CMLS: Second Homework
Banana Split

Circuitone
X.Luan, G.Costa, S.Stagno, A.Paoletti

May 22, 2023

Abstract

Banana Split is a FM synthesizer plug-in which has one oscillator with sine, saw, triangle and square wave and a fader by which you can control the oscillator gain. It also has a JUCE ADSR control and four effects: Reverb, Delay, Chorus and Distortion.

1 Introduction

1.1 Usage

Using the plug-in is very simple and intuitive. The presence of the keyboard at the bottom of the interface allows sounds to be generated even without the use of an external MIDI keyboard. The seven sliders then are immediate in their understanding and use: both the gain and the four parameters of the ADSR envelope and the two frequency modulation parameters are always in the foreground, ready to hand and without unnecessary complications. Finally, the four buttons on the right dedicated to the activation/deactivation of the corresponding four effects allow, with a single click, to add colour to one's sound without having to think about all the individual parameters that allow their implementation and functioning.

All this makes Banana Split a perfect plug-in for those who are approaching the world of synthesisers and want to have fun and play without too many pretensions, but without renouncing to an excellent quality.

1.2 Interface

The GUI is pretty basic but surely efficient. The basis is made up of the superimposition of two PNG files: a simple background and a glass image that gives to the GUI that nice glossy finish. These images are stored in the *Assets* folder and implemented in the main files that manage the GUI in a Juce plug-in, thus in *PluginEditor.cpp* and *PluginEditor.h.*

As shown in Figure 1, the GUI contains 3 main elements: a keyboard at the bottom, a series of seven sliders and four buttons in the upper part. The general positions of these three main elements are defined in the *PluginEditor.cpp* file. In order to implement the elements,

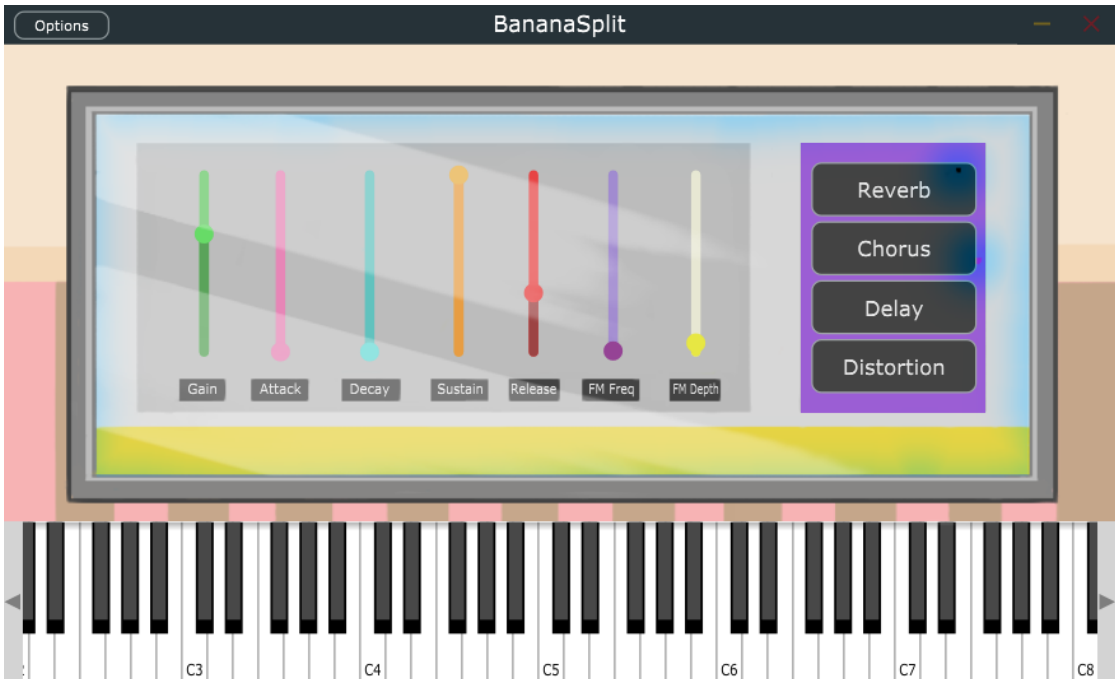


Figure 1: Graphical User Interface

the developers created a couple of files in the GUI folder for each type of item. Thus we have: *ControlsGUI.cpp* and *ControlsGUI.h*, that are related to the sliders; *FxGUI.cpp* and *FxGUI.h* dealing with the effect buttons and *KeyboardGUI.cpp* and *KeyboardGUI.h* that are obviously related to the keyboard. The sliders modify the values of the synthesizer's parameters: gain, attack, decay, sustain, release, FM frequency and FM depth. The buttons instead can be used to activate or deactivate the four effects: reverb, chorus, delay, and distortion.

Sliders and buttons' implementations follow the very same approach: in the header file a class that inherits from `juce::Component` it's declared and the following methods are declared too: the constructor, the destructor, paint and resized. Then, in the source files, this methods are defined and buttons and slider are given colours and labels, are made visible and a parameter is assigned to each element. An interesting thing to notice is that neither for the sliders nor for the buttons, parameters' values are shown to the user. In the case of the buttons, being the effect parameters' values predefined, the only thing to show is the on/off state of the effect and this is simply visualized thanks to the change of color of the single button. But for what concern the sliders, it could be useful for the user to have some numerical feedback of the sliders' action.

The keyboard's implementation it's a bit different: in the header file the main class, called `KeyboardGUI`, inherits from `juce::Component` but also from `juce::Timer`. This allowed the developers to use the `timerCallback`, `startTimer` and `stopTimer` methods in order to manage the states of the keyboard.

Finally, another thing to notice is that the developers defined the wave shape parameter in the *PluginProcessor.cpp* file, listing the following options: "Sin", "Square", "Triangle", "Saw". Despite that, there's no trace of the implementation of this parameter in the GUI. Indeed, the user hasn't the opportunity to change the wave shape.

2 MIDI Controller

The project, more than being a Juce plugin only, has also a hardware interface that enables the user to control the main parameters of the plugin also through physical buttons, offering an alternative to the available GUI displayed in the main window of the plugin running on the PC. The hardware components are:

- an Arduino Mega board;
- 2 protoboards and a certain number of cables to interconnect the devices;
- a USB cable to connect the Arduino board with a PC;
- 12 push buttons to simulate the 12 notes of one octave of the virtual piano keyboard;
- 6 potentiometers - these are used to control 6 parameters: Attack, Decay, Sustain, Release, FM Frequency and FM Depth;
- objects to touchpad converter (some conductor objects - in this particular case, candies - are used to control the effects of the plugin);
- 4 conductor objects - the candies of the previous point - used to control Reverb, Chorus, Delay and Distortion (the effects implemented in the plugin).

MIDI Hairless is the software used in this project in order to correctly receive and control MIDI messages.

3 Plugin Processor

As we already stated in the introduction, Banana Split is a Juce Plugin. Therefore, it contains the two fundamental classes of every Juce plugin: the `PluginEditor` and the `PluginProcessor` class.

Inside `PluginEditor` there are the main functions used to define the main window of the GUI, of which we will talk about later in the report.

In `PluginProcessor` instead, the most important functions are `prepareToPlay` and `processBlock`.

In `prepareToPlay` there are the operations that need to be done just before executing the plugin, in order to get it started. For instance, the sample rate of the synthesizer is set. Then, for every voice of the synthesizer, the `prepare` method gets called: this specific method belongs to the `SynthVoice` class and sets the sample rate for each voice of the synth, other than the samples per block and the number of input channels. The objects of the class `SynthVoice` contain also other attributes and methods, but they will be analyzed later in detail. Finally, in the `prepareToPlay` function, the effects are prepared resorting to an object of the class `ProcessSpec`, belonging to the `juce::dsp` module. This object is a struct, and it's useful when dealing with effects (as it happens in this case) as it contains the main parameters used to define them:

```
    juce::dsp::ProcessSpec spec;
    spec.sampleRate = sampleRate;
    spec.maximumBlockSize = static_cast<juce::uint32>(samplesPerBlock);
    spec.numChannels = 2;
    reverb.prepare(spec);
    chorus.prepare(spec);
    delay.prepare(sampleRate);
```

The second core function of the `PluginProcessor` class, `processBlock`, contains instead the operations that are done when executing the plugin.

Our `processBlock` function receives as arguments an object `buffer` of type `AudioBuffer<float>&` (or, in other words, references to objects of type `AudioBuffer<float>`) and an object `midiMessages` of the class `MidiBuffer`. The `buffer` object is a multi-channel buffer containing floating point audio samples. On this object, the method `clear` gets called: this method has more than one signature available; the one used in this particular case is the one that clears a specified region of one of the channels of the buffer. The method gets called in this way:

```
for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
    buffer.clear (i, 0, buffer.getNumSamples());
```

so this means that, if there are more output channels than input channels, each one of the ”exceeding” output channels gets cleared of all the samples contained in it.

After this, each voice gets its parameters updated through the `updateParameters` method: this method accepts as argument an object of type `AudioProcessorValueTreeState` (called here `apvt`). The objects of this class are useful to manage parameters that need to be controlled also from the GUI, as they have references to the parameters value: this is why the object `apvt` is used in this case. The parameters that we're talking about are *Type, Gain, ADSR, FMFreq* and *FMDepth.* These parameters will be later associated to a `SynthVoice` object.

Then, the MIDI messages and the audio buffer get processed by the synthesizer:

```
keyboardState.processNextMidiBuffer (midiMessages, 0, buffer.getNumSamples(),
true);
synth.renderNextBlock(buffer, midiMessages, 0, buffer.getNumSamples());
```

Another method defined in the `PluginProcessor` class is `MidiSynthAudioProcessor::createAPVT()`, that returns an object of class `juce::AudioProcessorValueTreeState::ParameterLayout`. An object of that class is an object that contains a set of RangeAudioParameters (meaning objects of the class `RangeAudioParameter`, an abstract base class from which different types of parametrized objects are derived). Then all the parameters that we need for our SynthVoice are added through the method `add`:

```
params.add(std::make_unique<juce::AudioParameterChoice>("Type", "Type", juce::
StringArray("Sin", "Square", "Triangle", "Saw"), 1));
params.add(std::make_unique<juce::AudioParameterInt>("Gain", "Gain", -60.0f,
12.0f, -12.0f));

params.add(std::make_unique<juce::AudioParameterFloat>("Attack", "Attack", 0.1
f, 1.0f, 0.1f));
params.add(std::make_unique<juce::AudioParameterFloat>("Decay", "Decay", 0.1f,
 1.0f, 0.1f));
etc...
```

The parameters contained in params, when all of them have been added, are:

- the type of the wave (sin, square, triangle or saw);
- the gain;
- ADSR (attack, decay, sustain, release);
- FM frequency and FM depth;
- the effects applied (reverb, chorus, delay or distortion).

4 The synthesizer

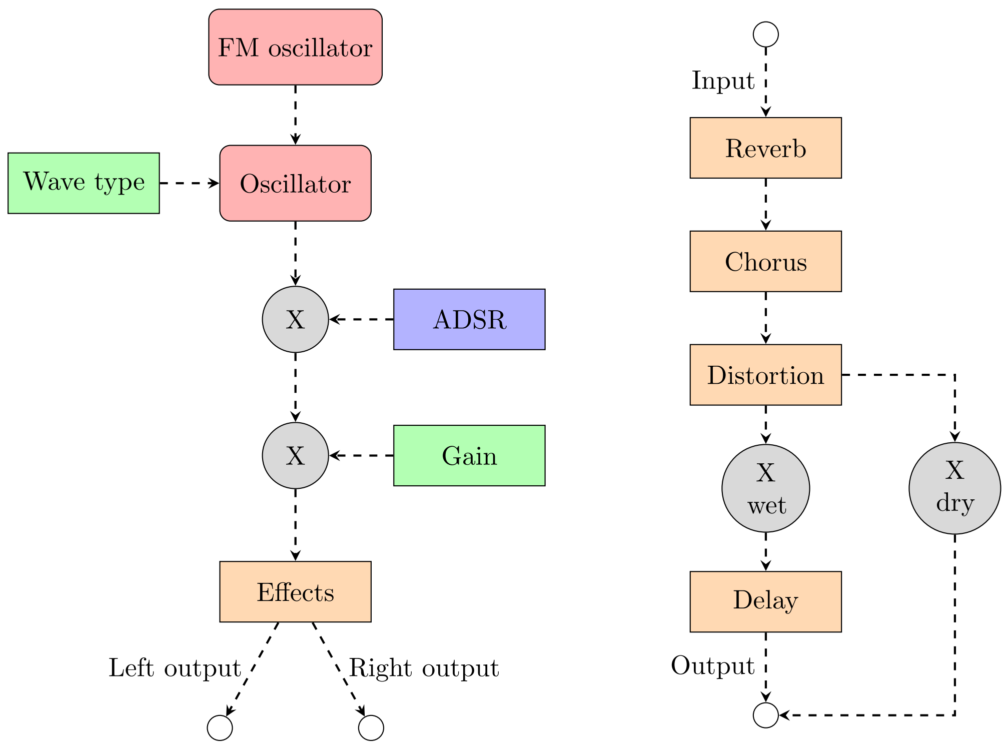


Figure 2: On the left is shown the general block diagram of the plugin, on the right the effects chain

Inside the Banana Split plugin processor (the file `PluginProcessor.h`), there is a reference to an object of the class `juce::Synthesiser`. This `Synthesiser` object will contain all the `SynthVoice` objects used for the sound generation. The SynthVoices will be added to the synth through the predefined method `addVoice` inside the constructor of the `PluginProcessor` class.

`SynthSound` and `SynthVoice` are the two classes used to deal with synthesizers in juce. In Banana Split the class `SynthSound`, which is derived from `juce::SynthesiserSound`, does not include many different components with respect to the base class from which it inherits. `SynthSound` objects too are added to the synthesiser from the `PluginProcessor` constructor.

On the other hand, the second class, `SynthVoice`, contains everything that we need to know about the basic sound generation of this plugin. A `SynthVoice` object contains as attributes:

- a `juce::dsp::Oscillator`, the basic Oscillator of the dsp module of Juce that we will use as the basic sound wave before that it gets “frequency modulated”;
- the gain value (of type `juce::dsp::Gain`);
- a `juce::ADSR` object, that, as the name suggests, contains the values of the sound envelope;
- a `juce::ADSRParameters` object;
- a `juce::AudioBuffer` object, that will contain the samples to be played (this will be useful especially inside the `PluginProcessor` class);

- finally, there are two parameters devoted to FMFrequency and FMDepth, other than the frequency modulation oscillator belonging to the class `juce::dsp::Oscillator`.

Inside the `SynthVoice` class, two of the most important methods are, for example, `startNote` and `stopNote`, whose functions are easily deducible by their own names. In `startNote` we can observe the changing of the frequency of the main oscillator due to the frequency modulation:

```
oscOne.setFrequency(frequency + fmMod);
```

Another operation performed by the `startNote` and `stopNote` functions is that of applying/removing the adsr filter. Then, the functions `prepare` and `updateParameters` take care of initialising the various parameters of the oscillator and updating them if they are changed.

The function `renderNextBlock` is probably the most important and most complicated one: it has the role of defining the interconnection between different synths. In this function, some major operations are performed:

- set the size of the `audioBuffer` called `synthBuffer`, contained in the private members of the `SynthVoice` class, with a number of channels equal to the output channels and with a number of samples given by the value `numSamples`, and then clear the buffer;
- create an `audioBlock` from the `synthBuffer` (an `audioBlock` is a data structure that contains pointers to audio buffers);
- for each sample of each channel, the modulation frequency is extracted and stored in the variable `fmMod`;
- finally, the adsr envelope is applied to the `synthbuffer` and the samples are added to the `synthbuffer`.

In the flow chart in figure 2 is shown the overall routing of the signal generation, from the oscillators to the output notes, and also the effects chain.

5 The effects

5.1 Reverb and Chorus

Chorus and reverb effects are implemented simply by exploiting the `juce::dsp` library. In this way, it is possible to achieve the effects using very few and very simple functions.

Both effects implement a `prepare` function, to which they pass a `ProcessSpec` structure containing the sample rate, the maximum size of a block of samples and the number of channels. In addition, both contain the `process` function, to which an `AudioBuffer` is passed, responsible for the actual functioning of the effect. The only implementation difference between the two effects is that the reverb also includes the few instructions necessary to set the gain to -12 dB. In particular the values that are set for the chorus and reverb effects respectively are the following[1]:

```
chorus.setRate(2.0f);
chorus.setDepth(0.9f);
chorus.setCentreDelay(50.0f);
chorus.setFeedback(0.7f);
chorus.setMix(0.7f);
```

```
float roomSize   = 0.5f;
float damping    = 0.5f;
float wetLevel   = 0.33f;
float dryLevel   = 0.4f;
float width      = 1.0f;
float freezeMode = 0.0f;
```

[1]As far as the Chorus is concerned, the parameters were decided by the programmer, while in the case of the Reverb, they are the default parameters in the library file *juce_Reverb.h*.

5.2 Delay

The delay effect was implemented using a circular buffer, i.e. a FIFO buffer, which is continuously read and updated, in order to play back samples generated in the past.

Apart from the `prepare` and `reset` functions, which set the correct sample rate and fill the circular buffer memory with zeros respectively, the `process` function takes care of replacing the samples processed by the delay algorithm for each channel and each sample. The algorithm that actually constitutes the delay consists of two main functions: `getInterpolatedSample` and `getDelayedSample`.

The first of these two functions takes care, as the name implies, to read from the `circularBuffer` the values corresponding to the index `readPosition` and `readPosition - 1`, taking care not to exceed the maximum size of this buffer, where the index is calculated as

```
float readPosition = delayIndex[channel] - inDelayTimeInSamples;
```

It then returns to the caller the value obtained by interpolating the two previous data points, using a `linearInterpolation` function.

The second of the two functions, as you can see in listing 1, actually implements the functionality required to calculate the delay time in samples, update the `circularBuffer` and `feedbackSample`, update the sample index in the main buffer and return the result of the computation. The delay time in samples is calculated by simply multiplying the time in seconds by the sample rate. This parameter is then passed to the `getInterpolatedSample` function, which returns the value called `out`. At this point the circular buffer is updated by adding the value calculated at the previous index, the `feedbackSample`, to the current index, multiplied by the `feedback` parameter (set by the programmer to 0.7). Now the `feedbackSample` can be updated with the newly calculated `out` value and the index can be incremented. Finally, the ultimate `out` value is calculated by weighing the original sample and the one calculated with the `dry` and `wet` parameters respectively, both set by the programmer to 0.5. In addition, this function also replaces the "traditional" values of delay time, feedback, wet and dry with a "smoothed" version of them, obtained from the function (in this case for the time parameter, but it's exactly the same for the other three)

```
smoothTime = smoothTime - SmoothCoefficient_Fine * (smoothTime - time);
```

where `SmoothCoefficient_Fine` is fixed to 0.002 and the initial value is set to 0. In this way, an imperceptible[2] increase in the delay effect is obtained, from zero to what is set by the programmer: for each sample, the four parameters are updated, resulting in values that are gradually closer and closer to the desired ones.

```
float Delay::getDelayedSample(float inSample, int inChannel)
{
    smoothTime[inChannel] = smoothTime[inChannel] - SmoothCoefficient_Fine * (smoothTime[inChannel] - time);
    smoothFeedback[inChannel] = smoothFeedback[inChannel] - SmoothCoefficient_Fine * (smoothFeedback[inChannel] - feedback);
    smoothWet[inChannel] = smoothWet[inChannel] - SmoothCoefficient_Fine * (smoothWet[inChannel] - wet);
    smoothDry[inChannel] = smoothDry[inChannel] - SmoothCoefficient_Fine * (smoothDry[inChannel] - dry);

    delayTimeInSamples[inChannel] = (smoothTime[inChannel] * sampleRate);
    const float sample = getInterpolatedSample(delayTimeInSamples[inChannel], inChannel);

```

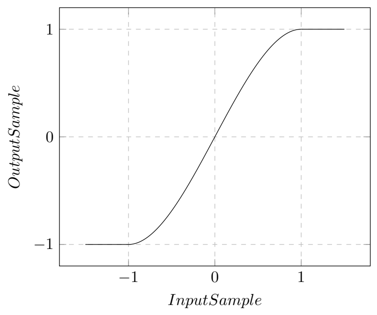
[2]By increasing the value of `SmoothCoefficient_Fine` from 0.002 to numbers such as 0.00001, the effect becomes much more noticeable.

```
        circularBuffer[inChannel][delayIndex[inChannel]] = inSample + (
    feedbackSample[inChannel] * smoothFeedback[inChannel]);
        feedbackSample[inChannel] = sample;

        float out = (inSample * smoothDry[inChannel] + sample * smoothWet[
    inChannel]);

        delayIndex[inChannel]++;

        if (delayIndex[inChannel] >= maxBufferDelaySize)
            delayIndex[inChannel] = delayIndex[inChannel] - maxBufferDelaySize;

        return out;
    }
```

Listing 1: getDelayedSample function

5.3 Distortion

What the developers mean by *Distortion* is in fact a type of soft-clipping, applied independently for each channel. In particular, the chosen transfer function uses a cubic polynomial, specifically the one in the equation 1. The output of this transfer function, shown in the Figure 3, is pre-multiplied by 5 and then scaled by 0.5.



$$f(x) = \begin{cases} -1 & x \leq -1 \\ \frac{3}{2}\left(x - \frac{x^3}{3}\right) & |x| < 1 \\ 1 & x \geq 1 \end{cases} \tag{1}$$

Figure 3: The soft-clipping transfer function.