



UNIVERSITÀ
POLITECNICA
DELLE MARCHE

FACOLTÀ DI INGEGNERIA

CORSO DI PERFEZIONAMENTO IN INGEGNERIA DEGLI STRUMENTI
MUSICALI

Progettazione e realizzazione di un Synth VST tramite il framework JUCE

Autore:

Leonardo Mannini

Relatore:

Prof. Leonardo Gabrielli

Anno Accademico 2021-2022

Indice

1	Introduzione	1
	Introduzione	1
2	Sintetizzatore	2
2.1	Le grandezze fondamentali	2
2.2	Cos'è la sintesi	3
2.2.1	Sintesi additiva	4
2.2.2	Sintesi sottrattiva	4
2.2.3	Sintesi FM	5
2.3	Componenti di un sintetizzatore	5
2.3.1	Oscillatore (VCO)	6
2.3.2	ADSR Envelope	6
2.3.3	Filtro (VCF)	7
2.3.4	LFO	8
2.3.5	Amplifier (VCA)	8
2.3.6	Effetti	8
2.4	Esempio: KORG Minilogue XD	10
3	Tecnologie utilizzate	12
3.1	Il framework	12
3.1.1	Ambiente di sviluppo	12
3.1.2	Perché C++?	13
3.1.3	Source Files, Headers, Librerie	13
3.2	Design Pattern MVC	13
3.3	VST	15
3.4	MIDI	15
4	leoSynth	17
4.1	Organizzazione del codice	19
4.2	Il Synth	20
4.2.1	Esempi di codice	20
5	Risultati e Conclusioni	24
5.1	Esempi audio	28
5.2	Release	28
5.3	Direzioni future	28

Capitolo 1

Introduzione

In questa relazione di progetto verrà presentata la progettazione e la realizzazione di un sintetizzatore virtuale chiamato “*leoSynth*” e realizzato con JUCE, un framework basato sul linguaggio C++ che fornisce metodi e librerie per l’elaborazione dei segnali audio digitali e per la realizzazione della relativa GUI (Graphic User Interfaces).

Saranno spiegati i concetti fondamentali alla base di un plugin audio digitale, alla base di un sintetizzatore classico nelle sue componenti più tradizionali, e verrà spiegata la loro effettiva implementazione tramite le tecnologie sopra elencate.

Nel capitolo 2 saranno illustrate le grandezze e i parametri utili a comprendere il funzionamento del sintetizzatore e le basi teoriche su cui si fondono le tecniche di sintesi più comunemente utilizzate. Poi, saranno introdotti le sezioni classiche presenti nella maggior parte dei sintetizzatori più commerciali, citandone la classificazione e il loro funzionamento. Infine, verrà illustrato un esempio (*KORG Minilogue XD*) per comprendere le componenti introdotte.

Nel capitolo 3 verranno trattati argomenti riguardanti le tecnologie utilizzate per la realizzazione del progetto: sarà presentato il linguaggio di programmazione utilizzato con il relativo framework, verranno giustificate le scelte di design e saranno brevemente introdotti i protocolli di comunicazione.

Infine, nel capitolo 4, verrà *finalmente* esposto il prodotto finale, ne sarà discussa l’architettura con alcuni esempi di codice rilevanti.

Nel capitolo conclusivo, saranno presenti alcuni esempi audio, riferimenti per reperire il codice sorgente ed una *release* funzionante, oltre ad alcune idee per l’ampliamento futuro del progetto.

Capitolo 2

Sintetizzatore

Un sintetizzatore è uno strumento musicale elettronico che genera segnali audio, sfruttando diverse tecniche di sintesi (additiva, sottrattiva, FM,...) per “costruire” diverse forme d’onda. Questi suoni vengono poi modificati da altri componenti come filtri, effetti, LFO ed altri, per ottenere timbri particolari. I sintetizzatori sono solitamente suonati attraverso tastiere o *sequencer* che comunicano tramite il protocollo MIDI (che verrà discusso nel capitolo 3).

2.1 Le grandezze fondamentali

Di seguito, sono presentate le grandezze fondamentali dell’elaborazione dei segnali audio, nonché per la comprensione dei fenomeni utilizzati in relazione ad un utilizzo del sintetizzatore in ambito prettamente musicale.

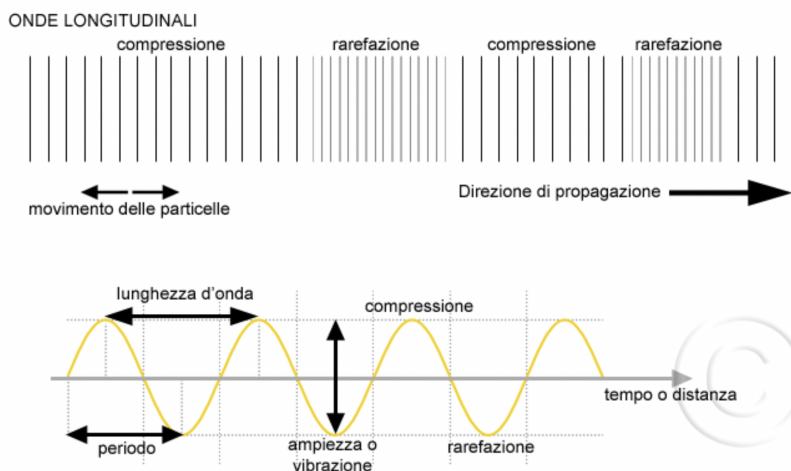


Figura 2.1: Le grandezze di un’onda

- *Periodo*: E’ il tempo t per cui si svolge una singola oscillazione.
- *Frequenza (frequency)*: La frequenza è una misura della ripetitività temporale (periodicità) di un’onda: quante volte nell’unità di tempo, per esempio in

un secondo, si ripetono in una posizione fissa i punti di massimo (o minimo) dell'onda.

- *Intonazione (pitch)*: E' la frequenza "percepita" dall'orecchio umano: è legata alla frequenza intesa in senso fisico, ma sono due concetti distinti. L'intonazione è un fenomeno soggettivo di percezione che dipende anche da altri fattori oltre alla frequenza come il timbro e l'intensità del suono ascoltato.
- *Forma d'onda*: è il profilo generato sul piano cartesiano, dove l'asse delle ascisse rappresenta il tempo e l'asse delle ordinate l'intensità del suono in dB (nel caso delle onde sonore).
- *Timbro*: Il timbro è ciò che distingue, all'orecchio umano, suoni diversi che suonano alla stessa altezza e intensità. E' la particolare qualità del suono che dipende da molti fattori, dallo spettro di fase e di ampiezza della serie delle armoniche e dal profilo dinamico (attacco, decadimento, sustain e rilascio - vedi ADSR) [1].
- *Aampiezza/Intensità*: L'ampiezza di un'onda sonora rappresenta il massimo spostamento, rispetto alla posizione di equilibrio, che le molecole del mezzo di propagazione compiono al passaggio dell'onda; al crescere dell'ampiezza, aumenta la forza con la quale viene colpito il timpano dell'orecchio e quindi l'intensità con cui il suono è percepito.
- *Fase*: La fase indica la posizione in cui si trova il ciclo dell'onda in un determinato istante.

Date due onde sinusoidali, si dicono in fase quando il loro ciclo è sincronizzato: in questo caso, la somma delle due onde è un'onda con ampiezza doppia. Viceversa, due onde si dicono in controfase quando il loro ciclo è opposto, ed in questo caso la somma delle due onde crea un annullamento delle due.

Due onde si dicono in discordanza di fase quando non sono né in fase né in controfase. In questo caso, la somma delle due onde crea un'onda con stessa frequenza ma con ampiezza dipendente dalla quantità di sfasamento tra le due.

2.2 Cos'è la sintesi

Con sintesi audio si intende la generazione di suoni, ovvero la creazione di segnali audio analogici o digitali.

La sintesi audio è utilizzata in diversi ambiti: da quello musicale, all'automazione di sistemi e processi (esempio: *text-to-speech*). Il plugin audio progettato discusso in questa relazione appartiene all'ambito musicale, fornendo di fatto un'interfaccia software per elaborare segnali nelle più semplici forme d'onda e controllare alcuni parametri (come *pitch*, *velocity*, *ADSR*, ...) tramite una tastiera o un sequencer esterni.

Esistono molte tecniche di sintesi sviluppate nell'ambito dell'ingegneria per la realizzazione di sintetizzatori, di seguito sono elencate le principali.

2.2.1 Sintesi additiva

La sintesi additiva è la tecnica più antica tecnica di sintesi [2], e si basa sulla somma di segnali sinusoidali per l'approssimazione di qualsiasi segnale più complesso, secondo la teoria legata all'analisi di Fourier: in breve, una qualunque funzione periodica può essere scomposta in una somma di infinite "opportune" funzioni o componenti sinusoidali (seno e coseno) dette armoniche [3].

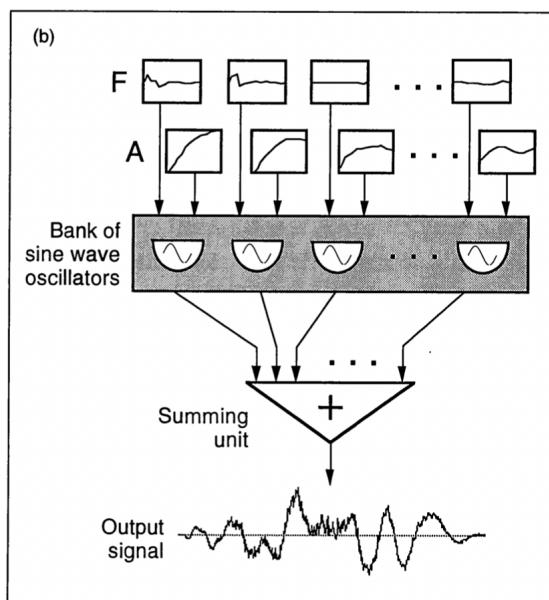


Figura 2.2: Somma di onde di diverso tipo in un segnale più complesso

La sintesi additiva è semplice nella realizzazione rispetto alle altre tecniche, ed anche intuitiva, ma richiede un grande numero di oscillatori ed operazioni al secondo per generare un suono elaborato e complesso.

Un esempio di strumento che utilizza la sintesi additiva, è il celebre organo Hammond.

2.2.2 Sintesi sottrattiva

Con la sintesi sottrattiva, a partire da un suono ricco di armoniche, si applicano dei filtri che permettono di escludere frequenze dello spettrogramma non desiderate ed ottenere il timbro ricercato.

Ad oggi, molti sintetizzatori utilizzano la sintesi sottrattiva, poiché permette di ottenere suoni piacevoli, elaborati, ricchi di armoniche, ad un prezzo minore, sia economicamente (per la componentistica) sia per la complessità di elaborazione del segnale.

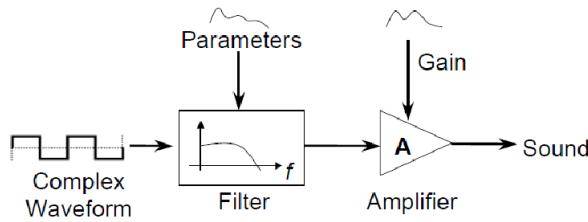


Figura 2.3: Sintesi sottrattiva: un segnale che viene filtrato e poi amplificato

2.2.3 Sintesi FM

La sintesi in Modulazione di Frequenza (FM) è un’ulteriore tecnica ad oggi molto utilizzata. Essa si basa sul modificare la frequenza di un segnale nel tempo, tramite un modulatore.

Un sintetizzatore che utilizza la sintesi FM è il Yamaha DX7, uno dei sintetizzatori con più successo commerciale di sempre.

Questa tecnica è presente nel *leoSynth* per modulare il *pitch* degli oscillatori.

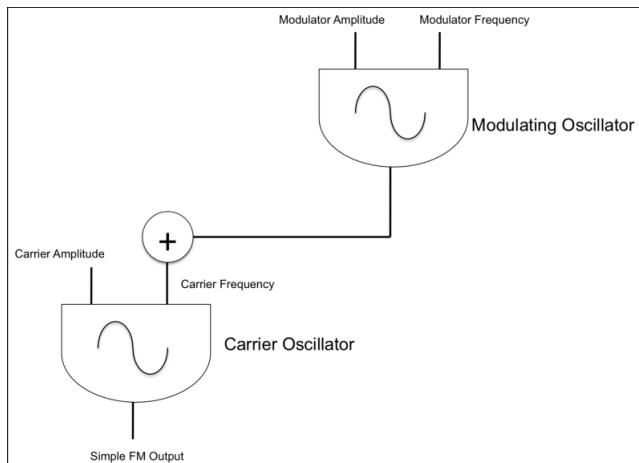


Figura 2.4: Sintesi FM: il segnale di un oscillatore “carrier” viene modulato dal segnale di un altro oscillatore “modulator”.

2.3 Componenti di un sintetizzatore

Nonostante la peculiarità di ogni sintetizzatore sia nella diversità delle tecnologie utilizzate e nelle connessioni tra i diversi moduli funzionali, è possibile identificare e descrivere quali sono le sezioni standard che fanno parte dei sintetizzatori più popolari, sia quelli vintage, sia i più moderni. Così facendo, sarà più facile dare progettare e descrivere il funzionamento del sintetizzatore *leoSynth*.

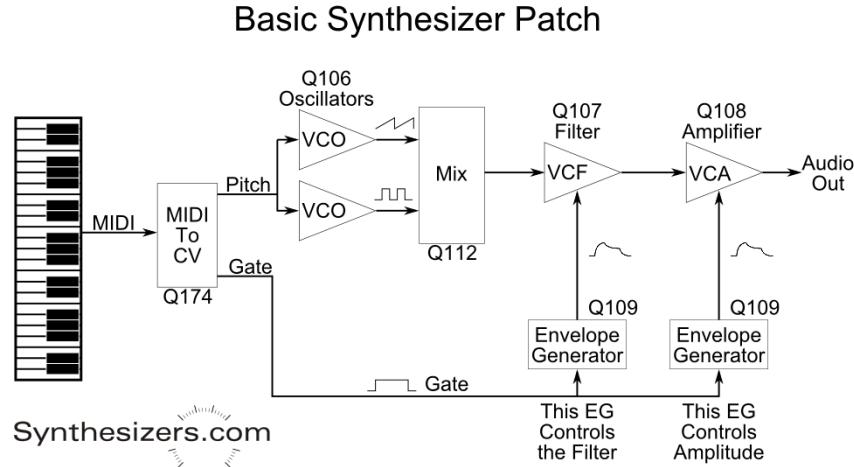


Figura 2.5: Esempio di routing in un sintetizzatore

2.3.1 Oscillatore (VCO)

L’oscillatore (Voltage Controlled Oscillator) è il componente del sintetizzatore che produce un segnale periodico, di norma un segnale sinusoidale, triangolare, quadrato o a dente di sega.

Soltanamente, è possibile scegliere una forma d’onda diversa per tutti gli n oscillatori, ed è possibile regolare il pitch di $n - 1$ oscillatori in base a quanto questi debbano discostarsi dalla frequenza dell’oscillatore principale.

2.3.2 ADSR Envelope

L’ADSR (Attack, Decay, Sustain, Release) Envelope è un componente che principalmente permette di controllare come il volume di un suono cambi nel tempo.

Oltre che il volume, l’ADSR Envelope può controllare anche ad altri parametri: ad esempio, nel leoSynth verrà utilizzato un altro componente ADSR per controllare l’inviluppo (l’andamento di una grandezza di un’onda dal momento in cui viene generata a quando si estingue) del filtro.

Generalmente, sono 4 i parametri controllabili da un ADSR:

- *Attack*: il *tempo* che impiega il parametro per raggiungere il suo livello massimo;
- *Decay*: il *tempo* che impiega il parametro per raggiungere il suo livello di *sustain* (vedi punto successivo);
- *Sustain*: il *livello* che si mantiene finché la nota non viene “rilasciata” (finché il tasto di un sintetizzatore non viene lasciato);
- *Release*: il *tempo* che impiega il parametro per passare dal livello di *sustain* fino al valore 0.

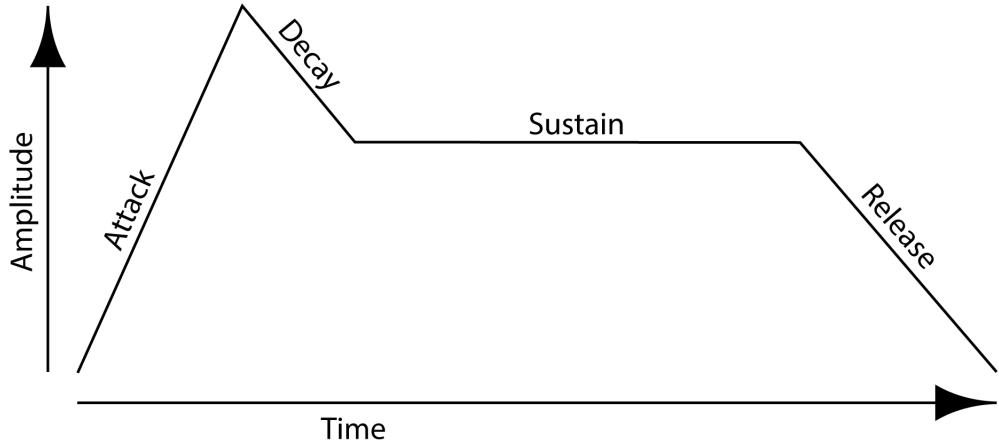


Figura 2.6: ADSR: Attack, Decay, Sustain, Release

Ad esempio, nel caso dell'*amplitude* (volume) di un suono si declina in questo modo: l'*attack* è il tempo per arrivare al volume massimo, il *decay* è il tempo per passare dal volume massimo al livello di *sustain*, il *sustain* è il livello di volume mantenuto fino quando non si tiene più premuta la nota, il *release* è il tempo necessario per passare dal livello di *sustain* allo 0.

2.3.3 Filtro (VCF)

Un filtro è un componente atto a modificare la timbrica del suono realizzato dal sintetizzatore.

Dato un segnale di input, il compito di un filtro è trasformarlo in un differente segnale di output: a seconda del tipo di filtro utilizzato, verranno quindi rimosse alcune frequenze caratteristiche del suono o accentuate altre.

Esistono diversi tipi di filtro:

- *Low-pass*: vengono “tagliate” le frequenze al di *sopra* di una certa soglia (*cutoff*);
- *High-pass*: vengono “tagliate” le frequenze al di *sotto* di una certa soglia (*cutoff*);
- *Band-pass*: viene accentuata una specifica frequenza ed attenuate tutte le altre;
- *Notch/Band-reject*: viene attenuata una specifica frequenza (non presente nel *leoSynth*);
- *Comb*: consiste in diversi filtri notch uno dopo l’altro, il cui grafico della risposta in frequenza ricorda il profilo di un pettine (non presente nel *leoSynth*).

La *resonance* è un parametro che deriva dagli originali filtri analogici realizzati. Nell'intorno della frequenza di cutoff, si genera un picco di volume che aggiunge sonorità alle frequenze in prossimità di quelle che vengono “tagliate”. La *resonance* regola, quindi, quanto il volume di queste frequenze venga accentuato.

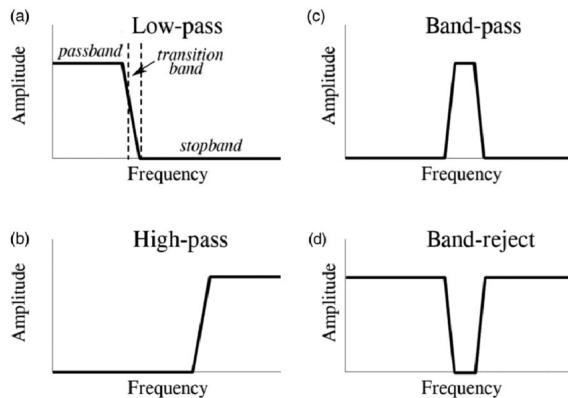


Figura 2.7: Tipi di filtro

2.3.4 LFO

L'LFO è un oscillatore a bassa frequenza (low-frequency oscillator): è un normale oscillatore che ha la peculiarità di generare frequenze molto basse rispetto agli altri oscillatori: solitamente sotto i 20Hz, la soglia udibile dall'orecchio umano (ovvero circa 20Hz-20.000Hz).

L'LFO è utilizzato per modificare in modo periodico alcuni parametri (a scelta dei costruttori di sintetizzatori) tramite il segnale generato dall'LFO. E' utile per realizzare effetti tipici degli strumenti acustici di vario tipo.

Per esempio, il segnale di un LFO può essere utilizzato per modificare nel tempo la frequenza di cutoff di un filtro (“*wobble*”), il pitch di un oscillatore (“*vibrato*”), il volume (“*tremolo*”), ecc. Non è stato implementato un LFO nel *leoSynth*.

2.3.5 Amplifier (VCA)

Un VCA (Voltage-Controlled Amplifier) è un componente che determina il livello di volume in output del segnale audio generato dal sintetizzatore.

2.3.6 Effetti

Gli effetti audio possono essere componenti dei sintetizzatori o stand-alone. Si utilizzano per alterare il suono di uno strumento musicale aggiungendo varie proprietà a seconda del tipo di effetto utilizzato.

Effetti dinamici

Gli effetti *dinamici* modificano il volume dello strumento.

- *Compressori*: servono ad attenuare il livello di volume di un suono, impostando una soglia massima. Inoltre, spesso sono utilizzati anche per stabilizzare la gamma dinamica di un canale audio, innalzando tramite un gain il volume quando troppo basso.

I compressori hanno un parametro, il *ratio*, che determina quanta compressione viene applicata: per esempio, se il ratio è 2:1, ad un segnale che eccede la soglia massima di 4dB verrà applicata una riduzione di 2dB. Se il ratio è $\infty:1$ (o comunque molto alto), il compressore è chiamato *limitatore*: esso di fatto imposta una soglia massima che non potrà essere oltrepassata, e tutto il segnale generato verrà compresso per ottenere un'intensità del suono sempre uguale o minore del valore desiderato.

- *Gate*: attenuano il volume di un segnale audio finché questo non supera un certo valore di soglia. In questo modo, è possibile eliminare una certa quantità di rumore nei momenti in cui lo strumento non sta suonando, ovvero nei momenti in cui quel rumore potrebbe risultare più udibile.

Effetti time-based

Gli effetti time-based si basano tutti sul principio del delay: riprodurre il segnale sonoro ritardandolo nel tempo.

- *Delay*: è realizzato prendendo il segnale audio suonato e riproducendolo dopo un certo intervallo di tempo. Il numero di volte per cui il segnale viene ripetuto è chiamato *feedback*.
- *Echo*: concettualmente è molto simile al delay, in quanto si tratta di segnale ripetuto dopo un certo intervallo di tempo; la differenza è che le successive “riflessioni” del segnale hanno dei tempi generalmente più lunghi e ad ogni riflessione il volume del suono diminuisce.
- *Reverb*: il reverb è un effetto utilizzato per simulare le riflessioni di un segnale prodotte in un determinato ambiente (ad esempio una chiesa o un teatro). A differenza del delay e dell'echo, il reverb simula moltissime più riflessioni (anche milioni) ed è per questo più pesante computazionalmente.

Altri effetti

Esistono innumerevoli altri tipi di effetti: per esempio, gli effetti di modulazione, che utilizzano un segnale *modulator*) per modulare il segnale *carrier* (*chorus*, *flanging*, *phaser*, *phase shifting*, etc.).

Non sono stati implementati effetti nel *leoSynth*.

2.4 Esempio: KORG Minilogue XD

Come esempio, viene di seguito visualizzata l’interfaccia di funzionamento di un KORG Minilogue XD e il suo relativo schema a blocchi, un sintetizzatore analogico molto comune presentato dall’azienda giapponese KORG nel 2016.



Figura 2.8: KORG Minilogue XD

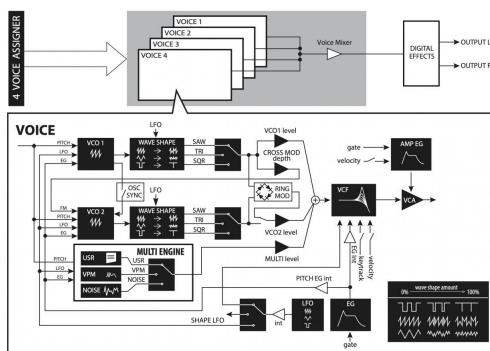


Figura 2.9: KORG Minilogue Block Diagram

Il sintetizzatore è diviso in quattro sezioni: i *controlli master*, i *controlli di sintesi*, gli *effetti*, e il *sequencer*.

Con *controlli master* si intendono parametri ed impostazioni “globali” della tastiera: il tempo, il volume Master, il portamento, l’ottava (per la tastiera) e il modulation stick (assegnabile a vari parametri a scelta).

Capitolo 2 Sintetizzatore

Nella sezione dei *controlli di sintesi* sono presenti gli oscillatori: due analogici e uno digitale. Per ognuno di essi è possibile modificare la forma d'onda, l'ottava, il pitch, ecc... Poi, c'è una banda per “mixare” gli oscillatori, ovvero regolare i loro volumi, singolarmente. Dopodiché, c'è il filtro di tipo *low-pass*, con i controlli per la frequenza di *cutoff* e per la *resonance*. E' possibile anche controllare il *drive* (un'amplificazione post-filtro) e il livello di *key-tracking* (attivare il *key-track* vuol dire rendere proporzionale il rapporto tra pitch e frequenza di *cutoff*). Poi, è presente la parte relativa ai *EG* (Envelope Generators: generatori di inviluppo), che permettono di controllare contemporaneamente il volume e uno a scelta tra il *pitch* degli oscillatori analogici e la frequenza di cutoff del filtro. Infine, è presente anche un *LFO* applicabile al *pitch* dell'oscillatore digitale, la forma d'onda o la frequenza di cutoff del filtro.

Gli *effetti* presenti (digitali) sono: *modulation*, *reverb* e *delay*, utilizzabili in modo esclusivo fra di loro e regolabili tramite due controlli *time* e *depth*.

Il *sequencer* è un dispositivo che permette di registrare dati di automazione per registrare informazioni sull'audio da riprodurre (note, velocity, e qualsiasi altro parametro). Nel caso del *Korg Minilogue*, il *sequencer* permette di salvare informazioni su 16 “step”, salvando per ognuno l'automazione di 4 manopole a scelta, permettendo quindi di realizzare suoni con un timbro nettamente diverso per ogni step.

Capitolo 3

Tecnologie utilizzate

In questo capitolo, introdurremo il framework utilizzato, l'ambiente di sviluppo, il linguaggio utilizzato assieme ad alcune sue peculiarità rilevanti ed il design pattern scelto. Poi, saranno introdotti i protocolli *VST* e *MIDI*.

3.1 Il framework

Per sviluppare il *leoSynth*, è stato adottato JUCE, un framework open-source, cross-platform basato sul linguaggio C++, molto utilizzato nello sviluppo desktop e mobile [4] di plugin audio grazie alla presenza di librerie audio e GUI particolarmente intuitive ed efficaci.

JUCE permette di scrivere del codice multipiattaforma che può essere compilato sia su Linux, sia su Windows, sia su MacOS.

L'organizzazione del codice e le funzionalità di JUCE verranno discusse più approfonditamente nei capitoli successivi.

3.1.1 Ambiente di sviluppo

Projucer è un'applicazione compresa nel framework JUCE che gestisce le dipendenze dei progetti JUCE, gestisce la creazione, la modifica, e la cancellazione di nuovi file di progetto, e di impostare il tipo di applicazione desiderata (plugin, synth, ...).

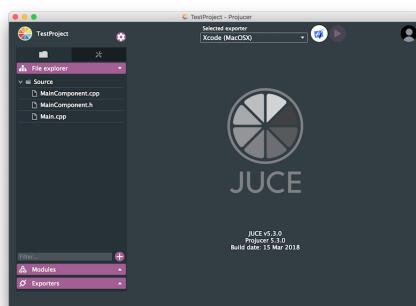


Figura 3.1: Il Projucer

L'IDE (*integrated development environment*) usata nel caso del synth “leoSynth” è *Xcode* per Mac OS, mentre per compilare il progetto su Windows è stato utilizzato Visual Studio 2022.

3.1.2 Perché C++?

C++ è un linguaggio di programmazione multi-paradigma (quindi estremamente flessibile), ed è un linguaggio fondamentale nello sviluppo di applicazioni audio. E' un linguaggio *compilato*, ovvero i programmi vengono generati da un programma esterno chiamato *compilatore* che traduce le istruzioni scritte in C++ dal programmatore in linguaggio macchina.

Oltre ai linguaggi *compilati* (come C++ o C), esistono linguaggi *interpretati*, come Python o Java, e sono, solitamente, linguaggi con un livello più alto di astrazione. Tuttavia, possono essere inefficienti nel caso di operazioni che richiedono tempi di elaborazione molto brevi (come nel caso di una routine di sintesi audio) [5].

Il motivo per cui è solitamente utilizzato C++, quindi, è che la compilazione permette dei tempi di esecuzione molto più brevi rispetto ad altri linguaggi compilati, oltre al fatto che, essendo un linguaggio multi-paradigma e multi-purpose, si adatta molto bene a diverse finalità e di paradigmi di progettazione.

3.1.3 Source Files, Headers, Librerie

Nel linguaggio C++, si distinguono tre diversi tipi di file:

- *Source Files*: sono file di testo contenente codice sorgente in C++, che verranno poi utilizzati per *compilare* il programma.
- *Headers*: sono file che contengono definizioni di parti di codice che verranno poi implementate successivamente nei Source Files: questo rende più leggibile il codice e, se la progettazione è coerente in tutte le parti del programma, permette di risparmiare errori impedendo al programmatore di modificare la struttura delle classi durante lo sviluppo.
- *Librerie*: sono dei file contenenti funzioni, definizioni di strutture dati, etc. che vengono aggiunti al programma nella fase chiamata *linking*, e vengono utilizzati perché, se già presenti nel linguaggio/framework, permettono di risparmiare tempo nella stesura del codice: per esempio, nel framework JUCE, sono già presenti librerie per la manipolazione audio e per la creazione di interfacce grafiche.

3.2 Design Pattern MVC

Un design pattern è una strategia con cui organizzare il codice di un programma (tipicamente ad oggetti). E' fondamentale da decidere in fase di progettazione quali

pattern implementare in un qualsiasi progetto per garantire leggibilità, accelerare lo sviluppo stesso, e garantire maggior durabilità nel tempo.

Per il progetto realizzato, si è deciso di adottare il design pattern *Model-View-Controller (MVC)* [6]

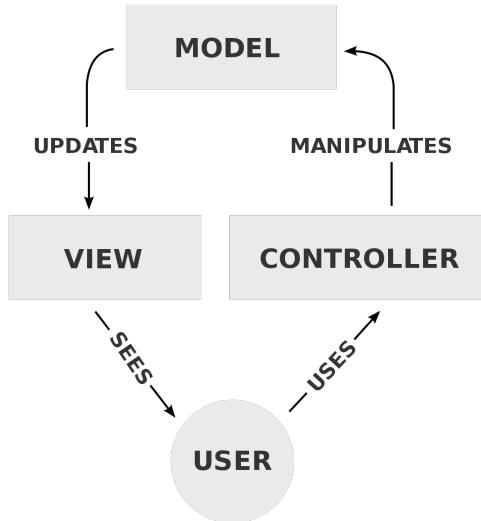


Figura 3.2: MVC: Modello, Vista e Controller

Il MVC è uno dei design pattern più utilizzati, ed è basato su 3 moduli funzionali interagenti fra di loro:

1. *Model*: le componenti del modello si occupano di mantenere informazioni e metodi sui dati, indipendentemente dall'interfaccia utente.
2. *View*: A partire dai dati immagazzinati nel *model*, la *view* sceglie come questi dati devono essere visualizzati dal punto di vista dell'utente.
3. *Controller*: è il “cervello” dell'applicazione, accetta gli input degli utenti e li converte in comandi da elaborare per poi indirizzarli verso il *Model* o le *View*.

La scelta del MVC per questo tipo di progetto, quindi, permette di dividere così il *leoSynth*: nel *model* vi sono i vari componenti del sintetizzatore (oscillatori, filtri, ecc...), il loro funzionamento, le loro caratteristiche e i loro metodi, nelle *view* verranno descritte le modalità con cui questi componenti verranno visualizzati a schermo e come l'utente potrà interagire con essi, e nel *controller* compariranno tutte le funzioni relative al routing del segnale audio generato, alla connessione fra i *model* e le *view* e all'impostazione ed all'aggiornamento di parametri fondamentali (modificabili dall'utente con l'interazione con la *view*).

3.3 VST

VST (Virtual Studio Technology) è un protocollo sviluppato da Steinberg e rilasciato nel 1996, nato come metodo per aggiungere effetti audio in real-time a tracce audio registrate digitalmente.

Il formato diventò rapidamente popolare data la sua natura *open*, per cui altre aziende cominciarono a vedere il suo potenziale e sviluppare software VST.

I plugin VST vengono generalmente aperti in una DAW (Digital Audio Workstation) come Ableton, Cubase, FL Studio.

Esistono tre tipi di VST:

1. *VST instruments*: generano audio, possono essere o sintetizzatori virtuali o campionatori virtuali. Alcuni di questi riproducono il suono e le scelte tecnologiche di sintetizzatori hardware più famosi.
2. *VST effects*: processano invece di generare audio, svolgendo la stessa funzione di processori audio hardware (come un reverb).
3. *VST MIDI effects*: processano i messaggi MIDI e indirizzano le informazioni MIDI ad altri strumenti o plugin.

3.4 MIDI

MIDI (Music Instrument Digital Interface) è il principale protocollo di comunicazione tra strumenti digitali e DAW. Nacque nel 1982, con la necessità delle diverse aziende produttrici di strumenti musicali di avere un linguaggio comune con cui far comunicare le proprie macchine.

Il linguaggio MIDI è basato su messaggi che vengono scambiati dopo un evento come la pressione di un tasto. Esso, non rappresenta il suono stesso, ma rappresenta una sequenza di istruzioni e le caratteristiche del suono che poi il sintetizzatore utilizzerà per generare il suono.

Grazie a ciò, il linguaggio MIDI è molto leggero in memoria ed quindi è rapido lo scambio di messaggi MIDI.

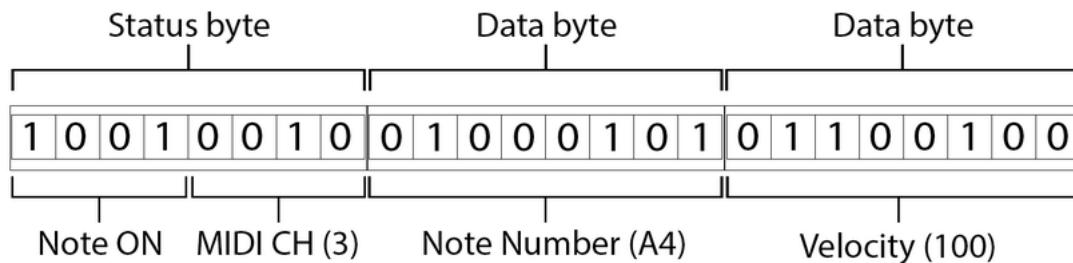


Figura 3.3: Esempio di MIDI message

Capitolo 3 Tecnologie utilizzate

I messaggi MIDI possono essere degli *STATUS byte* o dei *DATA bytes*: nel primo caso, il bit 7 è impostato a 1 e nel secondo è impostato a 0. Lo *STATUS Byte* determina il tipo di messaggio, il *DATA Byte* determina le informazioni caratteristiche di quel messaggio. Lo status byte, inoltre, contiene informazioni riguardanti il canale midi utilizzato: infatti, è possibile avere fino a 16 canali midi in comunicazione contemporaneamente, permettendo cioè di gestire 16 strumenti diversi.

I messaggi scambiabili con il protocollo MIDI sono **NOTE ON** e **NOTE OFF**, che rappresentano la pressione e il rilascio di un tasto di un sintetizzatore.

Nel caso del **NOTE ON**, le informazioni inviate nel *STATUS Byte* sono la comunicazione del tipo di evento (**NOTE ON**), e il canale. Le informazioni inviate nel *DATA Byte* sono il numero identificativo della nota, e la velocity, ovvero la forza di pressione del tasto. Il segnale **NOTE OFF** invia le stesse informazioni del **NOTE OFF** ma al posto della velocity comunica la release velocity.

Alcuni eventi MIDI: **NOTE ON**, **NOTE OFF**, **PAUSE**, **Aftertouch Polyphonic**, **Control Change**, **Program Change**, **Channel Aftertouch**, **Pitch Bend Change**, ecc.

Capitolo 4

leoSynth

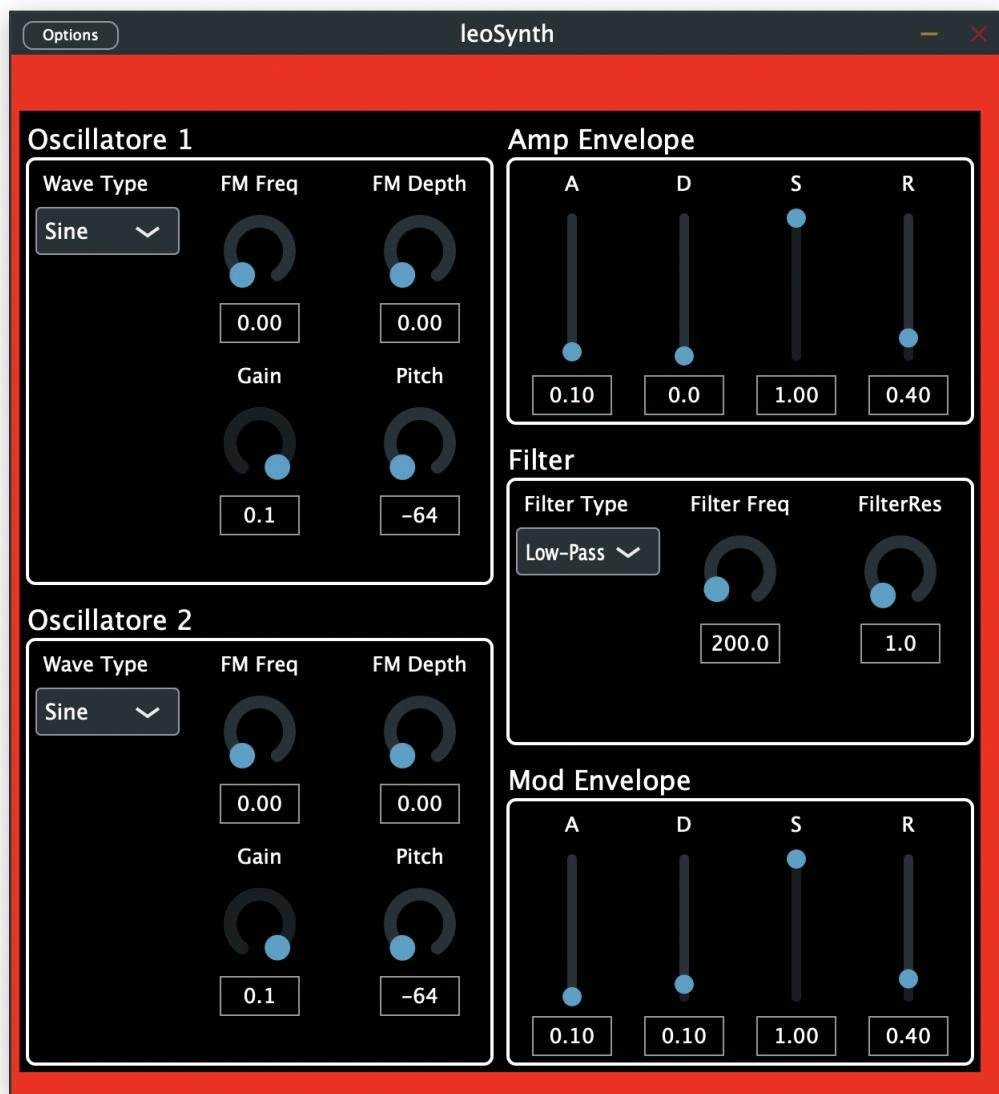


Figura 4.1: Interfaccia del leoSynth

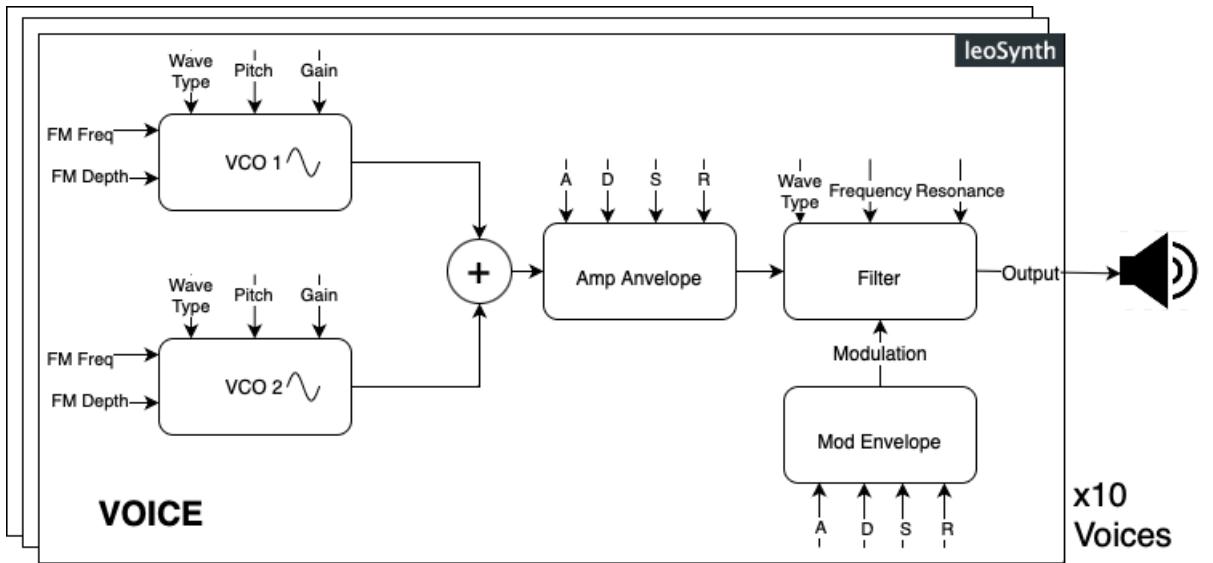


Figura 4.2: leoSynth: Block Diagram

Il *leoSynth* ha due oscillatori. In ognuno di essi, è possibile modificare la forma d’onda, scegliendo fra *Sine* (sinusoide), *Saw* (dente di sega) e *Square* (onda quadra). Inoltre, le manopole “*FM Freq*” e “*FM Depth*” permettono di aggiungere una modulazione di frequenza (*FM*) sul segnale generato, così da ottenere un effetto *vibrato*. Poi, è possibile regolare il *gain* e il *pitch*. I segnali generati dagli *oscillatori* vengono poi sommati fra loro.

Successivamente, è presente un primo *ADSR*, componente che controlla l’inviluppo per il livello di volume. Poi, il segnale passa per un filtro che può essere *low-pass*, *high-pass* e *band-pass*. In ogni caso, possono essere impostate la frequenza di *cutoff* e la *resonance*.

Infine, la sezione *Mod Envelope* è un altro *ADSR* che agisce sul filtro: l’*attack* regola il tempo impiegato dal filtro per raggiungere, dal valore 0Hz, la frequenza di *cutoff*; il *decay* il tempo per raggiungere il livello di *sustain*, una seconda frequenza su cui si “assesterà” il filtro, per poi, una volta rilasciata la nota, tornare nuovamente a 0 in un tempo dipendente dal *release*.

Questa procedura, può essere svolta contemporaneamente con un massimo di 10 voci. Il segnale, così elaborato, verrà poi inviato in output.

4.1 Organizzazione del codice

Come già detto, il codice è organizzato secondo il pattern MVC. Come visibile nell'immagine 4.3¹, nella root folder dell'applicazione è presente il *controller*, ovvero il file `PluginProcessor`.

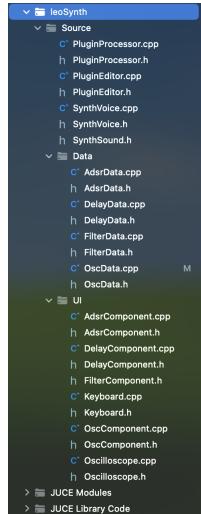


Figura 4.3: Controller, View, Data (Model)

Oltre al controller, sono presenti altri file nella root folder:

1. *PluginEditor*: Di fatto è il *view* del programma, imposta l'interfaccia grafica dal livello d'astrazione più alto possibile, ma, nel farlo, utilizza componenti che vengono dichiarati e configurati nella cartella *UI*. Di fatto, quindi, questo file serve da “*aggregatore*” di tutti i componenti visuali del programma.
2. *SynthSound*: E’ una classe che rappresenta il suono prodotto dal sintetizzatore, ma, di fatto, il rendering audio viene poi fatto dalla classe *SynthVoice*. In generale, un suono potrebbe essere composto da tante voci diverse fra loro (ognuna processata in modo differente): nel caso del *leoSynth*, il suono sarà composto da dieci voci ma tutte dello stesso tipo: di conseguenza, *SynthSound* è una semplice eredità dalla classe `juce::SynthesiserSound` [7].
3. *SynthVoice*: Questa classe rappresenta una *voce* del sintetizzatore: effettua tutto il rendering relativo ad una singola voce: rappresenta ciò che il `juce::Synthesiser` può utilizzare per riprodurre un `juce::SynthesiserSound`. Una voce tiene un singolo suono alla volta, e il `Synthesiser` ha un’array di voci che possono suonare in modo polifonico.

¹Nel codebase risultano alcune classi che non verranno discusse nel codice, come `Keyboard`, `Oscilloscope` e `DelayComponent`: questi componenti non sono ancora presenti nella versione attuale e sonon in via di sviluppo per futuri aggiornamenti: la versione discussa comprende i seguenti componenti funzionali: Oscillatori, ADSR, Filtro, e ADSR Filter Modulator.

4.2 Il Synth

Ad ogni componente del *leoSynth* è associato un file Data (*Model*), contenente proprietà e metodi di ogni componente, ed un file Component (*View*), che descrive il modo in cui i componenti saranno visibili dall’utente. Quindi, la logica e il funzionamento del sintetizzatore sono descritti nel file *PluginProcessor*, che ha la funzione di “collegare” la parte visuale ai dati dei componenti.

Di seguito, verrà analizzato il procedimento logico alla base del funzionamento di tutti i componenti del programma, in questo caso per scegliere la forma d’onda di un oscillatore, al fine di illustrare il percorso funzionale e comprendere la scelta architettonica presa e le sue conseguenze.

4.2.1 Esempi di codice

Il file *header* contiene la dichiarazione di tutte le proprietà e metodi dell’oscillatore: eredita un oggetto `juce::dsp::Oscillator` dalle librerie *JUCE*, un `Gain`, l’ultima nota suonata e i parametri necessari per la modulazione FM. Tra i metodi, notiamo quello che aggiornerà la forma d’onda (`setWaveType`), che viene riportato più avanti.

```
class OscData : public juce::dsp::Oscillator<float>
{
public:
    void prepareToPlay (double sampleRate,
                        int samplesPerBlock,
                        int outputChannels);
    void setWaveType (const int choice);
    void setGain (const float levelInDecibels);
    void setPitch (int pitch);
    void setWaveFrequency (const int midiNoteNumber);
    void getNextAudioBlock (juce::dsp::AudioBlock<float>& block);
    void updateFm (const float freq, const float depth);
    float processNextSample (float input);

private:
    void processFmOsc (juce::dsp::AudioBlock<float>& block);
    juce::dsp::Oscillator<float> fmOsc {
        [] (float x)
        { return std::sin (x); } };
    juce::dsp::Gain<float> gain;
    float fmMod { 0.0f };
    float fmDepth { 0.0f };
    int lastMidiNote { 0 };
};
```

Il metodo `OscData`, data una scelta su 3 valori disponibile all’utente nell’interfaccia principale, restituisce valori generati dal codice riportato di seguito.

```
void OscData::setWaveType (const int choice)
{
    switch (choice)
```

```
{
    case 0:
        // Sine
        initialise ([]( float x) {
            return std::sin (x);
        });
        break;
    case 1:
        // Saw wave
        initialise ([]( float x) {
            return x / juce::MathConstants<float>::pi;
        });
        break;
    case 2:
        // Square wave
        initialise ([]( float x) {
            return x < 0.0f ? -1.0f : 1.0f;
        });
        break;
    default:
        jassertfalse; // You're not supposed to be here!
        break;
}
}
```

La *view* ha, nell'*header*, la dichiarazione di tutti i vari componenti visuali che verranno visualizzati a schermo, con i relativi “Attachment” (connessione fra oggetti di input e variabili)

```
class OscComponent : public juce::Component
{
public:

    OscComponent (juce::AudioProcessorValueTreeState& apvts,
                  juce::String waveSelectorId,
                  juce::String fmFreqId,
                  juce::String fmDepthId,
                  juce::String gainId,
                  juce::String pitchId,
                  juce::String oscName);
    ~OscComponent() override;
    void paint (juce::Graphics&) override;
    void resized() override;
    void setTitle(juce::String newTitle);
    juce::String getTitle();

private:
    juce::String title;
    juce::ComboBox oscWaveSelector;
    std::unique_ptr
        <juce::AudioProcessorValueTreeState::ComboBoxAttachment>
```

Capitolo 4 leoSynth

```

    oscWaveSelectorAttachment;
    juce::Slider fmFreqSlider;
    juce::Slider fmDepthSlider;
    juce::Slider gainSlider;
    juce::Slider pitchSlider;
    //etc ...
    //...
};
```

Il *source code* della *view* contiene semplicemente istruzioni per impostare il layout del componente.

```

void OscComponent :: paint (juce :: Graphics& g)
{
    auto bounds = getLocalBounds().reduced (5);
    auto labelSpace = bounds.removeFromTop (25.0f);
    g.fillAll (juce :: Colours :: black);
    g.setColour (juce :: Colours :: white);
    g.setFont (20.0f);
    g.drawText (OscComponent :: getTitle(),
                labelSpace.withX (5), juce :: Justification :: left);
    g.drawRoundedRectangle (bounds.toFloat(), 5.0f, 2.0f);
    //etc ...
    //...
    //...
}
```

Nel *controller*, inizialmente vengono dichiarati le variabili modificabili direttamente dall'utente finale, che vengono tutti memorizzati in una struttura dati chiamata **AudioProcessorValueTreeState** (**apvts**). Una volta inizializzato un parametro, identificandolo con un ID e assegnandogli un nome, è possibile accederci e leggere il valore regolato dall'utente (tramite l'**Attachment**), per poi passare il valore ai metodi del *Model* (in questo caso, **osc1.setWaveType(oscWaveChoice)**).

Quest'operazione viene svolta per ogni parametro di ogni componente di ogni voce. Alla fine, si effettua un aggiornamento di tutte le componenti del synth (**adsr.updateADSR**, **filterAdsr.updateADSR** e **voice->updateFilter**).

```

//...
params.push_back (std :: make_unique<juce :: AudioParameterChoice>
    ("OSCWAVETYPE2", // parameter id
     "Osc2WaveType", //parameter name
     juce :: StringArray {
        "Sine",
        "Saw", //possible values names,
        "Square"

    }, 0)); //standard value is 0 (sine)
//ecc ecc...
//...
//...
for (int i = 0; i < synth.getNumVoices(); ++i)
```

Capitolo 4 leoSynth

```
{  
    if (auto voice = dynamic_cast<SynthVoice*>(synth.getVoice(i)))  
{  
  
        //OSC  
        auto& oscWaveChoice = *apvts.getRawParameterValue  
            ("OSCWAVETYPE");  
        auto& osc1Pitch = *apvts.getRawParameterValue  
            ("OSC1PITCH");  
        //etc...  
        //stessa cosa per gli altri parametri  
        //...  
        for (int i=0; i<getTotalNumOutputChannels(); i++)  
        {  
            osc1[i].setWaveType(oscWaveChoice);  
            osc1[i].setPitch(osc1Pitch);  
            //...  
            osc2[i].setWaveType(oscWaveChoice2);  
            osc2[i].setPitch(osc2Pitch);  
            //...  
        }  
        adsr.updateADSR(/* ... */);  
        filterAdsr.updateADSR(/* ... */);  
        voice->updateFilter(/* ... */);  
    }  
}
```

Capitolo 5

Risultati e Conclusioni

Di seguito sono mostrati alcuni grafici in tempo e/o frequenza di alcuni segnali generati dal *leoSynth*, con relativa schermata d'impostazione del plugin.

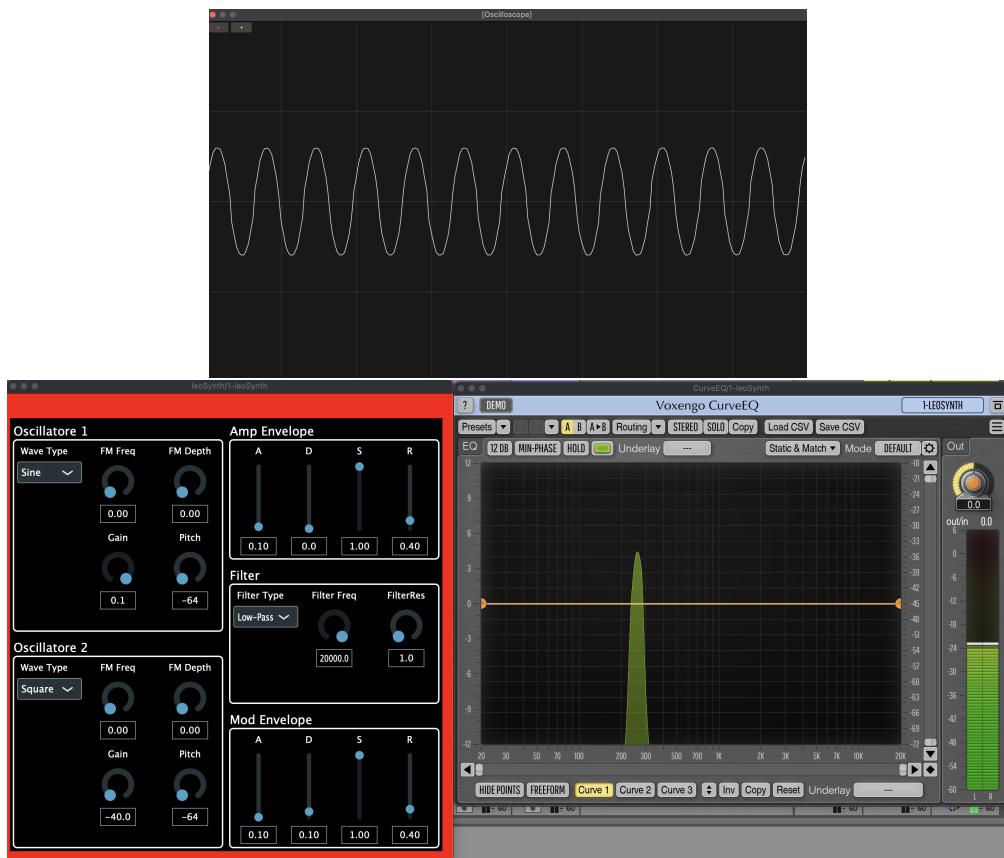


Figura 5.1: Sine Wave: semplice sinusoide. Un unico picco compare alla frequenza fondamentale (DO4, 262Hz)

Capitolo 5 Risultati e Conclusioni

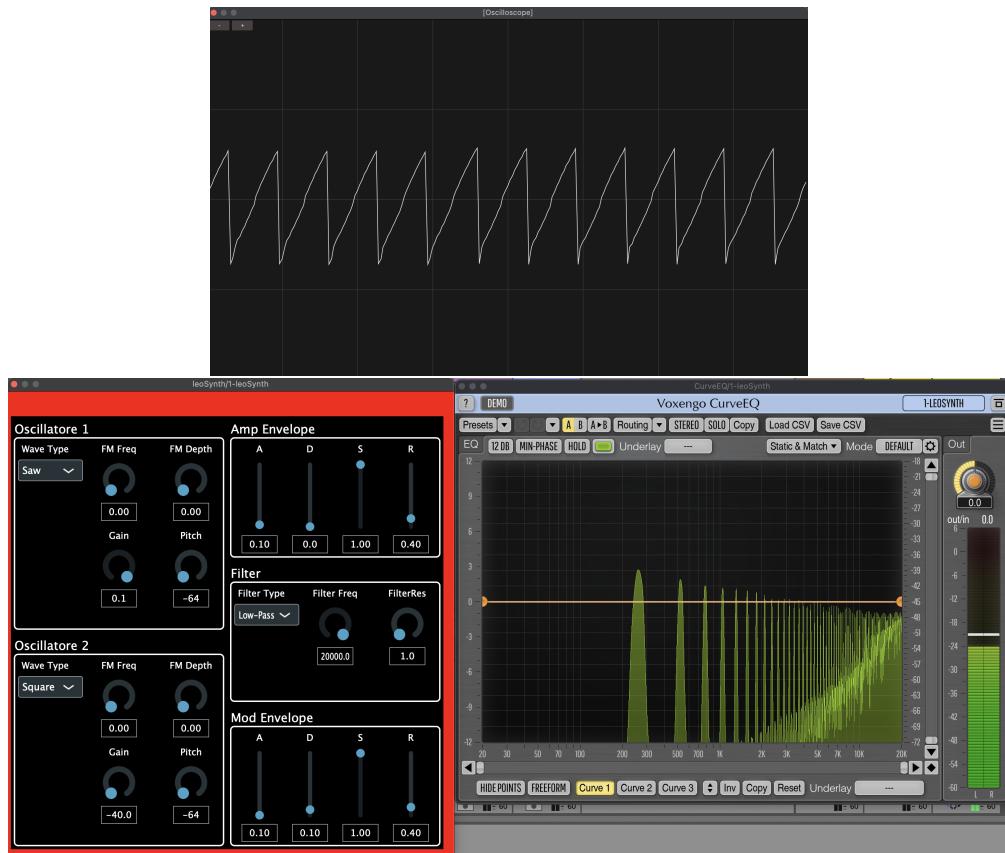


Figura 5.2: Sawtooth Wave: dente di sega. il suo spettro contiene armoniche sia pari sia dispari della frequenza fondamentale. Poiché contiene tutte le intere armoniche, è una delle migliori forme d'onda da utilizzare per la costruzione di altri suoni utilizzando la sintesi sottrattiva. (DO4, 262Hz)

Capitolo 5 Risultati e Conclusioni

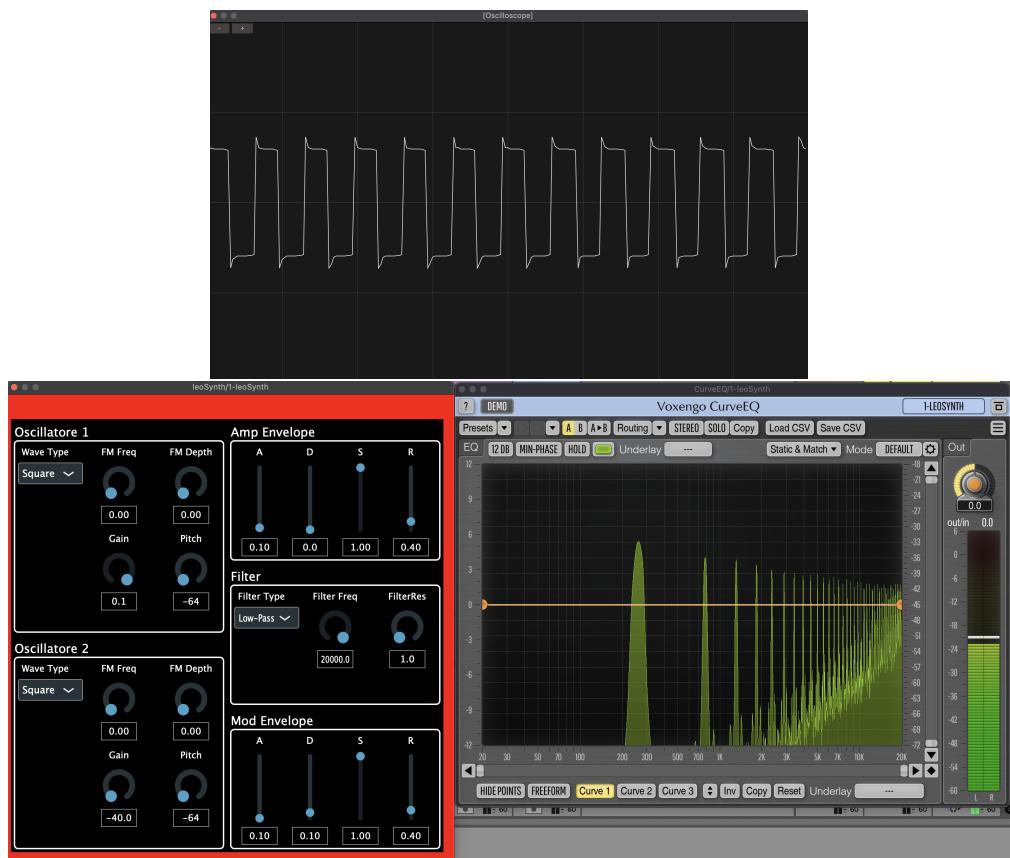


Figura 5.3: Square Wave: onda quadra. Nel suo spettro sono presenti esclusivamente le armoniche dispari. (DO4, 262Hz)

Capitolo 5 Risultati e Conclusioni



Figura 5.4: Square Wave + Filter Resonance @ 5kHz: Il segnale generato dall'onda quadra viene tagliato dalla frequenza 5kHz in poi, con un'enfasi di 9.7. (DO4, 262Hz)



Figura 5.5: Chord: Square + Saw + Filter Resonance @5kHz. Accordo di Do Maggiore con enfasi intorno alla frequenza 5kHz. (DO4, 262Hz) + (MI4, 330Hz) + (SOL4, 392Hz)

5.1 Esempi audio

Sono state effettuate alcune registrazioni per illustrare la gamma timbrica e il complessivo risultato ottenuto con lo sviluppo del *leoSynth*.

Le tracce sono ascoltabili su <https://github.com/leonardoman9/SYNTH/tree/master/recs>, oppure dai seguenti link:

- Test componente per componente;
- *J.S. Bach: Preludio in Do maggiore BWV 846*: in questa registrazione è stato aggiunto del riverbero in post-produzione.

5.2 Release

Il file .vst3 è stato compilato su Xcode 14.0 e può essere aperto tramite un qualsiasi DAW che supporti il formato.

Il software offre supporto nativo per controller USB esterni, è sufficiente caricare il plugin in un canale audio e fornirgli in input il controller desiderato.

Il source code e la Release possono essere visualizzati e scaricati dal seguente link: <https://github.com/leonardoman9/SYNTH>.

5.3 Direzioni future

Allo stato attuale, il *leoSynth* è costituito dai moduli funzionali più elementari per un sintetizzatore. In futuro potranno essere implementati altri componenti fondamentali: per esempio, un LFO applicabile al gain od al pitch degli oscillatori, oppure alla frequenza di cut-off del filtro.

Inoltre, potranno essere aggiunti alcuni effetti elementari ereditabili dalle librerie di JUCE: *delay*, *reverb*, *echo*... per aggiungere profondità e spazialità al suono.

Infine, nel codice sono già presenti file del *model* per sviluppare un oscilloscopio (integrato direttamente nell'interfaccia del plugin) ed una tastiera virtuale per utilizzare lo strumento senza l'ausilio di un controller esterno.

Elenco delle figure

2.1	Le grandezze di un'onda	2
2.2	Somma di onde di diverso tipo in un segnale più complesso	4
2.3	Sintesi sottrattiva: un segnale che viene filtrato e poi amplificato . .	5
2.4	Sintesi FM: il segnale di un oscillatore “carrier” viene modulato dal segnale di un altro oscillatore “modulator”.	5
2.5	Esempio di routing in un sintetizzatore	6
2.6	ADSR: Attack, Decay, Sustain, Release	7
2.7	Tipi di filtro	8
2.8	KORG Minilogue XD	10
2.9	KORG Minilogue Block Diagram	10
3.1	Il Projucer	12
3.2	MVC: Modello, Vista e Controller	14
3.3	Esempio di MIDI message	15
4.1	Interfaccia del leoSynth	17
4.2	leoSynth: Block Diagram	18
4.3	Controller, View, Data (Model)	19
5.1	Sine Wave: semplice sinusoide. Un unico picco compare alla frequenza fondamentale (DO4, 262Hz)	24
5.2	Sawtooth Wave: dente di sega. il suo spettro contiene armoniche sia pari sia dispari della frequenza fondamentale. Poiché contiene tutte le intere armoniche, è una delle migliori forme d'onda da utilizzare per la costruzione di altri suoni utilizzando la sintesi sottrattiva. (DO4, 262Hz)	25
5.3	Square Wave: onda quadra. Nel suo spettro sono presenti esclusivamente le armoniche dispari. (DO4, 262Hz)	26
5.4	Square Wave + Filter Resonance @ 5kHz: Il segnale generato dall'onda quadra viene tagliato dalla frequenza 5kHz in poi, con un'enfasi di 9.7. (DO4, 262Hz)	27
5.5	Chord: Square + Saw + Filter Resonance @5kH. Accordo di Do Maggiore con enfasi intorno alla frequenza 5kHz. (DO4, 262Hz) + (MI4, 330Hz) + (SOL4, 392Hz)	27

Bibliografia

- [1] Wikipedia contributors. Timbre — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=Timbre&oldid=1110965988>, 2022.
- [2] C. Roads. *The Computer Music Tutorial*. Mit Press. MIT Press, 1996.
- [3] Wikipedia contributors. Fourier analysis — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Fourier_analysis&oldid=1098381000, 2022.
- [4] Wikipedia contributors. Juce — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=JUCE&oldid=1108138804>, 2022.
- [5] R. Boulanger, V. Lazzarini, and M.V. Mathews. *The Audio Programming Book*. MIT Press, 2010.
- [6] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.
- [7] Juce documentation. <https://docs.juce.com/master/index.html>. Accessed: 2022-09-13.
- [8] Wikibooks. Sound synthesis theory/introduction — wikibooks, the free textbook project. https://en.wikibooks.org/w/index.php?title=Sound_Synthesis_Theory/Introduction, 2018.
- [9] Wikipedia contributors. Subtractive synthesis — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Subtractive_synthesis, 2022.
- [10] J. Chowning. *The Synthesis of Complex Audio Spectra by Means of Frequency Modulation*. Journal of the Audio Engineering Society, 1973.
- [11] Wikipedia contributors. Virtual studio technology — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Virtual_Studio_Technology&oldid=1110009999, 2022.
- [12] Arpege Music - Dominique Vandenneucker. Midi tutorial. <https://www.cs.cmu.edu/~music/cmsip/readings/MIDI%20tutorial%20for%20programmers.html>, 2012.