Politecnico di Milano
Computer Music: Languages and Systems
Academic Year 2022-2023
Eray Özgünay – Sebastian Mendez – Harry Foley

Monique Mono-Synth

Overview

Monique is a monophonic synthesizer plugin made in the JUCE platform. It implements subtractive synthesis with a unique signal path to create an interesting take on a classic design. One of the unique features is the "Morph Mixer" that allows a user to blend between preset settings for the oscillators, filters, arpeggiators and effects. Please see appendix for a copy of what the GUI looks like.

Block Diagram

In Figure 1, we can see the block diagram of Monique. Audio signals are shown by normal lines and control signals are shown with dashed lines. The main signal path starts from the three oscillators which are then processed by the filters, before the amplitude is controlled by an envelope and then finally the audio signal is processed by an EQ and a multi effect engine.

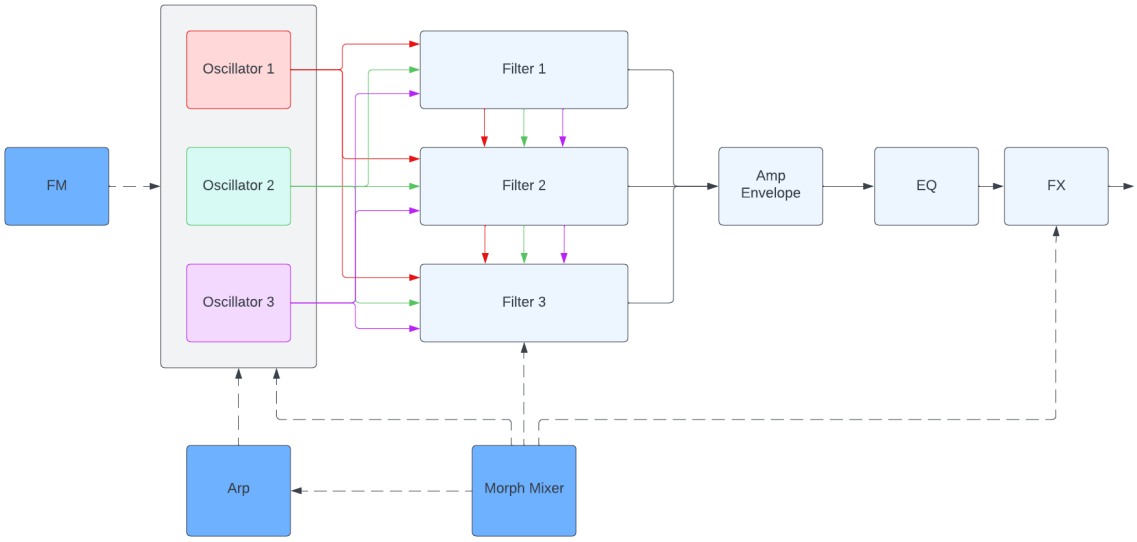

Figure 1: Block Diagram of Monique

The control signals are produced by an FM Oscillator, the Arpeggiator and the Morph Mixer. The FM oscillator provides the ability to modulate the frequency of the three oscillators at audio rate, this is known as Frequency Modulation.

The arpeggiator provides the ability to control the pitch and the velocity of the three oscillators.

The Morph Mixer sends control signals to the Oscillators, Arpeggiator, Filters and Effects Engine. For example, the Morph Mixer can be used to blend between Osc-L (left) and Osc-R (right), where Osc-L is one preset setting for the oscillators and Osc-R is another preset setting.

The signal path from the oscillators into the filters, and between the filters is interesting.

Figure 2: Monique Oscillators and Filters, reflecting the example inputs below.

For Filter 1 the user can select the input amount for each oscillator and also the output level from the filter into the Amplitude Envelope.

For Filter 2, the user can similarly select the input amount for each oscillator or can chose for the input to be taken from the output of Filter 1. This can be set individually. The third filter acts similarly to Filter 2

For example:

- Filter 1 takes Oscillator 1 and 2 as its input and applies an LPF with cut-off frequency 10 kHz.
- Filter 2 takes Oscillator 2 and 3 directly from the oscillator section as well as taking Oscillator 1 post Filter 1 as its inputs and applies a HPF of 300 Hz.

JUCE Code Structure

The source files for Monique are divided into two subfolders: Core and UI. The files in core are responsible for the backend code, the generation and processing of sound. The files in UI are related to the Graphical User Interface.

monique_core_Datastructures

Monique_core_Datastructures header and CPP files create all custom data structures that hold parameters. For instance, it holds FMOscData Struct which has parameters such as FM frequency, FM Swing and FM shape. It also holds their smoothed parameters, which are explained later. In this way, it provides a data structure for each element in the Monique Synthesizer.

monique_core_Parameters

This header and CPP file are mainly concerned with the definition of the Parameter class. This class handles the behavior for individual parameters of the synth. For example, the filter cutoff frequency would be an object of type Parameter.

This file defines behavior such as limiting value ranges, snaping values to min, max and zero, assigning modulation to parameters, reading, and writing of parameters from/to xml files for presets, setting initial values of parameters and how midi control messages are handled in relation to parameters.

monique_core_Processor

The header and CPP file of the core processor is responsible for creating various types of objects (e.g. UI, midi_control_handler, data_buffer, synth_data, synth, synth_voice), configuring them (by loading the parameters from MoniqueSynthData class created on Monique_core_Datastructures files), preparing to play and processing the audio. The same project is used for standalone and plugin applications, so there are conditional statements in the code that cause the synth to behave differently depending on how the software is deployed.

In the prepareToPlay() method, it mainly configures the audio that will be played, as well as resetting buffers and parameters. For instance, it sets the playback sample rate of the synth object and resets the internal state of the synth voice object.

The processBlock() method calls the process method for each audio sample buffer and MIDI message. The function process calls MoniqueSynthesizer::process_next_block method. The processing after that is explained in the Audio Processing Implementation section.

monique_core_Synth

This header and CPP file are responsible for the main functional blocks of the synthesizer. It defines the classes of MoniqueSynthesizer, MoniqueSynthesizerVoice and MoniqueSynthesizerSound.

The class MoniqueSynthesizerVoice inherits the JUCE synthesizerVoice class and contains classes related to sound generation and processing such as MasterOsc, FilterProcessor, LFO, ArpSequencer, etc. This class contains many important methods such as startNote and renderNextBlock.

The MoniqueSynthesizer class inherits the JUCE synthesizer class and is responsible for handling MoniqueSynthesizerVoice, MoniqueSynthesizer and midi events such as program change, sustain pedal and pitch wheel.

mono_AudioDeviceManager

This file configures the audio and MIDI devices connected to the project. The main class in this file is inherited from RuntimeListener, and if audio devices are used, it is also inherited from public juce::AudioDeviceManager with the help of Overloading Constructors. Since it is inherited from RuntimeListener class, it can be notified by the events occurring in run time.

dRowAudio_AudioUtility

This file is taken from dRowAudio and gives basic but useful methods that an audio programmer would need such as converting from midi number to hertz, or decibels to absolute values.

dRowAudio_Constants

This file is taken from dRowAudio and holds useful constants like the value root 2 or 2 pi.

monique_ui_AmpPainter

These files refer to the drawing of all the objects that compose Monique. In the header, the classes and variables for the objects are declared while in the CPP file these objects are drawn. The class monique_Ui_AmpPainter has methods for each main section of the synthesizer, they compare the values of the parameters with the inputs selected by the user and visualize them appropriately when the values change. This method also considers the resizing, repositioning and the coloring of the elements used in the interface according to the theme selected by the user.

monique_ui_LookAndFeel

The header contains the preset colors for the main theme palette of the application and some themes the user can choose to stylize Monique. In the CPP file, the values of the colors, themes and size for the objects are refreshed and then modified to be updated according to the configuration made during the manipulation of the parameters.

monique_ui_GlobalSettings

In the standalone version the user can select a preset theme or choose the color of each section of the User Interface. In that way, the header and CPP files oversee the general setup for the visual of Monique. Depending on the selection of the user, the section background, its text color, and the object color are updated independently or all at the same time. The choice of the color can be done by a color selector that is in the SETUP section, which is then sent to monique_ui_LookAndFeel where the style update of the application is done.

monique_ui_ENVPopup

This header and CPP file refer to the visualization of envelope for the filters. By clicking in the small circle in the top of the filter section, a pop-up window with the characteristics of the envelope will show allowing the user to modify the attack, sustain, decay and release.

monique_ui_DragPad

These files are responsible for drawing the slider changes according to the mouse movement and position. If the user clicks a slider and drags it to the right or the top, the slider bar will resize and increase respectively clockwise.

monique_ui_Morph

As the user is able to control the settings for the morph function, the header and CPP for the Morph User Interface are used to draw the EDIT visualization of the morph function. Here the user can select the programs for the left and right presets for the oscillator, filter, arpeggiator and FX. The CPP file includes the monique_core_Datastructures to update the value of the objects and thus visualize the changes in the objects.

monique_ui_Credits

These files are responsible for the visualization of the credits that are illustrated when the user move the mouse courser on the name of the Standalone application, by setting the color, font, size and position of each string, according to the size of the window.

Audio Processing Implementation Details

The processBlock method first calls the process method for each audio sample buffer and MIDI message, as explained previously.

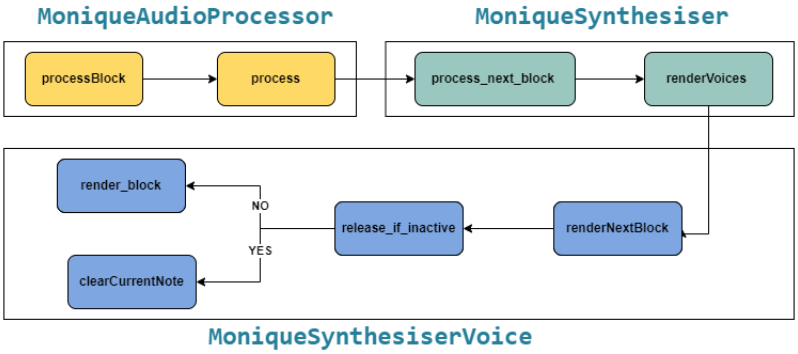

Figure 3: Block Diagram of the Audio Processing functions.

The method process, then calls MoniqueSynthesizer::process_next_block method.

MoniqueSynthesizer::process_next_block method starts the various envelopes, then for each voice it starts the release of their corresponding EQ and FX, and triggers the filter envelopes. Then, it calls the renderVoices method of the Synthesizer class which by default calls the renderNextBlock() for each voice.

The renderNextBlock method initially gets the playback position data for the audio, including the tempo and the number of samples. Then it iterates over the specified number of samples to render, checking for arpeggiator steps in the current buffer range for each block, computing the samples to the next step, and calls render_block() method to render the subsequent block depending on the determined parameters. The code also has logic for handling ARP control, such as retriggers, stops, and starts for the notes of the arpeggiator. If the audio playback is not currently playing, the function releases inactive voices and updates the relative samples since start.

The MoniqueSynthesiserVoice::release_if_inactive method is called in every renderNextBlock iteration. It checks if the envelopes are in the end state and if the arpSequencer is off. If these conditions are true, clearCurrentNote() is called and resets the voices.

Inside MoniqueSynthesiserVoice::render_block defines a SmoothExecuter struct. This includes pointers to LFO, OSC, ENV, and other smoothing-related data structures.

Next, the struct's exec method smooths the wave and phase shift arguments, calling smoothing and morphing methods. These are explained in the below sections. Then, process methods of the LFO, MasterOSC, SecondOSC, and FilterEnv are called.

Functional Block Analysis

In this section we will analyze the main functional blocks of this software synthesizer. The most unique and interesting functional blocks will be given greater attention.

Oscillators

Monique is a 3-oscillator synthesizer with a primary oscillator and two secondary oscillators. The primary oscillator is fixed to play the same note that is inputted by a midi note. The secondary oscillators can be detuned from this root note by ± 24 semitones.

The primary oscillator also provides control over the phase of the waveform. This is a nice additional parameter to have access to for sound design when combining the primary oscillator with the secondary oscillators.

Each of the oscillators allows the selection of a variable waveform, blending from Sine – Square – Saw – Noise.

FM Oscillator

Frequency Modulation can be achieved by using the FM oscillator. The amount of frequency modulation can be set individually for each oscillator. The FM oscillator can be set to operate at a multiple of the primary oscillator's frequency, from 2.01 – 8.01 times faster. The swing setting routes an LFO to the FM oscillator to achieve a vibrato like effect.

Filters

We have already introduced the operation of the filter section in the Block Diagram section above. We will now analyze the filters in greater detail.

Each of the three filter blocks is comprised of several functional blocks. First there is an input mixer for selection of the input signals and their gain. Then there is the actual filter, here the cutoff frequency and resonance can be set. The user can choose between Lowpass, Highpass, Bandpass or Bypass. Following filtering there is then a distortion effect and a panning effect before the output volume is set.

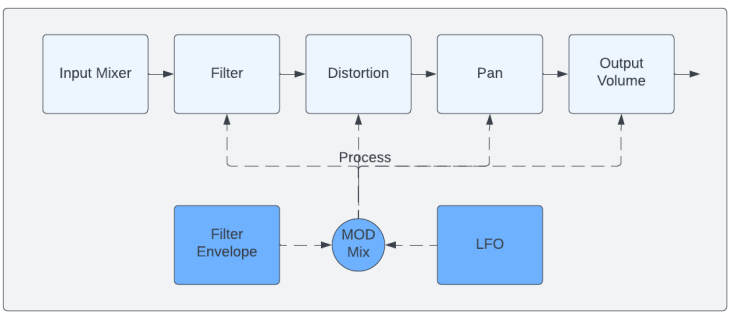

Figure 4: Block Diagram of the FilterProcessor.

We can see this functionality implemented in the Monique_core_Synth file in the FilterProcessor class. This class has control over all elements in the above block diagram. It utilizes the filter_data data structure to hold all relevant parameters. It also defines the mathematical function for implementing distortion:

```
x_ = x_ * (1.0f - distortion_power_) + (std::atan(x_ * 20) / 6.66) *
                                distortion_power_;
```

The plot of this function is shown in figure 5 with the distortion power set to 0.9.

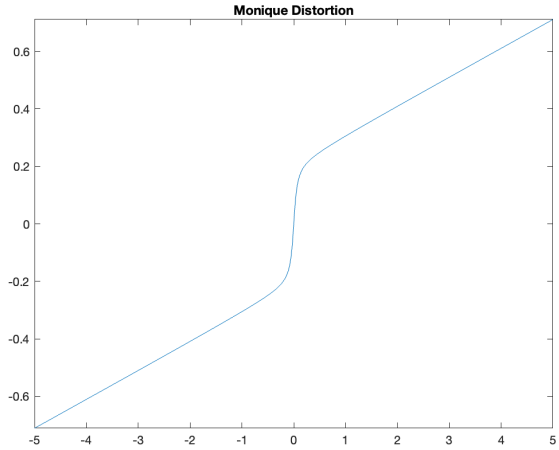

Figure 5: Monique's distortion function plotted in Matlab.

Process can be seen as the primary method of this class. When called it sets the input for the filter, the modulation handling and then uses a switch statement to choose between Low Pass, High Pass, Band Pass or Bypass.

Each of the parameters in the previous blocks can be modulated by the X-MOD. X-MOD is a linear combination of two modulators: filter envelope and LFO. This is shown in Fig.2. When mix is equal to 0.0, we can see that the amp_mix will only comprise of env_amps. As mix is increased the magnitude of env_amps is reduced and lfo_amplitudes is increased.

```
amp_mix[sid] = env_amps[sid] * (1.0f - mix) + lfo_amplitudes[sid] * mix;
```

Amplifier Envelope

After the output mix of the three main filters, we find the Amp Envelope stage, which controls the volume of the mixed filter signals before the EQ bank and FX section.

Equalizer Bank

The signal can be further shaped by a seven band EQ. For each band the user can set the resonance of the filter. Interestingly, the gain of each band can be controlled by its own individual envelope.

Effect Panel

Monique features a useful multi effects engine at the end of its signal chain. Here in series we have the effects of distortion, chorus, delay, looper and reverb.

Morph Mixer

The most unique functional block of Monique is the Morph Mixer. The Morph Mixer lets a user smoothly blend between presets like how a DJ mixes two or more records.

There are four sections in the morph mixer, Oscillator, Filter, Arpeggiator and Effects. For each section the user can select two presets they would like to be able to morph between.

Figure 6: Morph Mixer.

The Morph Mixer can alternatively be used to Morph from the current settings of the plugin to a previous state.

Smoothing Code Snippet

```
for (int sid = 0; sid != num_samples_; ++sid)
        {
            target[sid] = FORCE_MIN_MAX(simple_smoother.tick());
        }
```

The code above implements the smoothing operation for each sample to be processed. simple_smoother is an object of LinearSmoother class, which is defined in Monique_core_Parameters.h file. It basically increments the value with given step until target value. Thus, this interpolation provides a smooth transition for the values affected by the smoother. FORCE_MIN_MAX is just a custom function which ensures the signal stays in predefined boundaries and `target` is a pointer to the values that are to be smoothened.

Morphing Code Snippet

```
morph_power_smoother.set_value(morph_slider_state_);
      for (int sid = 0; sid != num_samples_; ++sid)
      {
              const float power_of_right = morph_power_smoother.tick();
              target[sid] = FORCE_MIN_MAX(left_morph_smoother.tick() *
              (1.0f - power_of_right) + right_morph_smoother.tick() * power_of_right);
      }
```

Morphing also follows the same idea with smoothing. After reading the morphing power value from its slider (between 0 and 1), smoother starts ticking as in the previous case (incrementing), and then `right_morph_smoother` which contains the values from first preset starts incrementing (ticking), by its corresponding step, and vice versa. Since the `power_of_right` is between 0 and 1 and `(1.0f - power_of_right)` is also between 0 and 1 but inversely proportional to `power_of_right`. Hence summation step gives us a mix between two different presets. The value of each parameter is then altered with the help of pointer target.

Our Python Implementation of morphing:

The code for our implementation is given in the Appendix. We implemented the morphing for one signal to another. Basically, the first signal (signal to be morphed), is converted to a target signal with small increments.

We use a sine wave with 400 Hz as our first signal, and a square wave with 100 Hz for our target signal. We also add a morphing_speed parameter that controls how fast is morphing done. Here are the plots before morphing, half morphed signal and fully morphed signal, respectively:

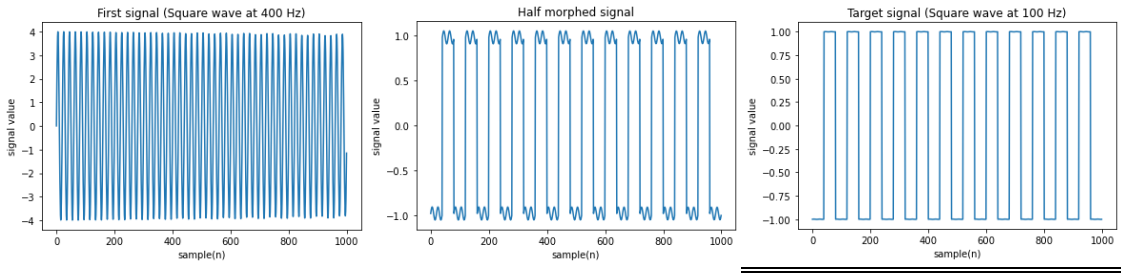

Figure 7: Morphing Explained. First signal, Half-morphed signal, and Fully-morphed signal

Arpeggiator:

Monique includes a 16-step arpeggiator/sequencer, referred to in the code as arpSequencer. For each step the user can select the note value and the velocity. The user also has control over the amount of swing/shuffle and the tempo of the arpeggiator.

The code for the arpSequencer is mainly split between Monique_core_Synth and Monique_synth_Datastructres. ArpSequencerData holds all information the sequencer needs. With arrays for steps, notes, velocity and parameters for shuffle, connect (sets legato between steps), speed_multi (sets speed of sequencer to multiple or division of master clock), step_offset (allows user to change the starting position of the sequencer) and fine_offset.

The shuffle parameter can have a value of 0-15 but is later converted when required into a metric subdivision ranging from 1/128th through to 7/8th. The shuffle feeling is achieved by adding a delay to the time when every second step of the sequencer should be triggered.

Figure 8: Arpeggiator.

Appendix

Figure 9: Overview of Monique GUI.

Python Code

```
import numpy as np
from scipy import signal
import matplotlib.pylab as plt
import sounddevice as sd

Fs = 8000
f = 100
sample = 80
x = np.arange(sample)
sine = 4*np.sin(2 * np.pi * f *4* x / Fs)
square = signal.square(2 * np.pi * f * x / Fs)

target_signal = square
diff = square - sine
update = diff

morphed = np.array([])
morphed_chunk = sine
morphing_speed = 0.003 # how fast morphing is
while True:
    morphed_chunk = morphed_chunk + morphing_speed*update
    morphed = np.concatenate((morphed, morphed_chunk))
    update = target_signal - morphed_chunk
    if np.mean(update**2) < 0.000001:
            break
sd.play(morphed, Fs)

plt.plot( morphed[-1000:-1])
plt.xlabel('sample(n)')
plt.ylabel('signal value')
```

```
plt.title('Target signal (Square wave at 100 Hz)')
plt.show()

plt.plot( morphed[0:1000])
plt.xlabel('sample(n)')
plt.ylabel('signal value')
plt.title('First signal (Square wave at 400 Hz)')
plt.show()

plt.plot( morphed[int(len(morphed)/2):int(len(morphed)/2)+1000])
plt.xlabel('sample(n)')
plt.ylabel('signal value')
plt.title('Half morphed signal')
plt.show()
```