the library, we have to register an audio callback. This is a function with a signature specified by Portaudio, which will be called by the engine whenever the operating system makes a call for more audio. It is in this function that time is crucial, as it is called regularly by the operating system's interrupt handler. Therefore, certain operations are forbidden, such as file IO, mutex operations, or even most other Portaudio API calls. We deal with these issues by keeping the audio callback itself as clean and simple as possible: we read or write from a buffer in memory, avoiding locks or file IO operations entirely; and we do only the most essential real-time audio processing (in this case, the numerical computations for the level meter.)

1.2.2.2 Ringbuffer

Keeping work in the callback to a minimum works without an issue while the buffers are small enough to keep entirely in memory (such as in *record_short)*. However, when files get too long to load entirely into memory, or if recording for an indeterminate amount of time, we need to do some extra work to ensure the timeliness of the audio callback. Instead of providing a simple array to the callback, we provide it with a ringbuffer, or circular buffer. This data structure acts as a FIFO-queue, allowing pushes on one end while popping from the other. Its implementation depends on a read and a write pointer cycling through a buffer of fixed size, wrapping around the end of the buffer. This setup, with data being continuously streamed into the same area of memory, makes it particularly appropriate to deal with audio, as it is conceptually easy to understand how audio fills the buffer and leaves it again in a constant stream.

While Portaudio provides a circular buffer implementation, it is not by default exposed and required a lot of internal dependencies to get working. Instead, we opted for TPCircularBuffer, by Michael Tyson.[1] This data structure uses a very basic interface: *TPCircularBufferHead* and *TPCircularBufferTail* returns a pointer to the next data to write and read respectively, and *TPCircularBufferProduce* and *TPCircularBufferConsume* perform the respective write and read operations, freeing the memory as they go along. An important feature of this data structure as it is implemented is that it is thread-safe with one producer and one consumer. This is crucial as it allows both the audio callback to work with the CircularBuffer on one end while at the other end a separate thread performs slower File IO operations.

1.2.2.3 FileIO

This module provides the bridge between our program and the file system. It mainly derives its functionality from the library libsndfile, which allows for a number of different file formats to be processed. Many of the FileIO functions wrap neatly around libsndfile functions, such as *file_open, read_from_file, write_to_file,* and *seek*. Additionally, though, there are two functions, *read_file_threadworker* and *write_file_threadworker* which are ready to act as the second player in the audio callback scheme. These take in an AudioFile (a struct keeping together SNDFILE pointer and SF_INFO struct from libsndfile) and a buffer and will write one to the other (depending on whether we are reading or writing to or from disk). In practice, AudioWorker will initialize a thread with this function, handing it the ringbuffer as its buffer argument. If we're reading from a file, then the AudioFile will be the already opened file on disk; if we're writing, this will be a temporary file opened for writing. Handling the reading and writing to and from file in a different thread, interacting with the audio thread through a thread-safe data structure relaxes the time requirements on the file IO operations. We have no idea how slow these will be in practice, but at least now we have faith in knowing these operations will not hold up the audio callback.

In order to avoid a redundantly high number of write operations to the filesystem, we only trigger a write four times per buffer fillup. This reduces the amount of processor overload.

1.2.2.4 Interface

Due to different development environments in the team we were not able to establish a common framework to develop a GUI. At the time of writing, a GUI in the MacOS environment was being developed (using Cocoa). In lieu of a graphical user interface, we currently have a command line interface, where users initiate a REPL from which to control playback and recording using simple commands and keypresses. It is important that during

[1] A simple, fast circular buffer implementation for audio processing. Retrieved December 09, 2016, from https://github.com/michaeltyson/TPCircularBuffer

this loop the program remains responsive to user input while at the same time not blocking the AudioWorker in their job. This is achieved through an event loop where the program continues to cycle through possible events seeing whether they need attention until the user terminates the program.

We foresee a similar procedure to ensure a responsive GUI while audio processing is taking place.

1.2.2.5 LevelMeter

The level meter is currently implemented by a method computing the root mean square of the current frame in the buffer, and subsequently taking the log of that root mean square. This method gets run within the audio callback, and writes its result to a float variable. Once per event loop this value is polled to see whether the value has changed or not. Currently the LevelMeter is a little glitchy but a few adjustments should get it working smoothly.

1.3 Programming a mixer in Pure Data with live remote control via OSC

1.3.1 Aim of this work

High-level audio programming allows the manipulation of multiple audio signals from multiple sources in parallel through pre-implemented functions applying various effects and processing to the audio signals. The idea behind this work is to mix a live vocal input with four pre-recorded instrumental tracks by creating a simplified digital mixing station able to control playback speed and volume of the master track, and effects such as reverb, low-pass and high-pass filters on the individual tracks. A key feature and challenge is that some of the effect parameters should be controlled remotely via an application on a smartphone.

1.3.2 Short introduction to Pure Data

Pure Data is an open source visual programming language designed to create audio and multimedia works graphically[2]. Its core users are musicians, visual artists, performers, researchers and developers who want to process and generate sound, video, graphics, sensors, etc. Pd was created in the 90s by Miller Puckette who also developed the commercially distributed Max/MSP[3] during his time at IRCAM.

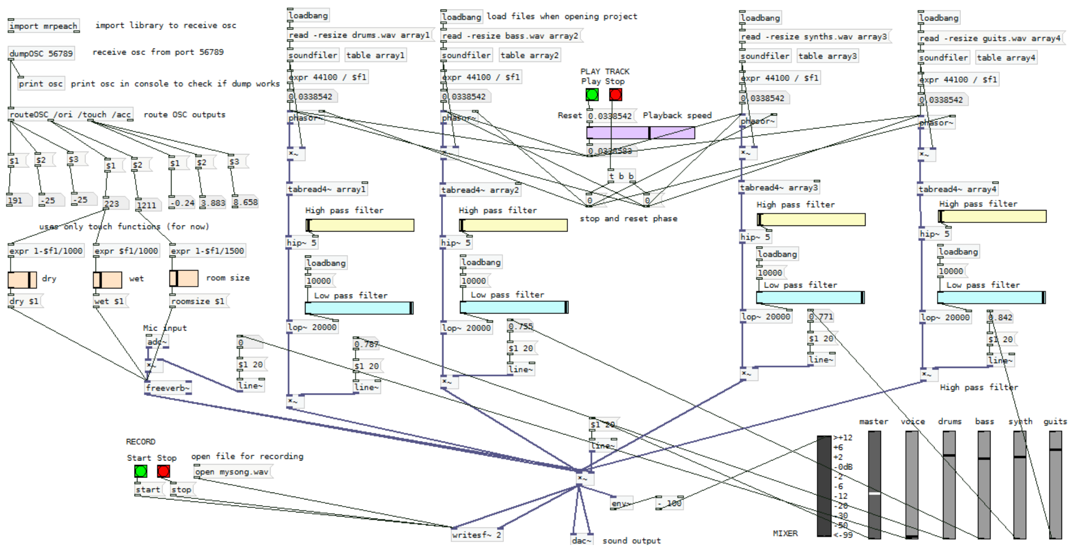
In Pd, algorithmic functions are represented by rectangular boxes called *objects* placed in a window called *canvas*. *Objects* can perform various tasks, from simple mathematical operations to more complicated audio or video processing. A project that can be made of several *canvases* is called a *patch*. A *canvas* can also contain *messages, numbers, symbols* and *comments*. The elements on a *canvas* are linked by connectors called *cords* that convey data from one another.

1.3.3 Overview of the patch

The patch that we created is a mixer controlling five channels. This *patch* is made of a single *canvas* for convenience of visualization. This *canvas* is shown in Figur 1. Four of these channels (named *drums, bass, synth* and *guits*) process instrumental sequences that are read from .wav files. The fifth track (named *voice*) controls a live audio input. The streams for these two types of track, instrumental and live, are explained in more details in the following sections. The *patch* has a *Play/Stop* (green and red buttons in the center part of the *canvas*) function to play the mixed instrumental section, with control of the playback speed (purple slider). A volume

[2] https://Pure Data.info/, accessed December 11, 2016.
[3] https://cycling74.com/products/max/, accessed December 11, 2016.

mixer placed in the bottom right corner allows individual volume control of the five tracks, and control of the master track. A record function is also implemented (green and red buttons in the bottom left corner of the *canvas*), to save the audio signal of the master track in a .wav file.

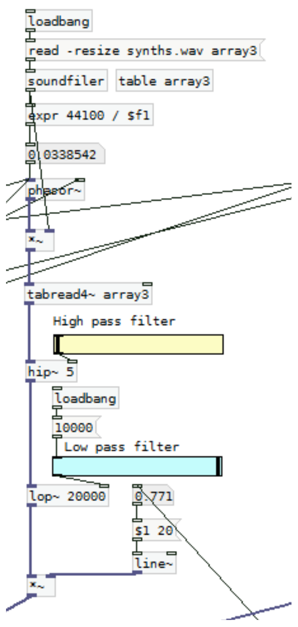


Figur 1: Overview of the Pd patch

1.3.4 Control of the instrumental tracks

The four instrumental tracks have identical streams, shown in Figur 2 and detailed below:

1. The .wav file containing the audio signal is read and stored in a table. A *loadbang* triggers the reading when opening the patch.
2. The default playback speed is computed as the sampling frequency over the number of elements in the table.
3. A sawtooth signal is created with a *phasor~* object, which will be used to play the signal at a constant speed in a loop. The speed, *ie* the slope of the sawtooth function, is controlled globally by the playback-speed purple slider.
4. The array previously stored is read using the sawtooth signal.
5. A high-pass and low-pass filters, respectively the yellow and blue sliders, are applied to the signal.
6. Volume control from the mixer is applied at the end of the stream, using a ramp to avoid clicking sounds caused by abrupt transitions.



Figur 2: Stream for the instrumental tracks

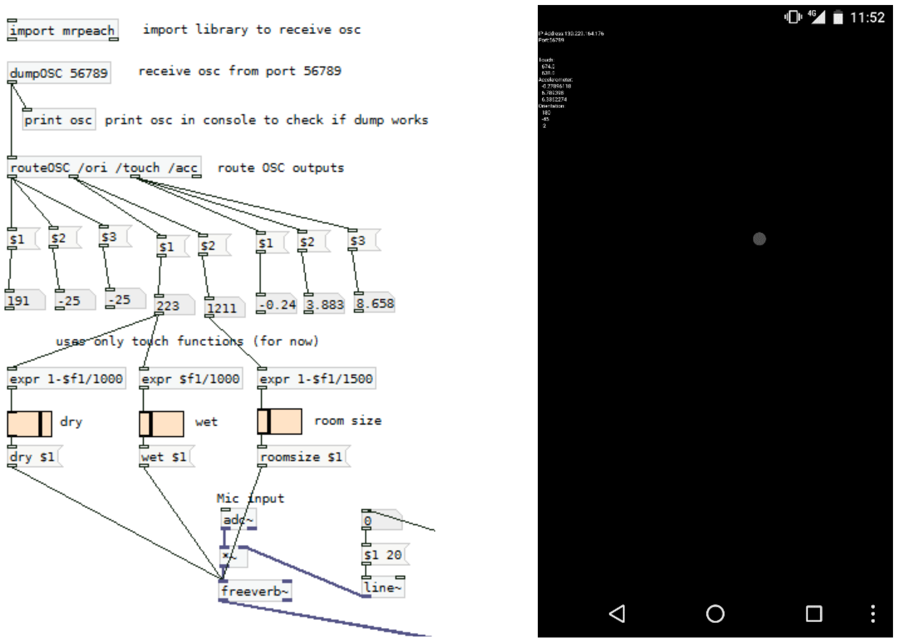
1.3.5 Live track with remote control via OSC

The live vocal track shown in Figur 3 has a specific stream allowing remote control. It reads a live audio input and applies a reverberation effect to it, the parameters of which can be remotely controlled using an OSC protocol via an application on a smartphone. The OSC protocol is a protocol for communication among computers, sound synthesizers and other multimedia devices[4]. An OSC *packet* consists of continuous block of binary data which content can be organized in an OSC *message*. An OSC *message* contains *address patterns* beginning with /. In this patch, an OSC *packet* is sent to the computer using the free application *andOSC* available for Android devices[5]. *andOSC* sends OSC *packets /ori*, */touch* and */acc* using respectively the orientation, a touch pad and the acceleration of the phone. Only the touch parameters used for now, but control via the other functions could easily be added to the *patch*. The touch parameters are used to control the *dry-wet-room size* parameters of the reverb. The horizontal touch direction controls the *dry-wet* balance of the reverb while the vertical touch direction controls the size of the room. The OSC message is sent from the smartphone to the computer IP via a specific port, then retrieved and routed in Pd using the *dumpOSC* and *routeOSC objects* of the *mrpeach* library[6].

[4] http://opensoundcontrol.org/introduction-osc, accessed December 12, 2016.

[5] https://play.google.com/store/apps/details?id=cc.primevision.andosc&hl=en, accessed December 12, 2016

[6] https://puredata.info/downloads/mrpeach, accessed December 12, 2016.



Figur 3: Stream for the live vocal track, left, and screenshot of the andOSC application on an Android device, right

1.4 Possible improvements and conclusion

In the low-level programming task, the challenges of real-time audio programming became readily apparent. While the overall system architecture is quite clean, we have to be constantly careful to not overstep the bounds imposed by the audio callbacks and by threading. Currently our system has a best-case scenario of working but some edge conditions still need to be dealt with (eg. Overwriting files etc.) and a clean GUI still needs to be hooked up to the functionality. However, with the main functionality in place we believe these aspects will fall in line soon.

The PD patch is definitely more robust and demonstrates how technology can enable us to deal with sound differently from a user's perspective. We would like to extend this patch to drive different kinds of effects using OSC, and employ these effects creatively.

1.5 Code

All the code can be found at https://github.com/jonathanadam/AudioTechProject .

1.6 Bibliography

AndOSC. Retrieved December 12, 2016.
https://play.google.com/store/apps/details?id=cc.primevision.andosc&hl=en,
A simple, fast circular buffer implementation for audio processing. Retrieved December 09, 2016. https://github.com/michaeltyson/TPCircularBuffer
Introduction to OSC. Retrieved December 12, 2016.
http://opensoundcontrol.org/introduction-osc, accessed December 12, 2016.
Max/MSP. Retrieved December 11 2016. https://cycling74.com/products/max/
Mr Peach. Retrieved December 12 2016. https://puredata.info/downloads/mrpeach
PureData. Retrieved December 11, 2016.https://Pure Data.info/