# Building Audio Effect Plug-ins

Most of the chapters in this book present the theory and implementation of the major types of audio effects, examining the mathematical principles behind each effect. This chapter describes how to put these principles into practice by creating an audio plug-in.

Audio plug-ins are the most common way of implementing audio effects in software. A typical plug-in is a self-contained block of code which is compiled to run on a particular processor and operating system, and can be used within an audio software environment. This chapter will examine the process of creating plug-ins using the *JUCE* (*Jules’ Utility Class Extensions*) programming framework, which can be used to create plug-ins for many different software platforms.

## Audio plug-in basics

Audio plug-ins are usually designed for use within digital audio workstations (DAWs). To ensure compatibility across different DAWs, several industry-standard plug-in formats have been developed. These include Steinberg’s *VST* (Virtual Studio Technology) format, widely supported in nearly all professional audio software; Apple’s *AudioUnit* format, supported on most Mac programs; and the *AAX* (Avid Audio Extension) format by Digidesign/Avid. Each format provides similar functionality, typically including ways of passing audio into and out of the plug-in, negotiating sample rates and number of channels, and querying and setting user-adjustable parameters for the effect.

### Programming language

The most common programming language for writing audio plug-ins, by far, is C++, though other languages may be used. The code will be compiled to run on a specific processor and operating system. However, the same code can typically be compiled to run on any hardware and operating system, as long as the code does not use any OS-specific functionality.

This book is not intended to cover programming fundamentals, or as an introduction to C++. Examples in the remainder of this chapter will be presented in C++ with the assumption that the reader has a basic familiarity (though not necessarily significant expertise) with the language. Excellent introductions to C++ programming can be found in many sources, including [94-96]. An audio-focused introduction to C++ is included in [97], which also goes into detail on many aspects of audio programming not covered in this text.

### Plug-in properties

The essential task of an audio plug-in is to receive an input audio signal, apply an effect to it based on some control parameters, and produce an output audio signal. In some cases, including virtual instrument or synthesizer plug-ins, no audio input is used, and the output may be produced in response to MIDI (Musical Instrument Digital Interface) messages. However, the effects in this book all assume an input and an output.

Audio effect plug-ins process audio. They receive digital audio and process it through to their outputs. Their function is similar to that of hardware audio processors They can also be chained together. That is, most hosts will allow the audio output from one effect to be used as input to another effect.

In contrast, virtual instrument plugins generate audio. They can act as standalone software synthesizers, samplers, or digital musical instruments. They typically use MIDI messages to control instrument parameters. Some effect plug-ins also accept MIDI input. In fact, the use of MIDI for control is not the distinguishing factor between effects and instruments. Think of the instruments as being similar to the effects, but without the need for digital audio input.

There are also other types of audio plug-ins. Monitoring effects are a special type of effect that provides feedback about an input signal without processing audio. Typically, they present a visualisation of the input signal, like a level meter. Think of these as having digital audio input, but no need for any output signal.

Finally, its worth mentioning MIDI effects, which do not really operate on audio at all. They have MIDI input and MIDI output. So they could, for instance, apply pitch shift on a MIDI data stream. The output MIDI messages could then be sent to other virtual instruments or hardware devices.

Our focus is on just the audio effect plug-ins. Important properties of such a plug-in include the number of channels it supports and the allowable sample rates. Many effects can operate with different numbers of channels, but others will require specific channel configurations. For example, a stereo panning effect would need at least two output channels, but it could take one or two input channels for stereo or mono input). Some effects may have restrictions on the sample rates they support, though it is useful wherever possible to write plug-ins that operate at any sample rate. Plug-ins will also define one or more user-adjustable parameters which can typically be changed either through a standard interface provided by the DAW, or by a custom GUI created by the plug-in author.

### The plug-in host

A plug-in host is a software application or hardware device in which the plugins run. It typically presents plugin user interfaces and routes digital audio and MIDI to and from plugins.

The most well-known form of host for audio effect plug-ins is the digital audio workstation, or DAW. These are applications and devices used for a wide range of audio production tasks, such as recording, editing and rendering multitrack music or soundtracks. However, audio effect plug-ins might also be used in a stand-alone host which lacks some DAW features but has been optimized for use in live performance. They might also be used in video editing tools, or in game engines.

### The JUCE framework

The example code for this book uses the JUCE (Jules' Utility Class Extensions) environment, created by Julian Storer, which provides a cross-platform, multi-format method for building audio plug-ins. JUCE was created in 2004 by Jules Storer. It is an open source cross-platform C++ codebase for developing desktop and mobile applications, and plug-ins. It is used to write software so that it will run with the same user experience on many platforms (Windows, macOS, Linux, iOS, Android…) and in many formats. JUCE furthermore provides a cross-platform set of graphical user interface (GUI) controls and a very large library of useful C++ classes covering commonly used features by apps and plug-ins, such as graphics, audio, XML parsing, networking, cryptography, multi-threading… Thus, it also reduces the number of third-party libraries needed in a project.

But the real reason why we focus on JUCE here is that it is the *most widely used framework for audio application and plug-in development.* As mentioned, JUCE has a very large number of built-in libraries, but this is especially true for those related toaudio functionality. JUCE has support for audio devices (CoreAudio, JACK, DirectSound) and MIDI playback, DSP building blocks, polyphonic synthesizers…, and built-in readers for common audio file formats (WAV, AIFF, FLAC, MP3, Vorbis…). It also comes with wrapper classes for building all major audio plugin formats. Since all platform and format-specific code is contained in the wrapper, one can build for almost any plug-in format & platform from a single codebase.

JUCE is free for use in open-source projects. Documentation and download links can be found on its website: <http://www.juce.com> .

### VST – Virtual Studio Technology

The most popular audio plugin format is VST, which stands for Virtual Studio Technology. VST was created by Steinberg (now owned by Yamaha) in 1996. VST is an audio plug-in software interface. That is, it is an audio plugin standard that allows virtual instruments and effects (and more, as we will see) to be integrated into digital audio workstations. It is the main industry standard for audio plug-ins.

VST plug-ins usually run within digital audio workstation (DAW), though stand-alone VST plug-in hosts also exist. VST plug-ins are intended to provide additional functionality to the host, and they often use digital signal processing to simulate traditional recording studio hardware. VSTs usually have graphical user interfaces that display controls similar to physical switches and knobs on audio hardware.

Most VSTs are either instruments (VSTi) or effects (VSTfx), though other categories exist, esuch as the monitoring effects mentioned previously.

## Theory of operation

JUCE audio plug-ins are divided into two components: a processor that handles the audio calculations and an editor or GUI that lets the user interact with the plug-in. The processor provides several functions: a callback function that the DAW calls every time it needs a new block of audio samples; methods for getting and setting effect parameters; and initialization and cleanup routines. The editor provides graphical controls for the user to see and change the parameters. Most DAWs will provide a generic editor when the plug-in does not define its own.

### Callback function

The most important task for an audio plug-in is receiving and processing audio samples. How does the plug-in know how many samples to process, and when to process them? If the effect is operating in real time, it is clearly impossible to wait for the entire audio signal to arrive before applying the effect. Instead, audio needs to be processed in small blocks or buffers of samples as it comes in. To receive blocks of samples, plug-ins implement a callback function, a function which the DAW calls every time it has new audio to process. Thus, it is always the DAW, and not the plug-in, which determines how many audio samples to process and when. The advantage of this arrangement is that the plug-in author never needs to be concerned with where the audio samples come from, when they should arrive, or where they go after the effect has been applied. The author simply needs to write a callback function which processes as many samples as requested by the DAW.

When the DAW runs the callback function, it will provide several pieces of information. These include the buffer size (how many audio samples to process), the sample rate, the number of input and output channels, and a buffer (region of memory) containing the input audio. The host will also provide a buffer in which the audio output should be stored. In JUCE (as in many plug-in formats), the plug-in is expected to put its output in the same buffer where the input samples were found.

When the callback function finishes, it returns control to the DAW, which decides what should happen to the processed samples. The callback function will be called again when there are more samples to process. The buffer size used by the DAW partly determines the overall latency (delay) from input to output; smaller buffer sizes produce a low delay but increase the risk of underruns (gaps) if the computer cannot respond quickly enough. A typical conservative buffer size would be 512 samples; a high-performance buffer size might be as small as 32 samples. At a 44.1kHz sample rate, a buffer size of 32 means that the callback function would run over 1300 times per second!

### Managing parameters

Nearly every audio effect will have one or more user-adjustable parameters. For example, a parametric equalizer plug-in might let the user change the center frequency, the gain and the Q of the filter. Most plug-ins provide a user interface to allow the user to adjust parameters. Every operating system and plug-in format provides different routines for managing a GUI, but there are several common requirements. In particular, the plug-in must provide a set of functions for the user to see or change the parameters, and there must be a way for the callback function to discover the current parameter values.

Figure .. Basic components of a JUCE audio plugin and their relationship to the digital audio workstation (DAW).

JUCE provides a convenient class for almost any parameter that the plug-in would expose to the DAW, the audioProcessorParameter. JUCE also provides several methods (C++ object functions) related to managing parameters, described in detail in the example in the next section. Two of the important methods are getParameterTree() and setParameterTree() within the audio processor object (see Figure 1.6). When the DAW calls getParameterTree(), the group of parameters managed by that audio processor is returned. Parameter values are most often in the form of floating-point numbers (C++ type float). It is up to the plug-in to define what they mean (for example, a parametric equalizer might have frequency as parameter 0, gain as parameter 1 and Q as parameter 2). With setParameterTree(), the group of parameters managed by the audio processor can be set. The host provides an index of the parameter to set and a floating-point value it should be set to. The plug-in stores this information so the audio callback function can access it later.

In JUCE, the plug-in keeps track of its current parameters by using instance variables, variables declared inside a C++ class which are accessible to any of its methods. When the DAW calls getParameterTree(), the current values of the relevant instance variables are returned; when it calls setParameterTree(), the values of the instance variable are changed. The callback function, processBlock(), accesses the values of these variables to discover the current parameter settings.

### Initialization and cleanup

Before a plug-in can process audio, certain initialization tasks must be performed. For example, in a delay plug-in, memory for delay buffers might need to be allocated; for an equalizer plug-in, filter coefficients might need to be calculated and internal variables holding previous samples might need to be initialized to 0. For plug-ins with low-frequency oscillators, the phase of the oscillator may need to be initialized.

Every plug-in format will provide a method for initializing the plug-in. In JUCE, basic initialization can be performed in the constructor of the audio processor object, which runs once when the plug-in is first loaded. JUCE also provides a method called prepareToPlay() which runs immediately before audio processing begins. By allocating resources just before the audio starts, the plug-in only uses resources while actively running.

All resources that are allocated by the plug-in will eventually need to be released. JUCE provides a method called releaseResources() which runs immediately after the host stops processing audio. This method should be used to free any resources allocated in prepareToPlay(). It is safe to assume that the audio callback function will never run after a call to releaseResources(). Similarly, the counterpart to the C++ constructor is the destructor, which runs once when the user removes the plug-in from the host environment. Anything that is allocated in the constructor should be freed in the destructor.

### Preserving state

Each time the DAW calls the plug-in’s callback function, it will provide only a small block of input samples to be processed. The plug-in often depends on previous state information to know how to process these samples. For example, phases of oscillators, pointers within circular buffers, and previous values of input and output samples may be needed to calculate the output. It is up to the plug-in to save any state that it needs during the callback function.

To understand the importance of managing state, consider a simple effect where the current output is equal to the sum of the last two inputs: y[n] = x[n] + x[n-1]. Suppose that the host requests 512 samples beginning at sample N. It will therefore supply a buffer containing 512 input samples x[N] to x[N+511] and require 512 output samples y[N] to y[N+511].

At first glance, it may seem as if the callback function has enough information to calculate the output without any reference to what has happened before. But what about calculating y[N], the very first sample in the buffer? We have y[N] = x[N] + x[N-1], but x[N-1] was supplied last time the callback was run, and that value is no longer available. The plug-in must therefore use a separate instance variable to remember what has come before. Here, a single float for each channel would be needed to remember the previous sample. The float should be declared inside the class, so its value is preserved across calls to the callback function.

## Set up

This section briefly covers how to set up the tools to begin audio plug-in development.

### Required software

To build the plug-ins that come with the book, the following software is required:

1. The JUCE (Jules’ Utility Class Extensions) C++ library by Julian Storer. JUCE runs on nearly every platform, including Mac, Windows and Linux, and can be downloaded free from http://www.juce.com.
2. An Integrated Development Environment, or IDE, with a C++ compiler. This is an application with lots of tools for software development, like a code editor and a debugger. Common ones include Visual Studio for Windows, Xcode for MacOS and iOS, Android Studio, and Code::Blocks. They are all generally available as free downloads, sometimes after a registration process. Here, we will do everything in Windows with Visual Studio, but the steps should be similar with different development environments and operating systems.
3. On the Mac, two other pieces are required: Xcode Audio Tools, which can be installed from within Xcode and the Core Audio Utility Classes, which can be found on Apple’s website. Platform-specific setup instructions often change over time; please consult the companion website for this book to find the most recent instructions.
4. To build VST plug-ins, the VST3 SDK (2.x versions are still supported but no longer recommended) is required. This is bundled with JUCE. Optionally on Mac, AudioUnits can be created instead. Whether you use VST or AudioUnits, your program in JUCE will be written identically.
5. A suitable digital audio workstation (DAW) or other host environment in which to test the plug-ins. JUCE ships with a VST plug-in host, which is sufficient for the sort of testing we need. On the Mac, the Xcode Audio Tools provide the AU Lab program which is a simple, lightweight way of testing AudioUnit plug-ins. Several other cross-platform options are available, including Audacity and Reaper.

### Download the tools

If you do not have one already installed, you will need to download an IDE. When installing, you should be able to do a basic installation, needing only C++.

Now download JUCE from <https://juce.com/download/> . Unpack the JUCE folder and put it in some convenient location on your computer.

### Setting up the ProJucer

In JUCE, new projects are configured using a program called the Projucer. The Projucer is JUCE’s tool for creating and managing JUCE projects. Once files and settings for a JUCE project are specified, it automatically generates 3rd-party project files to allow project to compile natively on each target platform; Visual Studio, Xcode, Android Studio, CodeBlocks, Linux Makefiles… The Projucer also has a code editor, integrated GUI editor, and wizards for creating new projects and files.

Depending where you download JUCE from, it may or may not come with a pre-built version of the Projucer. If not, then the Projucer needs to be compiled and built on your system before anything further can take place. Within the main JUCE folder, Projucer project files for each of the major development environments (Visual Studio, Xcode, etc.) can be found in [extras/Projucer/Builds](https://github.com/juce-framework/JUCE/blob/master/extras/Projucer/Builds). Open the project in your development environment and build it. It should compile without errors if JUCE was correctly installed. Run the Projucer once it finishes compiling.

A common problem is that the JUCE libraries, or modules, are not easily found. The steps to set their location are shown in Figure 1, and are as follows. First, the Global paths may need to be set in the Projucer app. Navigate to the menu item File -> Global Paths on Windows or Linux, or *Projucer -> Global Paths* on MacOS. Open global paths and set them to the right locations (1), if they are not set there by default. Now close the global paths window. Then, click in the *Modules* section on the left (2). Click on the cog wheel and scroll down (3). Select *Enable/disable global path for modules...* and ensure that the Global Path is used for all modules (4).

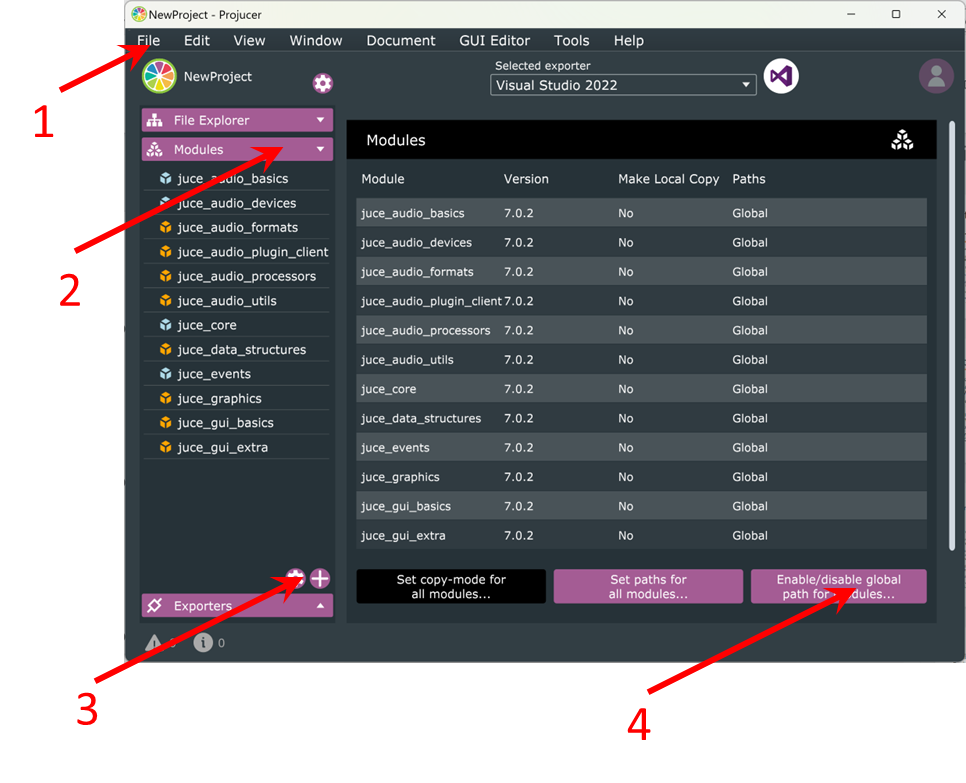


Figure . The steps in the Projucer for setting the global path for modules.

The Projucer can now be used to create new JUCE projects, view tutorials, run examples and much more. It is also possible to include the JUCE modules source code in an existing project directly, or build them into a static or dynamic library which can be linked into a project.

### Opening example plug-ins

This book comes with several example audio plug-ins which can be opened in the IDE or your choice. Within each plug-in directory, look in the Builds directory for a version specific to your IDE. If you do not find one, open the .jucer file in the Projucer and create a new target for your platform. Following the instructions in this chapter, you should be able to compile the plug-in and get it running in your audio host environment.

## Example: building a gain effect in JUCE

This section will describe the implementation of a simple plug-in in the JUCE environment. We will take as an example a basic gain effect, with user-adjustable control for a gain parameter that multiplies the incoming signal to produce the outgoing signal. It will demonstrate many of the concepts that have been discussed, including the management of parameters and preservation of state between callbacks.

The complete code for this effect can be found in the materials that accompany the book, along with example code for several other effects.

### Creating a new plug-in in JUCE

Once the Projucer is running, select New Project from the File menu, which brings up the *New Project* window. The Projucer will automatically generate different starter files depending on what you want to develop. Enter a project name (here, we call our project HelloWorld) and set the Project Type to Plug-In ->Basic (see Figure 1.2).

A screenshot of a computer program

Description automatically generated

Figure .. The new project window, for creating a new audio plug-in with the Projucer.

Click Create Project to make the JUCE project for your new plug-in. The Projucer displays a new screen now. On the left there is an accordion (stacked list user interface element), with three headings,

* + File Explorer
  + Modules
  + Exporters

The next step is to make an export target for your specific Integrated Development Environment. By default, the Projucer should create a target for the IDE on your platform. If not, or if you want to make an additional target, right-click within the Exporters section on the left, or click on the plus symbol in that section, to create a new export target (Figure 1.9).

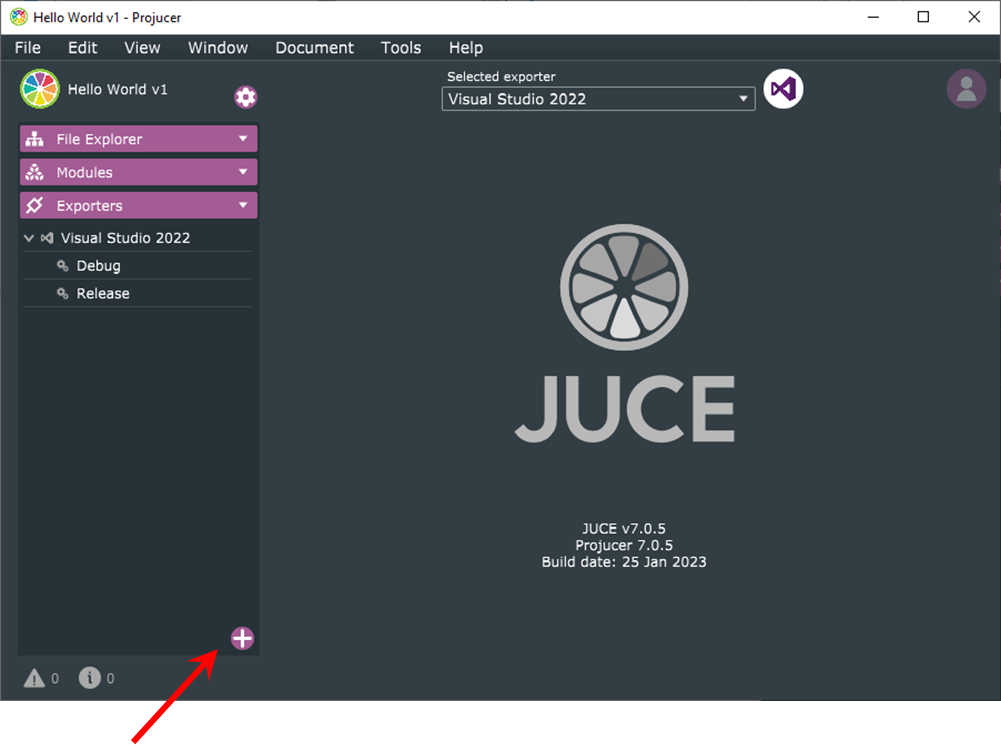


Figure .. Adding an export target in the Project window.

Selecting the export target from the list on the left-hand side of the window brings up settings specific to the platform. Most of these options can be left at their default values. Similarly, module settings can also usually be left at their defaults.

Now click on the cog wheel near the top left corner to get the *Project Settings* window where you can edit the details of the plug-in (Figure 1.4).

A screenshot of a computer

Description automatically generated

Figure .. The Project Settings window.

There are several fields worth mentioning here;

* Project *Name*: the name of the project. This is used for naming various methods and files when the projucer generates code. They can be renamed, but its useful to pick a preferred name at this stage.
* *Project Version*: the version number of your plug-in (the default value is fine for new plug-ins)
* Company Name: the name of your company or organization, as you want it to appear in the audio host environment
* Bundle Identifier: a unique identifier for the plug-in. The format looks like a reversed web address, which identifies the organization and the plug-in name.
* Plugin formats: these boxes select which format of plug-in to build. On most platforms, including Windows, VST3 is the most common format; on Mac, AudioUnit is the usual selection. Selecting those two checkboxes is a common option.
* Plugin Name: the name of your plug-in as you want it to appear in the host environment, usually the same as the Project Name
* Plugin Description: a short description of your plug-in
* Plugin Manufacturer: company name, similar to Company Name above
* Plugin Manufacturer Code: a four-letter code identifying your company or organization. Use a single code consistently across all the plug-ins you develop.
* Plugin Channel Configurations: pairs of numbers which identify valid configurations of input channels and output channels. For example, {1,1}, {2,2} indicates a plug-in that can take mono input and mono output, or stereo input and stereo output. The default is to leave this field blank.

Edit these settings as you wish, but make sure to give the plug-in an appropriate name and check VST3 and/or AU for the plug-in format. The other settings can usually be left at their defaults.

When you have finished entering all the details, click the icon next to *Selected exporter* at the top of the window. This should bring up the project within your IDE, where it can be built using the same procedure you normally use to compile projects. On the Mac, building the project will automatically copy the resulting plug-in to the system plug-ins directory (~/Library/Audio/Plug-Ins/Components/) where it will be found by audio software including AU Lab. On Windows, your DAW may need to be manually set to look for your plug-in. In many cases, restarting the audio host program (AU Lab, Audacity, etc.) is necessary every time the plug-in is recompiled.

### File overview

Juce plug-ins contain four source files by default:

* PluginProcessor.h: the header file for PluginProcessor.cpp; defines the C++ object whose code is filled out in PluginProcessor.cpp. Includes declaration of methods and object variables.
* PluginProcessor.cpp: the main file where the audio processing is done.
* PluginEditor.h: the header file for PluginEditor.cpp, containing the definition of the C++ object for the editor.
* PluginEditor.cpp: contains the code for the graphical user interface for the plug-in.

In your own projects, for more complex effects, you might find occasion to add extra source files to the project. But most effects can be implemented in just these four. We will examine each file in turn. Not all lines of each file are included in this text, so please refer to the accompanying materials for the complete source.

These files contain a lot of auto-generated code, but you don’t need to do anything about most of this code yet. To start we will focus on the PluginProcessor.cpp file, so open up this file in your IDE. This is where most of the audio-related code is located.

Scroll down until you reach the process block.

Delete all the pre-generated code between the brackets.

When JUCE auto-generated the project files, a buffer array was created. This is an array of samples that changes depending on the block size set in your DAW. So if your block size is set to 512 samples, the array will have a length of 512. We will use a for loop to change the data in this array, by iterating through the array and changing the value of the samples.

Now write this code into the process block:

for (int channel = 0; channel < getTotalNumOutputChannels(); ++channel)

{

for (int sample = 0; sample < buffer.getNumSamples(); sample++)

{ }

}

This loops through channels of input data. There could be any number of channels, but the most common situation is dealing with stereo audio, where channel 0 is the left channel and channel 1 is the right channel.

Currently, for every channel, this iterates through the audio buffer but does not change anything. Passing an audio signal as input to the plug-in will produce the same signal at the output.

Now lets use the inner for loop to change the value of each sample in the array.

Lets add a floating point gain variable inside the processBlock, and set its value to something less than 1. We will then multiply every sample by this gain.

**float gainValue = 0.1f;**

for (int channel = 0; channel < getTotalNumOutputChannels(); ++channel)

{

auto\* channelData = buffer.getWritePointer(channel);

for (int sample = 0; sample < buffer.getNumSamples(); sample++)

{

channelData[sample] = **gainValue \*** channelData[sample];

}

}

First, this assigns a variable channelData as the write pointer of the audio buffer. It then multiplies the input by a gain equal to 0.1. The input is written back into channel data. The for loop then increments the input to the next sample, and repeats.

So this code multiplies every sample value in the audio buffer by a gain of 0.1. When this plugin is placed on a track in your DAW it will reduce the level of the incoming signal.

Let’s try this out now in your DAW, first build the project. This will create a VST3 file in the project directory usually in a location like Builds->VisualStudioXX->x64->Debug->VST3. Open your DAW of choice and set this folder as a [VST](https://audioordeal.co.uk/top-free-vsts-of-2019/) location, or move the built file to your VST directory. You should hear that it has applied a gain reduction to any audio used as input to the plug-in.

However, the user has no control over this gain reduction. Lets give the plug-in a user interface.

The GenericAudioProcessorEditor is a user interface component that displays the parameters of an [AudioProcessor](https://docs.juce.com/master/classAudioProcessor.html) as a basic list of sliders, combo boxes and switches. It will also automatically lay out the controls on an interface. Here, we will use it to greatly simplify the user interface design and control., allowing us to focus mainly on the audio-related aspects.

For our plugin to use this generic interface, we replace the line

return new HelloWorldAudioProcessorEditor (\*this);

in the PluginProcessor.cpp file with

return new juce::GenericAudioProcessorEditor(this);

We then open the PluginProcessor.h file, and add this variable to the private section;

juce::AudioParameterFloat\* gainParam;

In the constructor in PluginProcessor.cpp, add a parameter to the audio processor.

addParameter(gainParam

= new juce::AudioParameterFloat("gain","Gain",0.0f,1.0f,0.0f));

where

* *gainParam* references the AudioParameterFloat we created in PluginProcessor.h.
* *gain* is the slider’s ID
* *Gain* will be shown next to the slider in the DAW.
* Float values set minimum, maximum, and default values for the slider

Since we are using the generic interface, it will automatically add a slider to the user interface, which can be used to control this audio parameter. However, the slider doesn’t yet do anything. So let’s modify the code in the processBlock to be

**float gainValue = gainParam->get();**

Build and run it again. Now we have user control of the gain slider, so that the user can change the volume of incoming audio data.

Add the new VST effect to an audio channel and you should see the following results:

A screenshot of a computer

Description automatically generated

Figure . The interface of the HelloWorld plug-in.

Here, we created a basic gain plugin. This isn’t very exciting, but it shows the process of developing and testing audio plugins using JUCE. More complicated plugins are made mainly by modifying the process block and adding more UI elements to further control the effect of these modifications.

### Summary

This section has examined the code for an example plug-in that implements a simple gain control. Four source files were required, two related to the audio processor and two related to the user interface. The most important task for the audio processor is to implement the audio callback function processBlock(), which the system calls every time it needs a new block of audio. Since audio is processed in small blocks, it is necessary to remember the state of the effect between callbacks, so instance variables were declared in PluginProcessor.h to record information that persists beyond the duration of processBlock().

The other important task was the management of effect parameters. Ordinarily, this would require careful coordination between all four source files; parameters to be declared in PluginProcessor.h alongside instance variables, controls to change their values declared in PluginEditor.h, and methods in both PluginProcessor.cpp and PluginEditor.cpp for getting and setting parameters. However, with the use of the generic user interface, the editor files are not needed at all. We can focus just on the audio-related content in the plugin processor files.

## Advanced topics

### Efficiency considerations

For more complex effects, it is useful to consider how the audio processing can be most efficiently implemented. This is especially true for effects on mobile or embedded devices, but computational cost is even a concern for the fastest of computers when many effects are used together. A standard metric of computational cost used in digital signal processing is multiplies per sample: how many multiplications does the computer have to perform for each audio sample? Though it is not a perfect metric of processing complexity, it provides a useful overall guide to how an effect can be made more efficient.

A straightforward example of considering multiplies per sample is filtering. An FIR filter of length *N* will require *N* multiplies per sample to implement; for long filters, this can become quite expensive. A particularly troublesome example is convolutional reverb where the input signal must be convolved with an impulse response that could potentially exceed 100,000 samples. Aside from shortening the filter, a variety of tricks are possible to improve efficiency using the Fast Fourier Transform. These are further discussed in **Error! Reference source not found.**.

Another example of managing computational complexity can be found in the wah-wah effect. Here, the effect is generated by a second order IIR filter whose coefficients change whenever the user adjusts a control. Applying the filter is inexpensive, requiring only 5 multiplications per sample. However, calculating the coefficients involves a complicated formula including trigonometric functions which require many more multiplications to implement. It is not efficient to recalculate the coefficients of the filter at each sample based on the current value of the cutoff frequency. Instead, a more efficient approach will recalculate the coefficients only when they are changed by the user, typically within the setParameter() method, and save these values for future use by the callback. This significantly reduces calculations but has the potential to introduce a subtle bug, discussed in the next section.

A third example is the calculation of low-frequency oscillators (LFOs). In an auto-wah effect, the cutoff frequency of the filter changes continuously over time under the control of an LFO. It might appear that recalculating coefficients every sample is necessary. However, with most LFOs, the change in value from one audio sample to the next is so subtle that the filter coefficients can be updated less frequently, for example every 16 or 32 samples, with no audible change in performance.

A final example concerns the use of complex mathematical functions. Suppose a phaser effect wanted to implement a complicated exponential LFO waveform to vary the location of the allpass filters. Calculating the value of this function each sample might require a large number of multiplies since internally, exponential functions are much more complex to calculate than basic multiplications. However, it might be possible to generate a lookup table with all the values of the function pre-calculated and stored in a block of memory. Then on each sample, rather than running the calculation, the callback function could access the table to determine its value. If greater precision is needed, interpolation between values in the lookup table could be used while still maintaining a significant efficiency improvement over raw calculations. See **Error! Reference source not found.** for further details on interpolating within buffers.

### Thread Safety

Most audio plug-ins are multi-threaded. This means that two different methods could be running simultaneously in different threads of execution, with no guarantee that one method finishes before the other begins. In particular, the thread that runs the audio callback function is often different than the thread that handles changing parameters. This results in the potential for subtle bugs and even outright crashes.

Consider the case of a phase vocoder plug-in which needs to gather samples in a buffer before performing a Fast Fourier Transform. Suppose that the user changes the window size of the effect using one of the graphical controls. When this happens, a call to setParameter() will be generated, and because the window size has changed, the plug-in may reallocate the window buffer to hold a different number of samples. In a multi-threaded environment, the call to setParameter() might happen at the same time as the callback processBlock() is running. The callback function will be unaware that the buffer is about to change size, so it may attempt to access an invalid index and cause the plug-in to crash, potentially bringing down the entire DAW. Similar problems may occur on recalculating coefficients for filters and other situations where memory is allocated or deallocated while the effect is running.

The solution is to provide a way of ensuring that only one of setParameter() and processBlock() can run at the same time. If processBlock() is running and setParameter() wants to change a buffer size, it must wait until processBlock() finishes, and the same is true in reverse of setParameter() begins first. This behavior can be achieved using a special variable type called a mutex (short for mutually exclusive). When one thread locks the mutex, the other thread must wait until it has been unlocked before continuing. Consider the following example code from a phase vocoder effect. First, in PluginProcessor.h, the mutex variable is defined within the audio processor class:

// Spin lock that prevents the FFT settings from changing in the middle of the audio

// thread.

SpinLock fftSpinLock\_;

SpinLock is a type of lock object provided by JUCE. At the beginning of processBlock(), the method acquires the lock before beginning its processing and releases it upon completion:

void PVOCPassthroughAudioProcessor::processBlock (AudioSampleBuffer& buffer, MidiBuffer& midiMessages)

{

// Helpful information about this block of samples:

const int numInputChannels = getNumInputChannels(); // How many input channels for our // effect?

const int numOutputChannels = getNumOutputChannels(); // How many output channels for our

// effect?

const int numSamples = buffer.getNumSamples(); // How many samples in the buffer for t

// this block?

int channel, inwritepos, sampsincefft;

int outreadpos, outwritepos;

// Grab the lock that prevents the FFT settings from changing

fftSpinLock\_.enter();

// [...]

// \*\*\*\* Phase Vocoder processing goes here

// [...]

fftSpinLock\_.exit();

}

Meanwhile, any call that changes the window size or invalidates the internal buffers will also acquire the lock. If one thread holds the lock and a second tries to acquire it, the second thread blocks (waits) until the lock has been released. This ensures that no two threads act on the same resources in incompatible ways.

Multi-threaded programming can be quite subtle and challenging, particularly as the number of threads and number of shared resources increase. Several good texts exist on the subject, e.g., [98].

## Conclusion

This chapter has shown how to put the principles of building audio effects into practice through the creation of audio plug-ins. The JUCE framework is a convenient environment for creating audio plug-ins since it allows the same code to be compiled into multiple plug-in formats on multiple operating systems. However, the general principles of writing audio callback functions, managing parameters, allocating resources and creating a user interface are similar across many plug-in formats. The example code included with the book offers templates from which you can start building your own effects, including examples from most chapters and a blank plug-in into which you can add your own code.

# References

[1] G. Milner, *Perfecting Sound Forever: The Story of Recorded Music*: Granta Books, 2010.

[2] M. Vdovin, "Artist Interview: Brad Plunkett," *Universal Audio WebZine,* vol. 3, October 2005.

[3] M. Lewisohn, *The Beatles Recording Sessions*. New York: Harmony Books (Crown Publishers), 1989.

[4] G. Massenburg, "Parametric Equalization," in *42nd AES Convention*, 1972.

[5] G. Martin and W. Pearson, *Summer of Love: The Making of Sgt Pepper*. London: Pan Books, 1994.

[6] J. O. Smith, *Physical Audio Signal Processing*: W3K Publishing, 2010.

[7] J. O. Smith, *Spectral Audio Signal Processing*: W3K Publishing 2011.

[8] A. V. Oppenheim and R. W. Schafer, *Discrete-time signal processing*, 3rd ed.: Prentice Hall, 2009.

[9] S. J. Orfanidis, *Introduction to Signal Processing* Orfanidis (prev. Prentice Hall), 2010.

[10] U. Zolzer, *Digital Audio Signal Processing*, 2nd ed.: John Wiley and Sons, Ltd., 2008.

[11] O. Niemitalo. (2001, Oct.). Polynomial Interpolators for High-Quality Resampling of Oversampled Audio. Available: <http://yehar.com/blog/wp-content/uploads/2009/08/deip.pdf>

[12] A. M. Stark*, et al.*, "Real-Time Beat-Synchronous Audio Effects," presented at the New Interfaces for Musical Expression (NIME), New York, NY, 2007.

[13] J. D. Reiss, "Design of audio parametric equalizer filters directly in the digital domain," *IEEE Transactions on Audio, Speech and Language Processing,* vol. 19, pp. 1843-48, August 2011.

[14] D. Self, *The Design of Active Crossover*. Oxford: Focal Press, 2011.

[15] A. Case, *Mix Smart: professional techniques for the home studio*: CRC Press, 2012.

[16] R. Izhaki, *Mixing Audio: Concepts, Practices and Tools*: Focal Press, 2008.

[17] E. Perez Gonzalez and J. D. Reiss, "Automatic equalization of multi-channel audio using cross-adaptive methods," presented at the 127th AES Convention, New York, 2009.

[18] M. Holters and U. Zölzer, "Graphic Equalizer Design Using Higher-Order Recursive Filters," in *Digital Audio Effects (DAFx)*, Montreal, 2006.

[19] ISO, "ISO 266, Acoustics– Preferred frequencies for measurements," 1975.

[20] L. Bregitzer, *Secrets of Recording: Professional Tips, Tools & Techniques*: Focal Press, 2009.

[21] Z. Ma*, et al.*, "Implementation of an Intelligent Equalization Tool Using Yule-Walker for Music Mixing and Mastering," in *134th AES Convention*, Rome, 2013.

[22] A. Loscos and T. Aussenac, "The wahwactor: a voice controlled wah-wah pedal," in *New interfaces for musical expression (NIME)*, 2005, pp. 172–175.

[23] R. G. Keen. (1999, Sept.). Technology of Wah Pedals. Available: [www.geofex.com/article\_folders/wahpedl/wahped.htm](http://www.geofex.com/article_folders/wahpedl/wahped.htm)

[24] R. Parncutt, "A Perceptual Model of Pulse Salience and Metrical Accent in Musical Rhythms," *Music Perception,* vol. 11, Summer 1994.

[25] H. Fastl and E. Zwicker, *Psychoacoustics: facts and models*, 3rd ed.: Springer, 2007.

[26] S. Rosen, "Temporal Information in Speech: Acoustic, Auditory and Linguistic Aspects," *Philosophical Transactions of the Royal Society B,* vol. 336, June 1992.

[27] B. Lachaise and L. Daudet, "Inverting dynamics compression with minimal side information," in *Digital Audio Effects Workshop (DAFx)*, Helsinki, 2008.

[28] D. Giannoulis*, et al.*, "A Tutorial on Digital Dynamic Range Compressor Design," *J. Audio Eng. Soc.,* vol. 60, June 2012.

[29] B. Rudolf, "1176 Revision History," *Mix Magazine Online,* June 1 2000.

[30] D. Barchiesi and J. D. Reiss, "Reverse Engineering the Mix," *Journal of the Audio Engineering Society,* vol. 58, pp. 563-576, July 2010.

[31] R. J. Cassidy, "Level Detection Tunings and Techniques for the Dynamic Range Compression of Audio Signals," in *117th AES Convention*, 2004.

[32] J. S. Abel and D. P. Berners, "On peak-detecting and rms feedback and feedforward compressors," in *115th AES Convention*, 2003.

[33] P. Dutilleux*, et al.*, "Nonlinear Processing, Chap. 4," in *Dafx:Digital Audio Effects*, U. Zoelzer, Ed., 2nd ed: Wiley, John & Sons, 2011, p. 554.

[34] J. Bitzer and D. Schmidt, "Parameter Estimation of Dynamic Range Compressors: Models, Procedures and Test Signals," presented at the 120th AES Convention, 2006.

[35] L. Lu, "A digital realization of audio dynamic range control," in *Fourth International Conference on Signal Processing Proceedings (IEEE ICSP)*, 1998, pp. 1424 - 1427.

[36] Sonnox, "Dynamics plug-in manual," *Sonnox Oxford Plug-ins,* April 1st 2007.

[37] M. Zaunschirm*, et al.*, "A sub-band approach to musical transient modification," *Computer Music Journal,* vol. 36, Summer 2012.

[38] M. Senior, *Mixing Secrets for the small studio*: Focal Press, 2011.

[39] D. Gibson, *The Art of Mixing*, 2nd ed.: artistpro.com LLC, 2005.

[40] M. J. Terrell*, et al.*, "Automatic Noise Gate Settings for Drum Recordings Containing Bleed from Secondary Sources," *EURASIP Journal on Advances in Signal Processing,* vol. 2010, pp. 1-9, 2010.

[41] S. Gorlow and J. D. Reiss, "Model-Based Inversion of Dynamic Range Compression," *IEEE Trans. Audio Speech Lang. Proc.,* vol. 21, 2013.

[42] B. De Man and J. D. Reiss, "An Intelligent Multiband Distortion Effect," presented at the AES 53rd International Conference on Semantic Audio, London, UK, 2014.

[43] D. Yeh*, et al.*, "Simulation of the diode limiter in guitar distortion circuits by numerical solution of ordinary differential equations," presented at the Digital Audio Effects (DAFX), 2007.

[44] V. Välimäki*, et al.*, "Virtual Analog Effects, Chap. 12," in *Dafx:Digital Audio Effects*, 2nd ed: Wiley, John & Sons, 2011.

[45] D. Yeh*, et al.*, "Numerical methods for simulation of guitar distortion circuits," *Computer Music Journal,* vol. 32, pp. 23-42, 2008.

[46] J. Pakarinen and D. Yeh, "A review of digital techniques for modeling vacuum-tube guitar amplifiers," *Computer Music Journal,* vol. 33, pp. 85-100, 2009.

[47] J. Pakarinen and M. Karjalainen, "Enhanced wave digital triode model for real-time tube amplifier emulation," *IEEE Transactions on Audio, Speech, and Language Processing,* vol. 18, pp. 738-746, 2010.

[48] W. M. J. Leach, "SPICE models for vacuum-tube amplifiers," *Journal of the Audio Engineering Society,* vol. 43, pp. 117-126, 1995.

[49] H. Shapiro and C. Glebbeek, *Jimi Hendrix: Electric Gypsy (New and Improved ed.)*: St. Martin's Press, 1995.

[50] J. L. Flanagan and R. M. Golden, "Phase Vocoder," *Bell System Technical Journal,* pp. 1493-1509, Nov. 1966.

[51] M. Portnoff, "Implementation of the digital phase vocoder using the fast Fourier transform," *IEEE Transactions on Acoustics, Speech, and Signal Processing,* vol. 24, pp. 243- 248, 1976.

[52] J. Laroche and M. Dolson, "Improved phase vocoder timescale modification of audio," *IEEE Trans. Speech and Audio Processing,* vol. 7, pp. 323 -332, 1999.

[53] M. Dolson, "The phase vocoder: A tutorial " *Computer Music Journal,* vol. 10, pp. 14-27, 1986.

[54] J. Vilkamo*, et al.*, "Directional Audio Coding: Virtual microphone-based synthesis and subjective evaluation," *Journal of the Audio Engineering Society,* vol. 57, 2009.

[55] D. Beheng, "Sound Perception," Deutche Welle, Radio Training Centre (DWRTC), Cologne 2002.

[56] B. Bernfeld, "Attempts for Better Understanding of the Directional Stereophonic Listening Mechanism," presented at the 44th Audio Engineering Society Convention, 1973.

[57] V. Pulkki, "Virtual Sound Source Positioning Using Vector Base Amplitude Panning," *J. Audio Eng. Soc.,* vol. 45, 1997.

[58] D. G. Malham and A. Myatt, "3-d sound spatialization using ambisonic techniques," *Computer Music Journal,* vol. 19, pp. 58-70, 1995.

[59] J. S. Bamford, "An Analysis of Ambisonics Sound Systems of First and Second Order," MSc, University of Waterloo, Waterloo, Ont., Canada, 1995.

[60] M. A. Poletti, "Three-Dimensional Surround Sound Systems Based on Spherical Harmonics," *J. Audio Eng. Soc.,* vol. 53, November 2005

[61] E. G. Williams, *Fourier Acoustics: Sound Radiation and Nearfield Acoustical Holography*. London: Academic Press, 1999.

[62] P. M. Morse and K. U. Ingard, *Theoretical Acoustics*. New York: McGraw-Hill, 1968.

[63] M. Gerzon, "Design of Ambisonic Decoders for Multispeaker Surround Sound," in *58th Audio Egineering Society Convention*, 1977.

[64] E. N. G. Verheijen, "Sound Reproduction by Wave Field Synthesis," PhD, Delft University of Technology, 1998.

[65] J. Blauert, *Spatial Hearing: The Psychophysics of Human Sound Localization*: MIT Press, 1997.

[66] F. Rumsey, *Spatial Audio*: Focal Press, 2001.

[67] J. W. Strutt, "On our perception of sound direction," *Philos. Mag.,* vol. 13, pp. 214-232, 1907.

[68] V. R. Algazi*, et al.*, "Motion-Tracked Binaural Sound," *J. Aud. Eng. Soc.,* vol. 52, November 2004.

[69] C. P. Brown and R. O. Duda, "A Structural Model for Binaural Sound Synthesis," *IEEE Trans. Speech and Audio Processing,* vol. 6, Sept. 1998.

[70] R. O. Duda, "3-D Audio for HCI," 2000.

[71] V. Pulkki, "Spatial Sound Generation and Perception by Amplitude Panning Techniques," Electrical and Communications Engineering, Helsinki University of Technology, Helsinki, Finland, 2001.

[72] D. d. Vries*, et al.*, "The wave field synthesis concept applied to sound reinforcement: restrictions and solutions," presented at the 96th Convention of the Audio Engineering Society, Amsterdam, 1994.

[73] A. Farina, "Simultaneous measurement of impulse response and distortion with a swept-sine technique," presented at the 108th AES Convention, 2000.

[74] A. Pierce, *Acoustics: an introduction to its physical principles and applications*: American Institute of Physics, 1989.

[75] J. O. Smith, "Doppler simulation and the Leslie," in *5th Intl. Conf. on Digital Audio Effects (DafX)*, Hamburg, 2002.

[76] B. G. Quinn, "Doppler speed and range estimation using frequency and amplitude estimates," *J. Acoust. Soc. Am.,* vol. 98, pp. 2560-2566, 1995.

[77] A. R. Gondeck, "Doppler time mapping," *J. Acoust. Soc. Am.,* vol. 73, pp. 1863-1864, 1983.

[78] D. Rocchesso, "Fractionally addressed delay lines," *IEEE Transactions on Speech and Audio Processing,* vol. 8, November 2000.

[79] L. L. Beranek, "Analysis of Sabine and Eyring equations and their application to concert hall audience and chair absorption," *J. Acoust. Soc. Am.,* vol. 120, Sept. 2006.

[80] D. Howard and J. Angus, *Acoustics and Psychoacoustics*, 4th ed.: Focal Press, 2009.

[81] D. Davis and E. Patronis, *Sound System Engineering*, Third ed.: Elsevier, 2006.

[82] D. Davis, "A More Accurate Way Of Calculating Critical Distance," Altec TL-207, 1971.

[83] M. R. Schroeder and B. F. Logan, "Colorless Artificial Reverberation," *Journal of the Audio Engineering Society,* vol. 9, 1961.

[84] M. R. Schroeder, "Natural Sounding Artificial Reverberation," *Journal of the Audio Engineering Society,* vol. 10, 1962.

[85] J. A. Moorer, " About this reverberation business," *Computer Music Journal,* vol. 3, pp. 13-18, 1979.

[86] J. Allen and D. Berkley, "Image method for efficiently simulating small-room acoustics," *J. Acoust. Soc. Am.,* vol. 65, pp. 943–950, 1979.

[87] E. Lehmann and A. Johansson, "Prediction of energy decay in room impulse responses simulated with an image-source model," *Journal of the Acoustical Society of America,* vol. 124, July 2008.

[88] P. Peterson, "Simulating the response of multiple microphones to a single acoustic source in a reverberant room," *Journal of the Acoustical Society of America,* vol. 80, November 1986.

[89] W. G. Gardner, "Efficient Convolution without Input-Output Delay," in *97th Convention of the Audio Engineering Society*, San Francisco, CA, 1994.

[90] D. M. Huber and R. E. Runstein, *Modern Recording Techniques*, seventh ed.: Focal Press, 2010.

[91] D. K. Wise, "Concept, Design, and Implementation of a General Dynamic Parametric Equalizer," *Journal of the Audio Engineering Society,* vol. 57, pp. 16-28, Jan. 2009.

[92] P. Pestana, "Automatic Mixing Systems Using Adaptive Audio Effects," PhD, Universidade Catolica Portuguesa, Lisbon, Portugal, 2013.

[93] P. D. Pestana and J. D. Reiss, "Intelligent Audio Production Strategies Informed by Best Practices," presented at the AES 53rd International Conference on Semantic Audio, London, UK, 2014.

[94] B. Stroustrup, *The C++ Programming Language*: Addison Wesley, 2000.

[95] A. Koenig and B. E. Moo, *Accelerated C++*: Addison Wesley, 2000.

[96] S. B. Lippman*, et al.*, *C++ Primer*, 5th ed.: Addison Wesley, 2012.

[97] R. Boulanger*, et al.*, *The Audio Programming Book*: MIT Press, 2010.

[98] M. Walmsley, *Multi-Threaded Programming in C++*: Springer 1999.