15-323/15-623 Spring 2019

Project 4. Real-Time Audio Processing

Due: April 2

Last updated: 6 March 2019

1 Overview

In this project, you will create a program that performs real-time audio generation. There are three levels of tasks:

1. Audio processing only.
2. Audio processing only, limited real-time control through the command line.
3. Audio processing with control via O2 (using a TouchOSC interface).

15-623 students should implement level 3 and 15-323 students should implement at least level 2. Each level lower than your expected level reduces your grade by 10% (i.e. a letter grade).

The audio generation task is to explore one-liners, such as:

```
(t*(((t>>12)|(t>>8))&(63&(t>>4))))
```

which, for t = 0, 1, 2, …, generates a sequence of samples that is quite interesting and complex as audio.

2 Getting Started

- Download PortAudio from http://www.portaudio.com/download.html. Get pa_stable_v190600_20161030.tgz
- If you do not have a C compiler, download either Xcode (OS X) or Visual Studio Community 2017 (Windows). These are BIG, so plan to spend some time where you have a high-speed connection.
- Install PortAudio; build and run some test programs. In particular, test/patest_mono.c is a good example to start with: It is called "mono" because it does nothing but generate a mono (1-channel) signal.
- If you are implementing level 2 or 3, get O2 (`https://github.com/rbdannenberg/o2`), a library that implements the O2 protocol. You will have to build it.

3 Software Development Hints

This may be the first time you have compiled a C program without extensive scaffolding and support. Welcome to the real world! Some things to be aware of are:

- You will probably be compiling several modules: portaudio, o2, and your code. *Each* of these is a separate "project" (Visual Studio) or "target" (XCode), although you *might* be able to obtain pre-compiled libraries for portaudio.
- Portaudio and o2 are *libraries* – bundles of compiled code that must be *linked* with your code to make a complete program. A common error is to compile your code and discover *many* undefined functions – in that case you need to find the library in which those functions are defined and add the library to your project/target. Linking means the

compiler combines all the separate bundles of code – libraries and object files – so that all the references to external functions are resolved and a single big bundle, the executable, is produced and can be run. On Windows, the executable is a ".exe" file. On Unix – linux or OS X – executables do not have extensions.)

- Compilers typically generate any number of variants: 32-bit (e.g. Intel 386 architecture) code vs 64-bit (e.g. x64 architecture) code, debugging vs release (optimized) code, code where libraries are dynamically linked (loaded from a separate place at run time) or statically linked (included in the executable), etc. A common problem is producing a library using one set of options, e.g. 64-bit DLL, and trying to link with a main program compiled with a different set of options, e.g. 32-bit static. Although most people tend to "hack away," ignore warnings, and claim victory without knowing why if and when things finally work, I have only found a few workable solutions:
    - Ignore libraries, the "brute force" approach. Add all the source files into one monolithic project and compile it. You'll be sure to have a consistent compilation and you'll be able to use the "source code" debugger, but it's a pain to figure out exactly what source files to use, and Portaudio in particular has many many files.
    - When in doubt, recompile everything from source code in a single Visual Studio solution or XCode project. *Very carefully* run through all the settings for architecture, dynamic vs. static, c-runtime options, etc. and make them all match.
    - Learn to use CMake. CMake is a language that will create Visual Studio and XCode project files from fairly high-level descriptions of what sources, libraries, and targets you want to generate. Since it specifies all the details consistently, you can systematically eliminate configuration problems by editing the CMakeLists.txt file and regenerating projects rather than manually examining hundreds of parameters through a visual interface.

3.1 xcode Hints

Let's assume you build and install a portaudio library. One approach for the project is to start with a portaudio test program that's already set up to compile and run. Modify it to meet the project requirements.

But instead, let's assume you want to make a new Xcode project for your project. Let's also assume you built and installed portaudio. You will have to tell xcode about 3 things:

- Where to find portaudio header files (.h files, included in your project .c or .cpp files).
- Where to look for the library, which has the actual code.
- What is the name of the library?

In Xcode, select the "Reveal in Project Navigator" item in the Navigate menu. Click on your project (which has a little Xcode icon followed by a project name) and "Build Settings" near the top. In the center panel that appears, click the project again, and to the right of that, select the "All" and "Combined" tabs.

- Look for "Other Linker Flags" and add –lportaudio to the list of flags. This tells xcode to link with the libportaudio.a or libportaudio.dylib library.
- Look for "Header Search Paths" and add something *that will depend on the actual*

path on your computer to the portaudio include folder. This is where to look for portaudio.h.

- Look for “Library Search Paths” and add “/usr/local/lib” – *this is probably correct, but make sure it is the folder containing* libportaudio.a or libportaudio.dylib. This is the directory where xcode will look for one of these files.

If you are using o2, follow similar procedures to add –lo2_static to “Other Linker Flags”, “/Users/rbd/o2/src” (adjusted to the correct local directory for o2) to “Header Search Paths”, and “/Users/rbd/Release/o2_static” and/or “/Users/rbd/Debug/o2_static” (adjusted to the correct local directory) to “Library Search Paths”.

3.2 Microsoft Visual Studio Community 2017 Hints

First of all, many developers on Windows use MinGW (Minimalist GNU for Windows) which offers linux-like command-line tools on Windows.

Nevertheless, I use Visual Studio because it offers nice debugging tools and I think most Windows developers use it rather than MinGW.

If you use Visual Studio, the *best* way to use it (IMO) is to use CMake. CMake takes descriptions of projects and generates Visual Studio “solution” files you can open and use with Visual Studio. In most cases, CMake generates consistent projects and avoids easy-to-make mistakes like adding a search path to the Debug version but not to the Release version, or configuring a 32-bit executable when you meant to build a 64-bit executable.

However, CMake is yet another language and takes some learning. For just one small project like this one, it’s probably not worth the effort to learn.

With all that in mind, here are a few tips for making a Visual Studio project:

3.2.1 Making the PortAudio library (probably a .dll)

First, make a new project. You’ll also need to build portaudio and o2. For portaudio, I built it like this. The main thing here is that if you compile with ASIO support (a proprietary audio API), your executable will require the ASIO library which you probably do not have. Therefore, do not compile with ASIO support…)

- Open portaudio.sln in portaudio\build\msvc
- Under portaudio : “Source Files” : hostapi : ASIO, right click pa_asio.cpp and select “Delete”
- Under portaudio : “Source Files” : hostapi : ASIO, right click each “asio*.cpp” file and right click and “Delete”
- Under portaudio : “Resource Files, open portaudio.def and delete the 4 lines with PaAsio, e.g. delete the line “PaAsio_GetAvailableBufferSizes @50.”
- At the top of the window, select Debug (not Release) and x64 (not x86); then type

F7 (or the Build : Build Solution menu item).

3.2.2 Making the O2 static library

For Level 3, you will also need to build o2. For this, you use CMake (an application to download from https://cmake.org/download). Open o2/CmakeLists.txt in CMake and create a Visual Studio "solution" file. Then open the o2.sln file with Visual Studio and build the BUILD_ALL project.

3.2.3 Making your application

In Visual Studio, you choose to make a Debug or Release version, and an x86 (32-bit) or x64 (64-bit) version. Choose Debug and x64 in the little white boxes with text near the top of the screen. (Not the Debug menu!) On the right, in the "Solution Explorer," select the project, right click, and select "Properties" to add libraries to the project.

Under Configuration Properties : General, click the option for "Configuration Type" and select "Static library (.lib)".

Under Configuration Properties : C/C++ : General, click the option for "Additional Include Directories" and select "<Edit…>" and add directories to the box at the top by clicking on the little folder icon. You should end up with include folders for the two libraries. In my project, I have "C:\Users\Roger\pa\portaudio\include" and "C:\Users\Roger\o2\src" but of course you need to change these to your computer's actual directories with portaudio.h and o2.h, respectively.

Next, under Configuration Properties : Linker : General, click the option for "Additional Library Directories" and add the directories where you have portaudio and o2 libraries. In my case, these are "C:\Users\Roger\pa\portaudio\build\msvc\x64\Debug", and "C:\Users\Roger\o2\Release".

Finally, under Configuration Properties : Linker : Input, click the option for "Additional Dependencies" and add library names. In my case, these are portaudio.lib, and o2_static.lib.

Now you can build your application with F7 or the Build : Build Solution menu item or right click on the project in the Solution Explorer at right and select "Build."

If you build without errors, you can run using "Local Windows Debugger" (top center of the VS window) or right click on the project in the Solution Explorer at right and select "Debug : Start new instance." (Editorial: The fact that there are not only two very different ways to run the debugger but two very different *names* to run the debugger is a good indication that Microsoft has no clue how to design interfaces.)

4 One-Liner Audio Expressions

You probably know of pseudo-random number generators that create long sequences of seemingly random numbers. A little thought should convince you that if the state of these algorithms is N bits, then the generated sequence cannot be longer than 2^N. Thus, repetition will occur even if it takes eons. On the other hand, if the state is small, say N=20, then at audio rates of around 44,100 Hz, the maximum sequence length is $2^{20}/44100 = 23$ seconds. Furthermore, really poor random-number generators can have internal periodicity so that the output does not act like random "noise." Putting all this together, we can design really bad pseudo-random number generators that, when turned into audio, contain interesting non-random structure that can be perceived as pitch and rhythm. The challenge then is to design "really bad" algorithms that sound as interesting as possible. It's a perverse pursuit, but some amazing one-liners have been created. You read more at http://countercomplex.blogspot.com/2011/10/algorithmic-symphonies-from-one-line-of.html, and there are more links there that you can follow or search.

For this project, you must implement one particular one-liner:

```
(t*(((t>>12)|(t>>8))&(63&(t>>4))))
```

and play it in mono or stereo (by duplicating the same sample to right and left channels) with an 8000Hz sample rate. What is `t`? Remember that `t` is an increasing integer sequence. You will be generating blocks of samples. Let's say the portaudio callback asks for 64 samples each time it is called. Then the first callback will evaluate the expression with `t` ranging from 0 to 63. But the *next call must continue the sequence* with `t` ranging from 64 to 127. Therefore, you'll need to remember the previous `t`, either in a global variable or better yet, in a structure that is passed as a parameter of the callback – portaudio lets you do that.

One thing not too clear in the descriptions of these one-liners is that the output uses 8-bit samples, so the high-order bits are ignored. Since you will probably want to write floating point samples in Portaudio, you can add some code to convert the low-order 8 bits to floats as follows:

```
long sample = (t*(((t>>12)|(t>>8))&(63&(t>>4))));
sample &= 0xFF; // take only 8 bits
// convert to float and offset so 128 becomes 0.0:
float fsample = sample * (1.0F / 256.0F) - 0.5F;
*out++ = fsample;
// if output is interleaved stereo use the next line
//   to copy the left to the right channel:
// *out++ = fsample;
```

5 Requirements

- Your program should produce audio output using a "one-liner" as described in the previous section.
- You should output mono or stereo at a sample rate of 8000Hz (otherwise, the starting one-liner will not sound "right").
- The default "one-liner" should be the one described above; your program should begin playing this one immediately.
- To keep things simple, it is preferable to process only one channel out, but stereo is allowed. More than 2 channels are not allowed (to simplify equipment issues for the graders – feel free to experiment with many channels on your own.)
- Your program should implement at least 3 different "one-liner" algorithms, selectable using command line or TouchOSC control.
- Your program should exhibit some additional control over audio generation. Control can include changing parameters in the one-liners, changing volume (you can add a separate line of code to scale your output samples by a value from 0.0 to 1.0), panning (if you use stereo; panning can be simply scaling the left channel by 0 to 1 while scaling the right

channel by 1 to 0), and reverse (just cause the parameter t to decrement instead of increment). If you do not implement TouchOSC control, you can read control values as lines of text from the console.

- For extra credit, you can also add an audio effect with controls. For example, a *delay* can be added by putting each sample *x* into a FIFO queue (circular buffer) of length 4000 (more or less), then take the output of the FIFO, which is a delayed copy of the signal, scale it by 0.5 (more or less), and add it to *x* to form the output sample. Or *echo* can be created by with the same algorithm except put the output sample into the FIFO instead of *x*. You can also implement digital filters or other effects if you have some DSP knowledge.
- Your program should include a "stop" control that cleanly shuts down audio processing, and when the audio stream has stopped, exit the program. (Note that simply calling exit(0) in C will shut down the process, but the audio thread will probably encounter an exception and crash, causing the OS to clean up the mess – it's not harmful, but it is likely to cause some audio glitch and does not satisfy the "cleanly shuts down" part of this requirement.)
- You should produce and hand in both source code and an executable binary (plus any special dynamic libraries needed to support it). Depending on your development environment, you should also hand in whatever is needed to compile (e.g. IDE project files, Makefile, etc.).
- Also hand in documentation of what you implemented and how to build and run it.

6 Control

You should get your program working without any control to begin with. Then add controls one at a time.

6.1 Level 2 Control

For a level 2 project, you only need to add control from the command line. E.g. the main program can be a loop that prompts for one or more parameters, reads them (see scanf()) and stores them in a way that affects the audio computation. Note that the main program will be running in a separate thread from the audio processing, so it is OK for the main program to block on input. If you pass parameters by updating shared global variables (recommended), you should ask yourself if this is a safe use of shared memory. The answer is probably yes: when the main program writes a float or a double, the write is atomic, so the reader (the PortAudio thread) will read either the old value or the new value, but never some combination of bits from both. The worst thing that could happen is that audio processing could interrupt the main process while parameters are being written, perhaps receiving the first parameter change but not the second one (in the case of multiple parameters). You should ask yourself if this is a problem or not.

6.2 Level 3 Control

For a level 3 project, you should use the main thread to process O2 messages. These messages can then write to global variables as described in the previous paragraph. Do not worry about atomically dispatching multiple parameters or bundles of O2 messages. This would require either locks (which might lead to priority inversion problems) or processing of O2 messages in the PortAudio callback thread (which might add significant processing, or might result in calls to malloc – which in turn uses locks, which again might lead to a priority inversion problem).

7 Grading Criteria:

1) Correctness: Does the program compile, run, and produce the desired behavior?

2) Modularity: Your program should isolate different concerns. In particular, you should clearly separate and label code for different threads, preferably in different files.

3) Programming style: Code should be clearly written and commented to optimize readability (include high-level design and specifications at the top of the file or in a separate document, give concise comments within the code, avoid verbose or redundant inline comments.) No tabs in code. No very long lines.

4) Instructions: A 1- or 2-page instruction manual might be about right for this project. At least we want to be able to read what it does, how to use it, *what system it was compiled for*, and how it works. Don't write a book, but don't leave graders in the dark.

5) 10% off for each level below the assigned level.

8 Hand-in Instructions:

You should put all your work, Source files and Documentation, into a zip file and submit it through Autolab. (If there is a file size problem, tell us on Piazza.)
Be sure that you include:

- Instructions on how to run and demonstrate everything you have implemented.
- What does your audio processor do?
- The specific control interface to use – either command line commands, a built-in control panel from TouchOSC (and tell us what device you used to run TouchOSC), DontTouchOSC, or a custom interface (submit the file and tell us the file name in the instructions).
- Include the OSC port number for TouchOSC (ideally, make it 8000).