A sampler plugin for digital audio workstations

Trabajo de Fin de Grado en Ingeniería Informática

Juan Chozas Sumbera

Dirigido por
Jaime Sánchez Hernández
Miguel Gómez-Zamalloa Gil

UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMÁTICA

Abstract

Music production is at anyone's reach. Nowadays, computers, smartphones and tablets are able to run software that provides the necessary tools to compose music. Digital Audio Workstations (DAWs) are found at the more sophisticated end of the spectrum. These feature-packed programs are the type that you would find in locations ranging from the aspiring musician's personal computer to the most professional recording studio in your city. One of the most powerful characteristics of these environments is being extensible via plugins.

The sampling techniques that emerged during the end of the 20th century gifted the world with new methods of music composition. The technique that this project develops upon consists in the creation of an instrument using audio recordings as its only source of sound. The nature of the process is far from complex and it can be streamlined to provide a fast way to design a bespoke instrument. By implementing this tool as a plugin, the instrument takes the form of a loadable module that is readily accessible to anybody working on a DAW.

Key words

music, sound, sampler, sampling, instrument, MIDI, DAW, plugin, VST, synthesizer

Resumen

Cualquier persona tiene a su alcance la producción musical. El mundo en el que vivimos esta plagado de dispositivos capaces de ejecutar programas que ofrecen todas las herramientas necesarias para componer música. Las estaciones de audio digital son los programas que ofrecen la mayor gama de herramientas para la creación de música. Es el tipo de software que no puede faltar en los ordenadores de los músicos principiantes ni de los estudios de grabación más sofisticados. Estos entornos pueden ser enriquecidos mediante módulos externos (plugins), extendiendo la gama de utilidades que pone a disposición del usuario.

A finales del siglo 20 emergieron nuevas técnicas de muestreo, nuevas maneras de producir música utilizando grabaciones de sonido. Este proyecto utiliza estas técnicas de base para ofrecer una manera rápida de crear un instrumento a medida que utiliza regiones de una muestra para producir sonido. El proceso es sencillo por naturaleza y se puede implementar de forma eficiente para hacer un proceso sin demoras innecesarias del diseño de un instrumento basado en muestras. La implementación en forma de plugin hace que el instrumento sea accesible para todo el que trabaje con una estación de audio digital.

Palabras clave

música, sonido, muestra, muestreo, instrumento, MIDI, DAW, plugin, VST, sintetizador

Contents

1 Introduction **5**
1.1 Background 5
1.1.1 Sampler history 5
1.1.2 Pulse-Code Modulation 5
1.1.3 Modern music production 6
1.1.4 Sampling technique 8
1.2 Motivation 10
1.3 Objectives 11

2 Plugin design **12**

3 Implementation **13**
3.1 JUCE framework 13
3.2 Audio preview 15
3.3 Chopping the main sample 17
3.3.1 Manual creation 17
3.3.2 Peak detection 18
3.4 Subsection management 19
3.5 Audio playback 21

4 Results **22**
4.1 Conclusion 23

5 Bibliography **24**

1 Introduction

1.1 Background

1.1.1 Sampler history

The technique of sampling dates back to the 1960s when recordings were captured on tape. Thanks to a hardware design that made contact between the playback head and different sections of a moving tape, musicians could play the distinct recordings on keyboards of instruments such as the Mellotron. During the 1980s, the popularity of drum machines increased significantly. Many drum machines were sample-based, i.e. they created sounds using digitally stored samples, whereas the alternative was to synthesize sound in an analog fashion. In 1988, the first Music Production Center (MPC) by Akai was made available to the public: a sample-based music workstation that was capable of arranging samples of all lengths in its sequencer to produce full fledged music tracks without the need for additional instruments or hardware. In a recipe for music, an MPC could be the only ingredient, reaching a vast number of musicians because of its affordability in comparison to previous means of music production. It is a tool that gave artists a new way to create music, a technique that has been a foundation to several genres and highly influential to music as a whole.

1.1.2 Pulse-Code Modulation

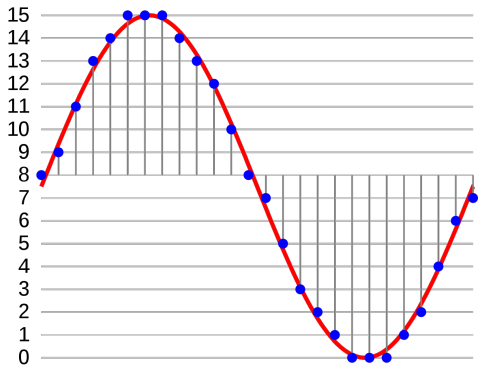

Figure 1: Linear pulse-code modulation at 4 bit depth. [4]

The method that makes sampling a possibility is Pulse-Code Modulation (PCM for short). Dating back to the 1930s, it consists of the digital representation of analog signals, and is subject to two parameters: sampling rate

and bit depth. This technique processes an input analog signal by making readings at equidistant intervals. Commonly referred to as samples, these readings are what the resulting digital representation will be composed of. The sampling rate refers to how often samples will be read from the analog signal, and the bit depth indicates the amount of bits with which to describe each sample. Every time an analog reading is being converted into a digital sample, it is rounded to the nearest integer value in the range determined by the bit depth. This rounding process, visible in Figure 1 is known as quantization. Since samples are encoded to integers in binary, the levels to which they can be quantized are uniformly distributed. This describes the method of Linear PCM (LPCM), a sub-method of PCM that is used for audio CDs and the popular formats WAVE and AIFF, among other applications.

1.1.3 Modern music production

In a world of highly complex information systems, digital music production has been developed thoroughly. When it comes to modern music composition, Musical Instrument Digital Interface (MIDI from now on) protocols provide grounds for an alternative to regular, "analog" instruments. This standard erects a bridge for communication between digital instruments and computers, providing the world of digital music production with the tangible interfaces of normal instruments. The MIDI standard provides, among many other features, a communications protocol that encodes music events. This manifests in the form of messages which describe data such as the pressing, releasing, and velocity of musical notes.

A digital audio workstation (DAW from here on out) is a software environment for music production. These workstations revolve around MIDI, which provides an encoded language for musical notes. See Table 1 for a table of the syllabic and alphabetical equivalents of musical notes. Note-related MIDI messages always store data regarding the note subject to the event.

Syllabic	Alphabetical
Do	C
Re	D
Mi	E
Fa	F
Sol	G
La	A
Si	B

Table 1: Musical notes in syllabic and alphabetical form

One can have an extensive range of digital synthesizers, samplers, and effect chains all working simultaneously in a DAW, the type of software that provides a playground that routes audio between racks of components and effects, making music production a possibility to anyone that owns a computer. DAWs have an element called a sequencer, which serves as a sort of digital equivalent to sheet music. The sequencer is where producers draw out musical events, placing small rectangles on a grid to represent the notes, their length, and their duration. The arrangements of events are then encapsulated in what are called patterns. These patterns can be applied to instruments, such as synthesizers, or samplers, to trigger the sounds they produce. Patterns can be drawn out manually, auto generated, or recorded in real time with a MIDI controller. Auto generated patterns consist of filling a set length of time with notes separated by constant intervals, so as to mimic the timing of a metronome, for example. Patterns that are recorded in real time are made with the DAWs recording function, which is activated to record MIDI events from a controller.

When recording patterns, incoming MIDI events are timestamped to then be displayed in chronological order and resemble the translation of the user's input. A user may have an instrument loaded during this recording period, so the user can know what sounds are being produced. It is important to highlight what is captured by the DAW is a sequence of MIDI events, not the actual sounds being produced by the instrument, meaning that a pattern is just a sequence of data, void of sound until an instrument that can process the events is associated to it. More so, since patterns are composed of MIDI events meant to trigger responses from instruments, there is no binding between them and instruments, and a pattern made for one instrument can perfectly be used for another.

The relevant events associated to the pattern building process are: note-on events, the signal that a note has been pressed; note-off events, the signal that a note has been released. The subtraction of the timestamps associated to these two events determines how long a note is held down. A musical note is associated to note-on and note-off events, having the note and octave being encoded as an integer in the range of $[0, 127)$. In addition, associated to note-on events is an element called velocity, which is a measure of the intensity of the note. The velocity of a note is sometimes processed by instruments to generate softer or stronger sounds. Some samplers use note velocity to trigger a sample from a set, i.e. a lower intensity recording of a snare drum for lower velocities as opposed to a sample of a powerful snare drum hit when the velocity is a higher range.

DAWs can be extended with plugins in the form of effects or instruments. The former normally have a sound-based input-output relation with

the DAW, meanwhile instruments will take MIDI messages and produce a sound as an output. The most common standard for plugins is Virtual Studio Technology (VST), created by a german company called Steinberg. VST plugins have multi platform support on major operating systems, however, plugins must be exported for a target operating system, i.e. a VST plugin that is exported for Windows will not work on Mac OS or Linux. A possible VST plugin could be a sine wave based instrument, for example. The wave's frequency will depend on the key being pressed, and the force applied to the key will dictate the amplitude of the wave. This plugin, although simple, could be considered a synthesizer. A sampler is a synthesizer that uses samples rather than oscillators to produce sound. The sources of the sounds it produces are samples, which can be described as clips of audio that are stored in a digital format within the sampler's memory. You're likely to find that hardware and software samplers offer sets of "stock" samples that are loaded by default. The option to load samples from an external source such as an SD card is also common, just like recording external input from a recording device such as a microphone. As is the case with non sample based synthesizers, there are several ways that the samples can be manipulated.

1.1.4 Sampling technique

The key concept of sampling techniques is to modify and manipulate audio samples to create sounds. Many types of transformations can be applied to samples in order to drastically change the way they sound, although products that sound far different to the source sample are not always the objective. A sample can be manipulated by delimiting subsections within it, so as to create separated sub-samples that can be treated as individual notes. This process is colloquially referred to as *chopping* the sample, resulting in a set of *chops*, the sub-samples obtained from the main sample.

Another way to manipulate a sample is via *envelopes*, which dictate the way a sound changes over time. Envelopes affect playback every time it is triggered, and are commonly composed of four parameters. The first one comes into play upon note-on events, i.e. on a key press: *attack*, describing the time it takes for the sound's level to rise from zero to its peak. Next, *sustain* is the level at which the sound will hold until a note-off event, i.e. when a key is released. *Decay* depends on sustain, and it describes the time it will take for the level to go from peak to sustain. Finally, *release* is the tail end of the envelope. Starting on note-off events, it describes the time it will take to reach zero level from sustain level. These four concepts form the acronym ADSR, and can be pictured as a plot of level against time in Figure 2. You may have noticed that three of the parameters refer to a time

measure, whereas sustain refers to a level.

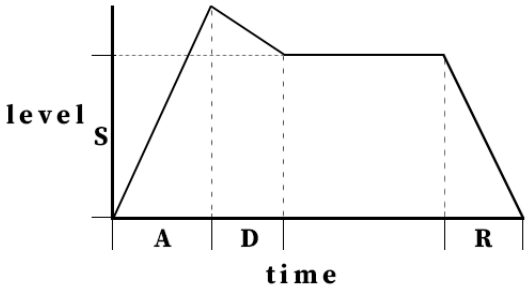

Figure 2: ADSR parameters in envelopes

The entity that is the sample based instrument is what holds and modifies the configuration that revolves around a main sample. It knows where to find the main sample, where each subsection starts and ends, and what the envelope parameters are set to for each chop. The sampler is equipped for playback, and is configurable in the way sounds interact as they are triggered.

A parameter that can be found in most instruments is termed *polyphony*, a measure that limits the playback at any moment to a particular number of notes. Polyphony is commonly described with an arbitrary number of *voices*: an instrument with 3 voices can play at most 3 sounds at a time.

Furthermore, a series of effects may be applied to the sample in order to modify the sound that is output. Filters and compressors are some of the most common effects you may come across. Filters eliminate frequencies from the sound. Compressors apply reduction to the sound when the level crosses a certain threshold. Being applicable to any source of sound, these types of effects are most likely included with any arbitrary DAW, although they are sometimes also found within samplers.

1.2 Motivation

Modern DAWs have built in samplers or similar tools that allow a user to sample sound and directly manipulate it so as to employ sampling techniques. In most cases, these tools are more than capable of providing the means to translate these techniques into action, however, depending on the DAW, you may be hindered due to the workstation's design. Whether it be the user interface, the imposed workflow, or the minimum amount of steps required to reach your sampling goals, it is likely that the process involves inconveniences.

Personally, I find that there are a minimum of three steps to achieve a usable sampler configuration. First, a main sample must be loaded. Second, a number of chops are delimited in the bounds of the main sample. Third comes the assignment of chops to MIDI notes, in other words, the mapping of sample subsection playback to a controller's keys. These steps are a prerequisite to playing chops on a controller, or to lay notes out within the DAW's sequencer. At this stage, the user can experience the instrument and judge whether it is necessary to take a step back and make adjustments, for example, shifting the start time of a chop, creating/deleting a chop, or moving the trigger note to another of higher convenience. All of this implies that the design of a sampler instrument is an iterative process, and that the user may keep reforming it in several ways to fit the necessities of the creative process in musical composition.

Since the essential number of steps and functions required to obtain a usable instrument are few, there is room for improvement in comparison to the process that takes place when using a DAW's default tools. By focusing on the indispensable functions and actions, the process can be simplified to provide a fast way of putting together a sample based instrument. Hindrances such as overloaded user interfaces are eliminated, leaving out the buttons and functionalities that are seldom used.

By implementing the instrument as a VST plugin, a vast number of DAWs become candidates for the inclusion of this plugin and a streamlined sampling process. Any DAW's inherent limitations and/or hindrances would no longer be obstacles when it comes to creating a sample based instrument. This way, an instrument strictly designed for sampling would be attachable to the majority of workstations, providing a simple way to put sampling techniques to use on most platforms.

1.3 Objectives

For the simple implications of sampling techniques, the means by which you achieve a minimum setup with which you can play around and manifest ideas should be straightforward. The aim is to reduce the amount of user interaction required to reach a usable sample based instrument. To do so, I will downsize in the features that any arbitrary DAW has to offer, including only those that are strictly necessary. This translates to the reduction of hindrances that may arise when there is an excess amount of functionality that is rarely used. By implementing the instrument as a VST plugin, I aspire to build an accessible sample based instrument optimized for the application of sampling techniques.

The core capabilities the sampler can be listed as four points:

- Providing a visual representation and an audio preview of the main sample and its chops.
- A way to create subsections in the audio clip.
- List component that shows all chops and the trigger notes they are associated to.
- Chop playback: adjustable envelope parameters for each chop, and a polyphony setting for the sampler as a whole.

Furthermore, there are more features that would be of great value, such as the following:

- *Automatic* chop creation using peak detection algorithms.
- The ability to load and manage a set of main audio samples simultaneously.
- Plugin state saving and loading, for the storage and recovery of configurations.
- Pitch shifting algorithms to modify the tone of the main sample and its chops.
- A series of effects for each chop: filters, compressors,

2 Plugin design

3 Implementation

3.1 JUCE framework

To implement the sampler, I used an application framework called JUCE[3]. It has cross-platform support and is written in C++. The amount of documentation and tutorials available online was enough to convince me that I would need nothing else to implement the majority of the instrument.

JUCE makes use of the object oriented capabilities of C++ to present a wide array of classes and utilities. It offers the building blocks to application development with a reasonably gentle learning curve that discards the need to reinvent the wheel. These utilities help with tasks that range from audio playback, to event driven data structures, to the design of user interfaces. Furthermore, the framework is in line with the event-driven paradigm, which makes the development of action based applications a simple and efficient task. As stated, the documentation is vast and there are numerous descriptive tutorials. On top of that, at the programmers disposal are more than 70 example applications that showcase the several features the framework has to offer.

The Projucer is JUCE's cross-platform application that allows for the creation, management and configuration of JUCE projects. To enable development on Linux, for example, one can add a Makefile exporter to their project. The Projucer will create an architecture specific Makefile that links external libraries and builds the application. The templates offered from which to start new projects include GUI Application, Animated Application, Console Application, Audio Application, Audio Plugin, and Static Library and Dynamic Library. Each of these archetypes are configurable in distinct ways, and are initialized to include the fundamental code inherent to the nature of the application. For Audio Plugin projects, the Projucer can export several plugin formats, such that one build process compiles and exports the plugin formats that the project is configured to build. The formats include VST and Standalone Application, among other less relevant plugin formats. Building a Standalone Application allows for the usage of the plugin without the need of a DAW. Building VST plugins is only supported on Windows, which means that for development on Linux, one can only test the build in an isolated, standalone application, void of connections to a DAW.

The Projucer's base audio plugin project provides fundamental code containing two classes. The dichotomy encourages the separation of the audio processing logic from the GUI side of the plugin. The *PluginProcessor*, as the name suggests, is the audio processing heart of the application. This class extends the *AudioProcessor* class, a base class for audio processing. This is

where tasks such as MIDI connection management and audio rendering take place. The *PluginEditor* extends from the *AudioProcessorEditor* class, and is intended to be the GUI. It is the place where interaction with GUI and all on screen visualizations are created and managed. Unlike the *AudioProcessor* class, which inherits from no other, the *AudioProcessorEditor* class extends from the *Component* class. This is the base class for all user interface objects that can be visualized. The *Component* class has two virtual functions that are to be overridden by any class derived from it: `paint()`, the function that dictates the on screen layout of the component; `resized()`, which contains reactionary logic for when the component is resized. Both functions are callbacks, which means they are invoked automatically upon triggering events.

3.2 Audio preview

The first step in developing the sampler consisted in loading an audio file. With the help of the *Draw audio waveforms* tutorial [2], I was quickly able to identify the classes I would need to load, and, additionally, display a visual representation of an audio file. For starts, a *FileChooser* object is used to create a dialog box that selects files or folders. One can limit the formats allowed for selection by specifying semicolon separated regular expressions in its constructor. Using the string `"*.wav"` as the `filePatternsAllowed` parameter will only allow users to select WAVE files, for example. The class offers several member functions named with "`browseFor`" as a prefix, and `browseForFileToOpen()` opens a dialog box to select a file for opening, returning a boolean value that indicates whether the user selected a file or not. In the former case, the FileChooser's `getResult()` function will return the *File* object that was selected.

The class that would provide the visual representation of the audio file is the *AudioThumbnail* class. It works hand in hand with the *AudioThumbnailCache* class, which serves as a cache memory of audio file representations. Together, they store and print waveform representations of audio files. I created a separate class *SamplerThumbnail* to bundle the *AudioThumbnail* object and its related objects and functions into a separate component. For the thumbnail view to load the audio file, I created a public function `setFile(const File& file)` within the *SamplerThumbnail* class. Within `setFile`, the audio file is loaded with the `setSource(FileInputSource* newSource)` function of *AudioThumbnail*. The *FileInputSource* object can be created from the *File* received from the *PluginEditor* by passing the file as the only parameter to the constructor of a *FileInputSource* object. With this done, the *AudioThumbnail* will load the waveform representation of user-selected audio files.

Audio playback is achieved through the *AudioTransportSource* class, which enables the audio preview of the main sample that is loaded in the plugin. It inherits from the *AudioSource* class, which is a base class that enables continuous audio playback. The *AudioTransportSource* is used because it implements can be positioned, played, paused, and stopped. These features are useful for previewing the initially loaded sample. Audio subsection playback is handled differently, and will be discussed later on. The connection of audio files to the *AudioTransportSource* object is achieved via *AudioSource* subclasses. In this case, an *AudioFormatReaderSource*. The processor stores an instance of *AudioFormatReaderSource* in `currentAudioFileSource`, and in `transportSource` an instance of *AudioTransportSource*. *AudioFormatReaderSource* objects of this class obtain streams of sound from *AudioForma-*

tReader objects, which source their sound from audio file streams. Every time a user opens a file successfully, a function `loadFileIntoTransport(const File& file)` is executed. This function will stop the *AudioTransportSource* from continuing playback, and proceed to change the source. To do this, a new *AudioFormatReader* object is created for `file`. This is done through an `AudioFormatManager` object, which is used to keep a list of the formats that are to be accepted by the application. It includes a method called `AudioFormatReader* createReaderFor (const File& file)`, which returns a pointer to the *AudioFormatReader* that is necessary for the creation of an *AudioFormatReaderSource*. If `createReaderFor` fails, it returns a null pointer, so the following steps are only taken in cases where the pointer is not null. The `currentAudioFileSource` pointer is reset to the new reader source, and the `transportSource`'s new source is set with `setSource`.
All of these instances and connections lead to a functional way to play audio, however, there still is the need to connect the `transportSource`'s interface to GUI elements. The *AudioTransportSource* class extends *ChangeBroadcaster*, which allows for event driven reactivity to the state of the *transportSource* object. To aid in the process, an `enum TransportState` is declared to include the states of `Stopping`, `Stopped`, `Starting`, `Playing`, `Pausing`, and `Paused`. Buttons layed out in the application will change a private variable `TransportState state` to `Starting`, `Stopping`, and `Pausing` to invoke `transportSource.start()` or `transportSource.stop()`. By executing these methods of the `transportSource`, every object that extends *ChangeListener* that has been added to the `transportSource`'s list of listeners will be notified of a change. This notification manifests by invoking the callback function `void changeListenerCallback (ChangeBroadcaster* source)`, which is a virtual function that must be implemented by any class that extends *ChangeListener*. The plugin editor extends this class and adds itself to the `transportSource`'s listener list with the following instruction: `transportSource.addChangeListener(this);`. In the `changeListenerCallback` function, the source of the event is checked by comparing references like so:

```
void changeListenerCallback (ChangeBroadcaster* source) override
{
if (source == &transportSource)
{...
```

Inside the conditional block, the transport state is updated once more, and the GUI is synchronized in accordance to the next transport state, for example, when `Starting`, the state transitions to `Playing`, changing the play button's text to "Pause" and enabling the stop button.

3.3 Chopping the main sample

Chops are modelled as a struct and are comprised of 6 fields: an unique identifier, start and stop times and samples, the trigger note, and a boolean value named *hidden*. Start and stop measures are stored in seconds and samples. The trigger note is an integer encoding the MIDI note that is associated to the chop, and the *hidden* field dictates the chops' visibility in the components that display chops.

The storage of these fields is done via the *ValueTree*[7] class, which is a lightweight data structure that is capable of handling datatypes of all sorts, storing a list of properties, subtrees, and references to shared data containers. Using a struct to wrap this data structure brings the benefit of type and validity checks when creating chops. The creation of a chop with this construct creates a *ValueTree* object, setting the values for properties that resemble the fields previously described after performing some checks. This *ValueTree* can then be added to the general *ValueTree* that stores all chops with an assurance of validity in terms of types and value boundaries. The *ValueTree* data structure supports event-driven programming: listeners can subscribe to a tree in order to invoke callbacks when events arise, such as property changes or the removal/addition of a subtree.

3.3.1 Manual creation

The *AudioThumbnail* component plays a big part in manual chop creation. Since it displays a visual representation of the sample, it is possible to recognize patterns in the waveform to estimate the position and length of the subsection you wish to create. By default, the *AudioThumbnail* does not interface with mouse clicks. Nevertheless, the interception of clicks as events with coordinates is fully supported. I found out about this through the *AudioPlaybackDemo* that is part of the Projucer's examples, where it was possible to click on the *AudioThumbnail* to relocate the playback head of the *AudioTransportSource* to any position on the waveform.

To do this, the *AudioThumbnail* is instantiated as a member and placed within a parent component. This way, the parent component can override functions such as `void mouseDown (const MouseEvent& e)` and `void mouseDrag (const MouseEvent& e)` to define the interactions between the component and events triggered by the mouse. When one clicks on the thumbnail, the coordinates of the click event are part of the event's payload. The x coordinate is all that is needed to obtain the equivalent point in time in the sample: it is done with a few arithmetic operations that use the parent component's and the thumbnail's width and the as operands, as well as the x

coordinate. This works both ways, such that a position in time can be converted to an x coordinate. This feature allows for the rendering of a playback head that scrolls along the thumbnail as it plays in real time. Likewise, when a chop is created, its boundaries can be drawn upon the thumbnail with two markers to indicate the region it is contained in.

The thumbnail then becomes the means through which manual chop creation is possible. One way of creating a subsection is goes by the name of *Chop selection*. This is based on the possibility of overriding the `mouseDrag` function to display rectangles resembling selections over the thumbnail. The coordinates of the limits of these selected areas are stored upon the firing of the drag event. These pairs of coordinates are used to draw a see-through rectangles over the thumbnail, and, if need be, to create a chop out of the boundaries of the selection. Termed *Chop from here* is the second way to manually create chops. On activation, this option uses only the earliest coordinate of the thumbnail selection, making chop's end point always coincide with the end of the main sample.

3.3.2 Peak detection

Automatic chopping is achieved through the aubio library, a set of tools designed for the extraction of annotations from audio signals[1]. Being an open source library written in C, it seemed a perfectly suitable inclusion to the sampler. To interface with the library from the application, the Projucer must know the paths to the libraries and the header files. It then adjusts the exporter in order to link the libraries when building the application.

Onset, meaning the start of a musical note, is the name of the module relevant to this task. To use the peak or onset detection algorithms provided by aubio, the first step is to create a source object to be later passed onto the onset object. The source is created using the path to the file and the sample rate of the file. These values are found through two members of the *PluginEditor* class, with the *File* object's `getFullPathName()` function and the *AudioFormatReaderSource* object's `getAudioFormatReader()->sampleRate` attribute.

The library provides a range of eight different onset detection methods, so I created a button that allows for the selection of any method. To indicate the detection sensibility, the onset object requires a threshold be set, via `aubio_onset_set_threshold(aubio_onset_t * o, smpl_t threshold)`. The desired threshold is selectable with a slider element in the user interface. The plugin editor registers as a listener to this slider to know when the value of the threshold is changed. This is done to provide an anticipation of the number of chops that would be created with any configuration of an onset method and an arbitrary threshold setting.

Finally, associated to the press of a button, the execution of the onset algorithm is triggered with the current threshold value and onset method. Chops are created in the non-overlapping areas delimited by the detected peaks. In other words, the first chop begins on the first peak and ends on the second peak, the second begins on the second peak and ends on the third peak, etc.

3.4 Subsection management

A table component was included to visualize the amount of chops and some of their details. Initially empty, the table sees the addition of rows as chops are created. Rows are composed of four fields of information related to the chop: one for the chop's unique identifier, another two dedicated to the start and stop times describing the region covered by the chop, and a final one for the note that triggers the chop's playback. The latter field is interactive in the sense that it is represented with a drop down list, allowing the reassigning of chops' trigger notes. The notion of the essential steps to implement *custom* cells for the trigger note column was easily captured with the help of the table list box tutorial[5]. Furthermore, I added right click interaction with rows to open a menu from which chops can be deleted and hidden.

The chop list table allows for the selection of single rows. The selection of a row triggers a change on the thumbnail, which highlights the region associated to the chop. To do so, the chop list component keeps a member variable of the *Value* class named `selectedChop`. The *Value* class is a wrapper around a shared, reference-counted underlying data object [6]. It allows for the notification of changes to attached listeners, which is the main reason for choosing the *Value* class for this variable. Whenever the table's selected row changes, the thumbnail and other listening components will execute callback functions to react to the changes immediately.

On the lower part of the user interface is the chop settings component. By registering as a listener to the chop list component's `selectedChop` variable, this panel displays the selected chop's envelope parameters, which can be modified in order to determine the behavior of its playback over time. Each envelope parameter is associated to a *LabelledSlider* object, a component with the necessary objects that are a common factor to all parameters: a slider for its value and a label for its name. As will be mentioned in detail later on, each chop is associated to a *SamplerSound* object, which is responsible for it's playback. This object is the place to set the envelope parameters for each chop, so it is necessary to modify these whenever a parameter is changed.

Another function included in the chop settings component allows for the definition of the instrument's polyphony. With a button that toggles between

modes, the two settings are termed *mono* and *poly*. On one hand, *mono* playback will give the instrument one single voice for sound playback. One voice can play at most one sound, and, as stated in the previous paragraph, each chop is assigned a *SamplerSound* object for playback. This means that *mono* mode will only permit at most one chop to sound at any given moment: if a chop's playback is triggered while another is being played, the most recently activated chop will begin playback and cut off the previous. On the other hand, *poly* playback will create one voice per chop, allowing that every chop be played at the same time. Just like with the `selectedChop` variable, the chop settings component holds an instance of the *Value* class named `playbackMode`. The *SamplerAudioSource* component, in charge of managing sounds and voices, among other tasks, registers as a listener to `playbackMode` to reset the number of voices in accordance with the newest value for the instrument's polyphony setting.

For a more tangible and straightforward way to reassign chops' trigger notes, the chop settings section includes the concept colloquially known as MIDI Learn. It achieves the task of rebinding the note by *listening* to the connected MIDI controller. When a MIDI note press event is detected, the chop's trigger note is assigned to the event's note. This feature also makes use of the *Value* class. The processor declares one to know what the last recorded MIDI note is, and the chop settings component declares another to dictate whether or not to listen to MIDI notes at all. The latter, named `listenForMidiLearn` is listened to by the processor in order to dictate whether or not to update the former *Value* `lastRecordedMidiNote` to that of the incoming MIDI notes. The chop settings component listens to `lastRecordedMidiNote` to finally update the chop's trigger note with the note newly detected by the processor. In doing so, the chop list component notifies the processor via `listenForMidiLearn` that `lastRecordedMidiNote` should no longer updated.

3.5 Audio playback

synthesizer explained in voices, sounds. midi message detection in process-block

4 Results

4.1 Conclusion

Having no experience in digital audio processing, I have to say that the JUCE framework made the process straightforward and easy to grasp.

TODO juce's tutorials and example code allow for fast understanding of the framework

very powerful and capable of building audio applications

translate

5 Bibliography

References

[1] *aubio library*. URL: https://aubio.org.

[2] *Draw audio waveforms tutorial*. URL: https://docs.juce.com/master/tutorial_audio_thumbnail.html.

[3] *JUCE*. URL: https://juce.com.

[4] *Linear PCM*. URL: https://commons.wikimedia.org/wiki/File:Pcm.svg.

[5] *TableListBox tutorial*. URL: https://docs.juce.com/master/tutorial_table_list_box.html.

[6] *Value class*. URL: https://docs.juce.com/master/classValue.html.

[7] *ValueTree*. URL: https://docs.juce.com/master/classValueTree.html.