UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

DIPARTIMENTO DI INFORMATICA

Corso di Laurea in: INFORMATICA MUSICALE

Sviluppo di un plugin MPE con JUCE

Elaborato finale di: Simone Domenico Malacrida

Matricola: 840120

Relatore: Prof. Luca A. Ludovico

Correlatore: Ing. Emanuele Parravicini

Anno accademico: 2015/2016

Indice

1. Introduzione

A partire dalla fine dell'Ottocento, la tecnologia ha avuto un notevole impatto in ambito musicale. In questi anni vengono inventati innovativi dispositivi di registrazione e di riproduzione di suoni, sempre più perfezionati con il passare del tempo. È in questo periodo, infatti, che nascono i primi giradischi, altoparlanti, microfoni e stazioni radio, riscuotendo un grande successo commerciale e a livello di diffusione: la musica diventa parte sempre più rilevante della vita delle persone.
Anche per quanto riguarda il panorama degli strumenti musicali vi sono state notevoli innovazioni, grazie all'elettrificazione di strumenti già esistenti (quali chitarre, pianoforti, ecc…) e allo sviluppo di sintetizzatori analogici, che aumentarono esponenzialmente il numero di timbriche ottenibili.
La svolta tecnologica definitiva si ha negli anni Ottanta del Novecento, grazie alla digitalizzazione della musica e, in particolare, allo sviluppo del Compact Disc.
I vantaggi derivanti dal digitale erano evidenti, su tutti la maggiore stabilità e resistenza al deterioramento del supporto fisico del prodotto musicale e l'abbattimento dei costi in fase di produzione: viene ad instaurarsi quindi uno stretto legame tra musica e informatica, destinato a rafforzarsi ulteriormente nel corso degli anni successivi; infatti, oggi è possibile trovare numerosi software musicali di qualsiasi tipo: player di riproduzione, workstation per la registrazione e il missaggio di brani, editor di forme d'onda, editor di partiture e via dicendo.
Non irrilevante il fatto che queste tecnologie siano ormai accessibili a tutti, per cui è possibile realizzare produzioni musicali valide per conto proprio, senza dover ricorrere allo studio di registrazione professionale, servendosi del proprio computer per generare suoni di ogni tipo, anche in real-time: questa affascinante situazione ha portato allo sviluppo di varie tecnologie che mirano alla realizzazione di software che si comportano come veri e propri strumenti musicali.
Numerose sono le tecniche di sintesi sonora per la realizzazione di questa categoria di strumenti musicali, dalla classica sintesi additiva, dove tutto nasce da una sinusoide, alla sintesi per campionamento, dove si creano librerie di suoni registrati da strumenti reali, alle modellazioni fisiche mediante algoritmi che provano a spiegare il comportamento di un particolare strumento musicale e che permettono, in secondo luogo, di simulare in modo convincente la timbrica di ipotetici strumenti, inesistenti o irrealizzabili (per esempio un violino con manico di 60 metri e cassa armonica delle dimensioni di un barattolo).

Partendo dal presupposto del continuo sviluppo di nuove tecnologie informatiche, e dello stretto legame che esse hanno ormai da più di 30 anni con la musica, lo scopo di questo elaborato consiste

nello sviluppo di un software sotto forma di plugin di strumento musicale virtuale. Esso sarà programmato in linguaggio C++ attraverso il framework JUCE 4.
Inoltre, tale strumento avrà espressività polifonica e multidimensionale, ossia compatibile con le specifiche della più che recente tecnologia MPE (Multidimensional Polyphonic Expression).
Lo strumento virtuale sarà un sintetizzatore, contenente più oscillatori che generano una tra varie forme d'onda, sommate tra loro mediante una selezionabile modalità di sintesi.

1.1 Cenni sullo standard MIDI

Per meglio comprendere quella che è la tecnologia MPE, occorre accennare le caratteristiche fondamentali di quella tecnologia su cui pone le fondamenta, ossia lo standard MIDI (Musical Instrument Digital Interface).
Il MIDI è un protocollo standardizzato nato per la comunicazione e l'interazione tra strumenti musicali elettronici; sviluppato negli anni Ottanta, nonostante sia un protocollo ormai datato, rimane ancora intensamente utilizzato soprattutto nella musica elettronica e nella progettazione di piattaforme hardware (come tastiere, controller, ecc…). Le ragioni per cui questo standard trova ancora spazio sul mercato potrebbero essere legate a questioni di praticità di utilizzo, piuttosto che alla dimensione, in termini di byte, dei messaggi o delle rappresentazioni musicali, oppure alla volontà dei produttori sia di strumenti musicali elettronici, sia di hardware e software di mantenere la multimedialità dei propri prodotti.
È possibile implementare il protocollo MIDI in fase di sviluppo software: tutti i programmi in commercio, siano essi workstation o semplici plugin, conoscono il MIDI, che permette quindi la comunicazione tra le parti.
Una delle caratteristiche fondamentali del MIDI è l'essere un'interfaccia multicanale: vengono supportati fino a 16 canali, per permettere che il messaggio inviato tramite questo protocollo arrivi solo e soltanto alla destinazione connessa al canale specificato.

Come già accennato, i dispositivi MIDI comunicano tra loro attraverso messaggi: ogni messaggio è composto da parole di 8 bit, di cui la prima rappresenta il byte di stato (obbligatorio), seguita da n byte di dati a seconda delle caratteristiche del byte di stato.

- Il byte di stato, in particolare, identifica il tipo di informazione inviata via MIDI, e viene riconosciuto dal bit più significativo posto a 1. I successivi 3 bit identificano la funzione (000 =

Note Off, 001 = Note On, …), e gli ultimi 4 bit determinano il canale a cui il messaggio è destinato.

- Il byte di dati contiene valori numerici il cui significato dipende dal byte di stato. In questo caso, il bit più significativo è posto a 0, mentre i rimanenti 7 bit determinano il valore del parametro, in un range da 0 a 127.

Vi è inoltre da distinguere quelli che si definiscono channel messages, specifici per il singolo canale e system messages, rivolti a tutti i dispositivi collegati al sistema.

In questo elaborato verrà fatto spesso riferimento ai messaggi di Note On, Note Off e Pitch Bend, che ora verranno analizzati più in dettaglio.

- Un messaggio di Note On indica al dispositivo in ascolto che deve suonare una particolare nota, e viene generato alla pressione di un tasto della tastiera. È composto da un byte di stato (1001 xxxx, con xxxx = numero di canale) e due messaggi dati che codificano l'altezza (pitch) e l'intensità (velocity).

  La codifica del pitch si basa sul sistema temperato, per cui, ad esempio, note come Do# e Reb hanno la stessa frequenza. Le 128 note rappresentabili vanno da C-1 a G9, dove il numero indica l'ottava (C4 rappresenta il Do dell'ottava centrale, che ha valore di pitch = 60).

  La velocity, messaggio che alcuni dispositivi non riescono a leggere (come le testiere non pesate, in questo caso hanno un valore di default = 64), diversamente, è connessa all'intensità di un suono, il che non necessariamente si riferisce al volume: una nota di uno strumento a bassa intensità non è detto che mantenga le caratteristiche timbriche della stessa ad alte intensità, viceversa accade per il volume. Si pensi a un flauto, piuttosto che ad un sassofono, le cui note suonate ad alte intensità introducono molte armoniche e i così detti "overtones", oppure ad un pianoforte che scaturisce un suono più brillante quando si esercita una pressione maggiore su un tasto.

  Questa è una importante considerazione in fase di sviluppo di strumenti virtuali, dove occorre assegnare campioni differenti per una stessa nota (in una sintesi a campionamento), o meglio, creare algoritmi che descrivano il comportamento dello strumento nelle diverse situazioni di intensità (in una sintesi a modelli fisici), perché il software sia il più realistico possibile.
- Un messaggio di Note Off è simile al messaggio di Note On. Uno status byte (1000 xxxx) seguito da due data byte che identificano altezza della nota da spegnere e velocity di Note Off (spesso non utilizzata).

  A causa della somiglianza, non è raro trovare dispositivi che interpretano questo messaggio come

un Note On con velocity = 0, dato che un suono con intensità nulla non viene percepito dall'orecchio umano.

- Il messaggio di Pitch Bend viene generato da un controller che fa variare la frequenza della nota base, ottenendo una sorta di effetto di glissando. Nelle tastiere classiche è assegnato ad una rotella solitamente situata sulla sinistra, a lato dei tasti; nei controller moderni, in particolare modo quelli che sfruttano la tecnologia MPE, la rotella è stata rimossa e per ottenere l'effetto tipico del Pitch Bend, basta premere un tasto e, senza rilasciarlo, muovere il dito verso gli altri tasti fino a raggiungere quello corrispondente alla nota di arrivo.

  Il messaggio di Pitch Bend, oltre all'obbligatorio status byte (1110 xxxx), possiede due data byte per graduare accuratamente l'intervallo per fare in modo che la variazione di altezza risulti lineare, ed evitare di generare un non proprio orecchiabile "effetto gradino" nel passaggio tra una frequenza e l'altra. È intuitivo il fatto che se è minore l'intervallo di Pitch Bend, maggiore sarà la risoluzione a disposizione e si otterrà un effetto di glissando più gradevole.

Una importante premessa da tenere in considerazione per gli argomenti successivi di questo elaborato: i dispositivi hardware MIDI più comuni, generalmente sintetizzatori o comunque controller nella forma di tastiera di pianoforte, creati fin dagli anni Ottanta ricevono le sequenze di note e di messaggi su un solo canale, attraverso il quale il sintetizzatore comunica con le altre interfacce MIDI. Da subito i produttori di hardware avevano capito che il ristretto numero di canali era limitante per alcuni tipologie di strumenti polifonici e riduceva le possibilità espressive, in particolare il controllo totale per ogni nota, forzando di conseguenza l'utilizzo di più canali accorpati per un solo strumento. Esempi possono essere le chitarre MIDI, dove si utilizzano 6 canali, uno per ogni corda, o tastiere come il modello MK88 di Elka che consentiva all'utente di inviare i messaggi su 4 canali differenti in modo indipendente.

La tecnologia MPE, oltre alle possibilità elencate negli esempi appena citati, introduce il concetto di canale globale e definisce un nuovo standard a cui tutti i dispositivi MIDI polifonici possono aderire; tali argomenti verranno approfonditi nei paragrafi successivi.

2. La tecnologia MPE

È stato detto in precedenza che la tecnologia MPE si basa sul MIDI: può essere considerata come un superamento dello standard perché essa cerca di sfruttare il pieno potenziale di una tecnologia ormai obsoleta, e di creare quindi una sorta compromesso tra possibili nuovi protocolli con maggiori prestazioni e un mercato di prodotti standardizzati ormai dagli anni Ottanta.
In particolare, negli ultimi anni sono entrati in commercio nuovi dispositivi MIDI espressivi, che hanno l'obiettivo di permettere all'esecutore un controllo totale sulla singola nota, diversamente dai comuni controller che lavorano a livello di canale MIDI.
Tra le nuove piattaforme, di grande rilevanza sono Seaboard di ROLI e LinnStrument di Roger Linn. Questi sviluppatori hanno avuto la necessità di definire e caratterizzare un nuovo standard che si adatti alle innovazioni dei loro prodotti: nel dicembre del 2015 viene pubblicato il primo documento sulla tecnologia MPE, che spiega in poche pagine l'idea, le caratteristiche di base e le regole per sviluppare uno strumento di questo tipo.
Nello stesso periodo viene rilasciato l'aggiornamento alla versione 4 di JUCE, framework di casa ROLI, nel quale sono state inserite le classi che implementano la tecnologia MPE per lo sviluppo software di virtual instruments.

Attraverso l'MPE, viene definito un modo innovativo di utilizzare la gamma di canali, messi a disposizione dal MIDI, per riprodurre la musica polifonica, in modo da consentire il controllo di parametri, come modulazioni e Pitch Bend, di ogni canale in modo indipendente e permettere di rappresentare le informazioni relative a note espressive in tre o più dimensioni.
Viene a stabilirsi una intercomunicazione tra software e hardware MIDI espressivi sempre più ricca.
Di seguito sono elencati in breve alcuni aspetti significativi della tecnologia MPE:

- ad ogni nota suonata viene assegnato un canale MIDI, se possibile. Questo permette che i messaggi di controllo indirizzati su quel canale facciano riferimento solo e soltanto a quella particolare nota. Se ci sono più note attive che canali disponibili, alcune note dovranno condividere canale e messaggi di controllo con altre note: in tali circostanze, tutte le note continueranno comunque a suonare, ma non saranno più univocamente controllabili;
- vengono generati uno o più messaggi, inviati tramite un RPN (Registered Parameter Number), per stabilire un range di canali per l'invio o la ricezione di dati. Ognuno di questi messaggi supporta la suddivisione del canale MIDI in diversi sotto-spazi, dette zone di canale, in modo che venga utilizzata una sola interfaccia MIDI fisica per configurazioni multi - timbriche;

- un ulteriore canale dedicato è assegnato a ciascuna zona per trasmettere informazioni comuni come i dati del pedale Sustain, i messaggi di Program Change e Pitch Bend globale, da applicare su tutte la zona. Di default, questo canale è il Channel 1.

2.1 Canali e assegnamento delle voci

Lo standard MIDI, sin dalla sua prima versione, offre 16 canali indipendenti per trasportare separati flussi polifonici di note. Alcuni messaggi MIDI (come il Pitch Bend), si applicano ad ogni nota accesa sul canale in considerazione.
In MPE, ad ogni nuova nota viene assegnato un proprio canale MIDI; se tutti i canali sono già occupati, la nuova nota dovrà condividere un canale con una nota esistente. Entrambe le note suoneranno, ma saranno comunemente controllate con i messaggi su quel canale. Se i canali sono disponibili, la presenza di una nota attiva su un canale impedirà l'assegnazione di un'altra nota a quel canale, finché non verrà trasmesso sul canale in considerazione un messaggio di Note Off. I controlli di dati e Pitch Bend verranno applicati solo in caso di nota attiva sul canale.
Una voce, MPE Voice, rappresenta il mezzo con cui un sintetizzatore MPE riproduce un suono; siccome una voce può contenere un suono alla volta, un sintetizzatore deve prevedere un array di voci per garantire la polifonia. In altre parole, ogni volta che viene premuta una nota del sintetizzatore, quindi ad ogni Note On, viene creata una voce che contiene le caratteristiche timbriche del suono da generare.

2.2 Attivazione della modalità MPE

Si prevede che i messaggi di attivazione e di configurazione solitamente vengano trasmessi da una DAW o da un'applicazione di controllo per un controller o un sintetizzatore al fine di configurarlo per una situazione specifica. È onere dell'utente assicurarsi che la configurazione dei dispositivi sia corretta, al fine di evitare che i messaggi RPN ricevuti da un controller non siano automaticamente ritrasmessi a tutti i dispositivi collegati. I metodi di disattivazione, e il comportamento di uno strumento quando la modalità MPE è disattivata, sono lasciati allo sviluppatore.
MPE lavora in MIDI Mode 3 (Omni Off, Poly On), che è la configurazione standard della maggior parte dei sintetizzatori.

I dispositivi MPE generalmente verranno configurati con Channel 1 come Master Channel e i restanti canali disponibili per inviare e accogliere dati di note, in modo che, alla prima accensione, l'utente riesca ad avere una buona esperienza, anche se la configurazione delle interfacce in gioco non è propriamente corretta.

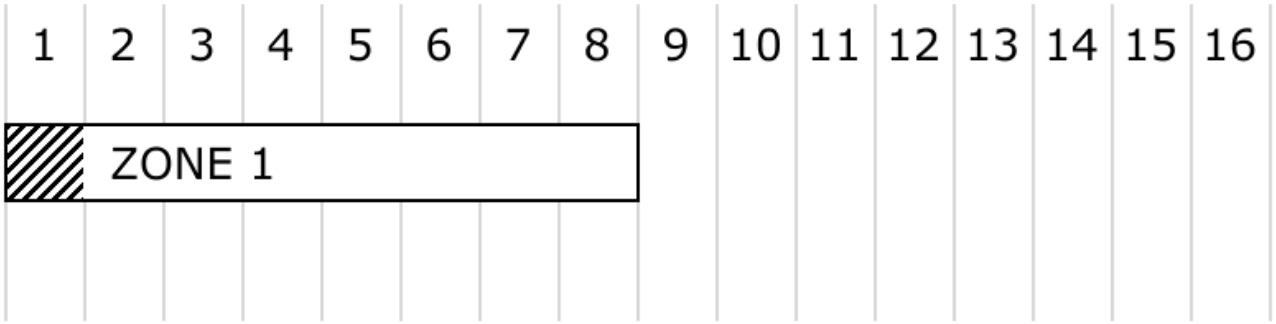
2.3 Setup delle Zone

La tecnologia MPE supporta la divisione dei canali MIDI in diversi sotto-spazi, che vengono chiamati zone. La definizione di una nuova zona parte dall'invio di un RPN 6 su ogni canale. Tale definizione deve essere in accordo con sei regole:

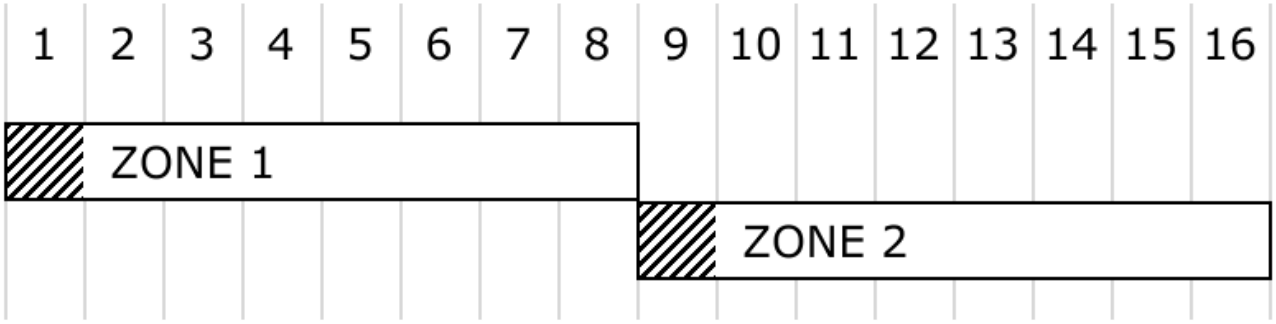
1. l'assegnazione del canale per la nota comincia dal canale su cui viene ricevuto l'RPN, e occupa un certo numero di canali specificati dal messaggio di Control Change;
2. il canale master delle zone, per la nuova zona, è il canale precedente rispetto al canale su cui è stato ricevuto l'RPN;
3. se parte di una nuova zona sovrascrive il canale master delle zone o il minore canale nota di una zona esistente, la vecchia zona viene eliminata del tutto;
4. altrimenti, se parte di una nuova zona sovrascrive parte di una zona esistente, la vecchia zona verrà accorciata;
5. definendo 16 canali nota, vengono cancellati tutti le correnti zone. Quando viene uno suonato uno strumento dove le zone sono cancellate, il comportamento predefinito del dispositivo è quello di lavorare in modalità singolo canale, quindi non in modalità MPE, e passare alla modalità Omni;
6. il protocollo non considera lo spazio canale. Il messaggio RPN non viene mai inviato o azionato sul Channel 1, perché in questo modo non si avrebbe un luogo dove mettere il canale master di zona. Per la stessa ragione, le zone non dovrebbero essere definite oltre il canale 16.

2.3.1 Esempio

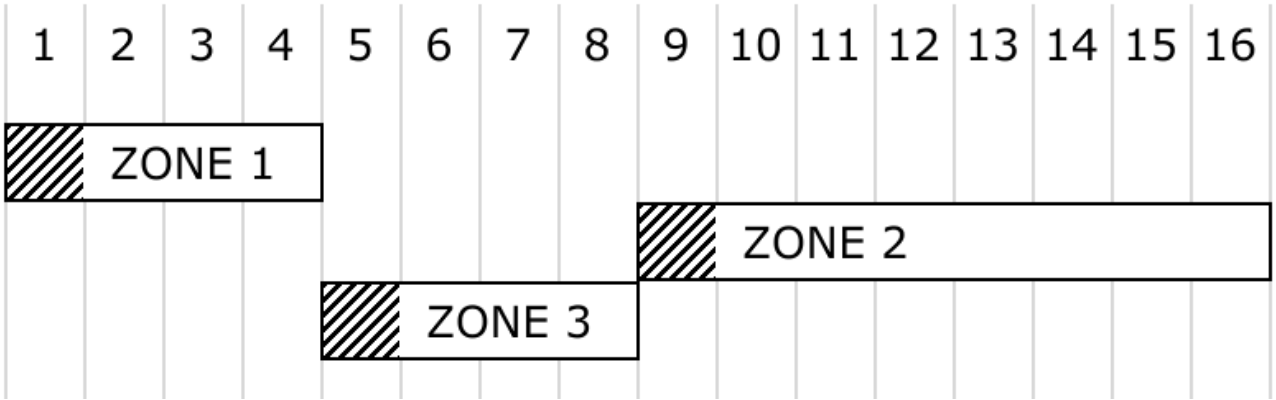
Si comincia con la creazione di una zona (Zone 1) di range di canali da 2 a 8. Il Channel 1, per la regola numero 2, sarà il canale master di zona.



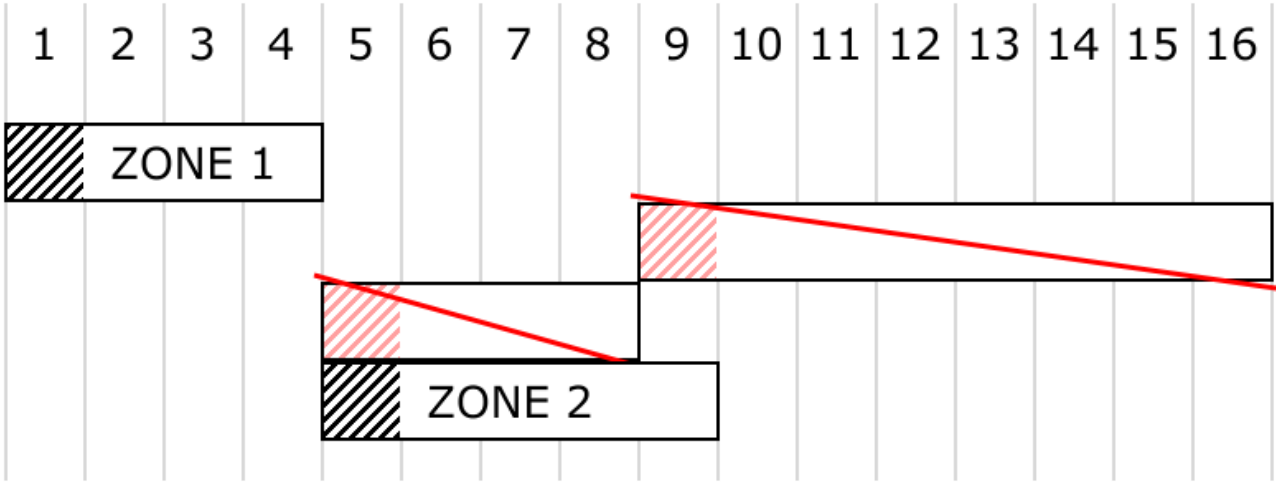
Allo stesso modo, viene creata una zona (Zone 2) nel range 10-16. Il Channel 9 sarà il canale master per questa zona.



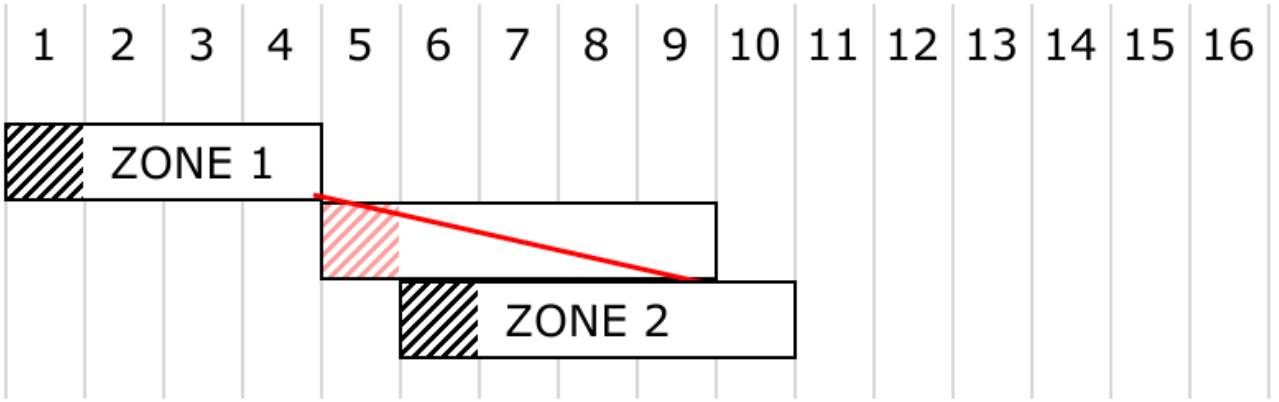
Ora viene creata una terza zona (Zone 3) nel range 6-8, che quindi sovrascrive parte della prima zona. Siccome in questi canali non si trova nessun canale master di zona, per la regola 4 la Zone 1 verrà accorciata per fare spazio alla nuova zona.



Se, a questo punto viene definita una zona tra i canali 6-9, con Channel 5 come canale master, si andrà a sovrascrivere due canali master (quelli delle zone 2 e 3). Per la terza regola, queste zone verranno eliminate, ottenendo una nuova Zone 2.



Un ultimo esempio evince come, sempre per la regola 3, se si elimina il primo canale nota di una zona, tutta la zona viene eliminata.



2.4 Note, Zone e System Scope

Il controllo real-time di uno strumento MPE può essere suddiviso in tre aree di azione:

- il Note scope, che fa riferimento al principale obiettivo della tecnologia MPE, ossia permettere il controllo per nota di Pitch, Pressure, o qualsiasi altra dimensione che lo strumento virtuale e il suonatore sono in grado di riprodurre quando suonano una nota;
- con lo Zone scope si riduce il traffico MIDI. È conveniente che alcuni messaggi vengano convogliati verso una zona specifica: Program Change, Sustain, Sostenuto e Volume vengono inviati solo sul canale master di zona. Alcuni controller danno la possibilità di applicare un'ulteriore Pitch Bend in tutto il sistema in una sola volta: per questo tipo di implementazione può essere trasmesso un messaggio di Pitch Bend utilizzando il canale master di zona;
- infine, per System scope vengono intesi tutti i messaggi di sistema MIDI che influenzano l'intera interfaccia MIDI.

Da queste definizioni si comprende meglio il ruolo del canale master di zona: esso permette di agire sulle zone ad esso connesse con messaggi di carattere globale, risparmiando così il numero di messaggi inviati al dispositivo. Nella tabella successiva vengono messe a confronto le tipologie di messaggio inviate alla nota con quelle inviate alla zona.

	Note scope	Zone scope
Channel pressure	✓	✓
Pitch bend	✓	✓
Modulation CC 1 [MSB] and CC 33 [LSB]	✓	✓
Pitch bend range message RPN 0	✓	✓
Other dimensions CC 70–79 [MSB] and CC 102–111 [LSB]	✓	×
All other controller messages, for example: CC 7 [volume]; CC 64 [sustain pedal]; CC 120 [all notes off]; RPN and NRPN messages	×	✓
Program change	×	✓

2.5 Channel Pressure e Aftertouch

La tecnologia MPE utilizza la Channel Pressure per trasmettere i dati di pressione con risoluzione a 7 bit. Tuttavia, a volte questa precisione potrebbe non essere sufficiente, perciò, se è necessaria una maggiore risoluzione (14 bit) potrebbero essere usati i CC 70 e 102; nello sviluppo di sintetizzatori si dovrebbe tenere in considerazione questa possibilità.

La precisione a 7 bit deve essere trattata come alternativa completamente separata da quella a 14 bit: non è consentito combinare i messaggi di Channel Pressure con un LSB per riempire i bit di ordine inferiore.

L'aftertouch polifonico (Key Aftertouch) è ridondante. Questo perché non è necessario che agisca su ogni singola nota del canale dal momento in cui ogni canale possiede una nota; basterebbe quindi un aftertouch monofonico (Channel Aftertouch). In ogni caso, potrebbe essere assegnato al canale master di zona, a discrezione del realizzatore, per conservare la compatibilità con modalità MIDI più datate.

2.6 Controllo della terza dimensione

I principali assi di controllo che si trovano nelle device MIDI convenzionali, ossia altezza e pressione, possono essere integrati da molti altri controlli. La terza dimensione di espressione, spesso usata per codificare il movimento perpendicolare agli assi di pitch e pressure, deve essere mappata come valore assoluto sui CC 74 e 106 come predefinita.

Il CC 74 è già assegnato dal protocollo MIDI come Sound Controller 5 (brightness); per convenzione, le implementazioni MPE chiameranno questo parametro Timbre.

3. Sviluppo di plugin audio in C++

In questa sezione vengono analizzati nel dettaglio i principali mezzi di base per lo sviluppo di plugin in C++.

Un plugin è un software non autonomo che viene caricato all'interno di un programma, chiamato host; esso interagisce con l'host come se fosse una parte del programma stesso, ampliandone o estendendone le funzionalità originarie.
La tecnica dei plugin, quindi, risulta un modo efficace per aggiungere dinamicamente funzioni ad un programma "base" e permette a sviluppatori terzi (estranei all'azienda produttrice dell'host) di aggiungere funzionalità ad un prodotto di un'altra azienda, senza nemmeno conoscere il codice sorgente del prodotto base.
Inoltre, tanto più saranno disponibili e funzionali i plugin scritti per un determinato programma, tanto più quest'ultimo potrà avere maggiore diffusione commerciale e quindi rallentare un'inesorabile situazione di obsolescenza digitale del software. Uno dei casi di maggiore successo è ben rappresentato dai software di produzione musicale di Steinberg, legato all'ampia diffusione dello standard VST per plugin musicali.

Il programma host ha il compito di ospitare l'esecuzione del plugin; per caricare rapidamente e in tempo reale una parte di software di un altro programma occorre servirsi di tecnologie comuni offerte dai sistemi operativi: DLL (Dynamic Link Library) su Windows, Components su Mac, SO (Shared Objects) su Linux.
Affinché l'host possa interagire con il plugin deve esistere un formato standard: in ambito musicale i più diffusi sono VST (Virtual Studio Technology di Steinberg), AU (Audio Units di Apple), AAX (Avid Audio eXtension di Avid per Pro Tools), ALSA (Advanced Linux Sound Architecture di Linux).
Il plugin deve esporre dei metodi predefiniti, ricevere ed inviare dati secondo lo standard: occorre quindi utilizzare le API (Application Programming Interface) fornite con gli appositi SDK (Software Development Kit). Tutti gli standard avranno metodi simili; in particolare, per quanto riguarda i plugin audio/MIDI si trova:

- un metodo per l'inizializzazione del processo audio, nel quale solitamente viene passata la sample rate (frequenza di campionamento, o di lettura dei campioni audio) e la buffer size (che rappresenta la dimensione del buffer di Input/Output dello streaming audio);

- un metodo per il processo audio e MIDI vero e proprio, che agisce direttamente sui buffer audio e buffer MIDI;
- un metodo per il settaggio dei parametri;
- un metodo per conoscere il valore di un parametro.

Inoltre vengono definiti alcune funzioni per comunicare con l'eventuale interfaccia grafica (GUI - Graphical User Interface):

- metodo per creare la GUI e associarla al processo audio;
- metodo che notifica alla GUI che ci sono stati dei cambiamenti ed è necessario un aggiornamento della grafica.

Non è detto che lo standard preveda anche le API per gestire la GUI, anzi spesso si usano SDK di terze parti.

Finora, per quanto stato detto, sembrerebbe che se si volesse sviluppare un plugin con interfaccia grafica per standard diversi, bisognerebbe conoscere le API di tutti i formati, scrivere il codice per ogni formato e scrivere diverse volte il codice per la GUI nel caso non sia supportata su tutti i sistemi operativi; questo problema viene risolto da framework che di fatto si occupano di includere le API corrette sia per il sistema operativo, sia per il formato desiderato e offrono dei propri metodi per la realizzazione di interfacce grafiche. Un esempio di framework multipiattaforma è sicuramente JUCE, di proprietà di ROLI, azienda molto attiva nel settore audio e musica digitale e tra i principali sviluppatori della tecnologia MPE.

3.1 Sviluppare plugin con JUCE 4

JUCE è un framework di sviluppo C++ multipiattaforma, orientato sia alla realizzazione di interfacce grafiche e soprattutto alla programmazione di applicazioni Audio/MIDI, anche standalone, non necessariamente plugin. Negli ultimi anni moltissimi produttori di software Audio/ MIDI hanno scelto JUCE per la programmazione dei loro prodotti in quanto il framework, alla creazione di un nuovo progetto, predispone l'ambiente di sviluppo correttamente a seconda delle indicazioni date dall'utente, e permette di sviluppare in un'unica interfaccia di lavoro per qualsiasi sistema operativo e per qualsiasi piattaforma, dall'ultima versione, anche per mobile.

In JUCE, nello sviluppo di un'applicazione audio, la classe cardine è AudioProcessor: rappresenta il processore audio e MIDI, perciò è necessario creare una classe che erediti da essa, affinché il software genere suono.

AudioProcessor presenta, oltre al costruttore, dove verranno create le istanze agli oggetti che servono, alcuni metodi principali che vanno sovrascritti:

- prepareToPlay (sampleRate, samplesPerBlock);
- processBlock (audioBuffer, midiMessage);
- setParameter (index, value), con value compreso tra 0 e 1;
- getParameter (index), con valore ritornato necessariamente tra 0 e 1;
- createEditor();

Si noti come le prime quattro funzioni rappresentano l'implementazione dei metodi audio/MIDI descritti nel paragrafo precedente.

Il metodo principale di AudioProcessor è sicuramente processBlock: è il metodo in cui avviene la generazione sonora e spetta all'utente sovrascrivere questo metodo affinché il buffer di uscita contenga i campioni che corrisponderanno alla forma d'onda desiderata. Importante è che i valori dei campioni siano necessariamente compresi tra -1.0 e 1.0.

Il metodo prepareToPlay viene chiamato all'inizio della riproduzione, dove il software inizializza i parametri necessari per la riproduzione sonora. La frequenza di campionamento passata rimane costante fino al termine della riproduzione, inoltre viene fornito il dato della grandezza dei blocchi di campioni.

I metodi setParameter e getParameter intervengono quando un valore del plugin cambia e deve essere nuovamente impostato, oppure quando si necessita di conoscere il valore di un determinato parametro.

La classe principale per creare l'interfaccia grafica è AudioProcessorEditor, quindi per sviluppare la GUI di un plugin occorre creare una classe che erediti da essa e sovrascrivere i suoi metodi, oltre ad altri metodi ereditati automaticamente da classi superiori (come Component o MouseListener).

È necessario il riferimento all'oggetto AudioProcessor in modo che la GUI sappia che è il proprio processore audio.

Un'interfaccia grafica contiene oggetti quali Slider, Button, Label, TextBox, tutti figli della classe Component. In più implementa il metodo paint() che permette di disegnare linee, forme e percorsi geometrici nell'interfaccia grafica.

L’interfaccia grafica del plugin, per essere una vera GUI e comunicare con il processore audio, dovrà avere altre classi “padre”. Esse sono:

- ChangeListener: per aggiornare la GUI se il processore audio informa che qualcosa è cambiato;
- SliderListener: per comunicare al processore audio che uno Slider è stato mosso, e quindi per passare al processore audio il nuovo valore affinché esso compia le azioni necessarie di aggiornamento dei parametri;
- ButtonListener: stessa idea di base della SliderListener, ma per i pulsanti.

D’altra parte il processore audio, per comunicare alla GUI i cambiamenti, deve ereditare da ChangeBroadcaster, il quale mette a disposizione il metodo sendChangeMessage(). È importante sottolineare che è importante che questo metodo venga chiamato solo e soltanto se qualcosa cambia: una continua chiamata, ad esempio a brevi intervalli regolari di tempo, ha come conseguenza che la GUI continua a ridisegnarsi e consumerebbe parecchia CPU.

3.2 Configurazione

Ovviamente, per poter compilare un plugin di un certo standard, serve l’SDK relativo: spesso questi kit di sviluppo vengono messi a disposizione gratuitamente dalle case proprietarie dello standard, quindi sono accessibili a tutti. Una situazione particolare riguarda il formato AAX, non semplice da ottenere in quanto AVID decide se lo sviluppatore è idoneo o no.

JUCE deve essere configurato correttamente: ad esempio bisogna specificare su quali piattaforme e quali formati compilare (non è necessario compilare tutti i formati), oppure includere le utility realmente necessarie. Deve essere anche dichiarato se il software è un plugin, oppure uno standalone, e parametri specifici quali il codice plugin, il tipo di plugin, la versione, ecc…
Projucer è un’applicazione fornita con JUCE 4 che aiuta ad effettuare tutte le configurazioni e crea i progetti per gli ambienti di sviluppo più comuni (VisualStudio, Xcode, CodeBlocks, …). È sempre buona norma effettuare la configurazione del progetto da Projucer in modo che automaticamente vengono aggiornate le configurazioni per tutti gli ambienti di sviluppo.

3.3 Esempio: generazione di rumore

Una volta configurato il nuovo progetto attraverso la procedura guidata di JUCE, vengono creati automaticamente 4 files, che saranno la base di partenza per lo sviluppo del software:

- PluginProcessor.h
- PluginProcessor.cpp
- PluginEditor.h
- PluginEditor.cpp

PluginProcessor.h e PluginProcessor.cpp implementano la sottoclasse di AudioProcessor, come precedentemente detto, la classe fondamentale di JUCE che definisce un processore sonoro, sia esso un Effetto o un virtual Instrument.
PluginEditor.h e PluginEditor.cpp implementano la sottoclasse di AudioProcessorEditor, che definisce l'interfaccia grafica del plugin.

Per generare un fruscio basterà semplicemente trovare nel file PluginEditor.cpp la funzione processBlock ed aggiungere al suo interno alcune righe di codice che producano una sequenza casuale di valori, che verranno percepiti dall'orecchio umano come rumore bianco.

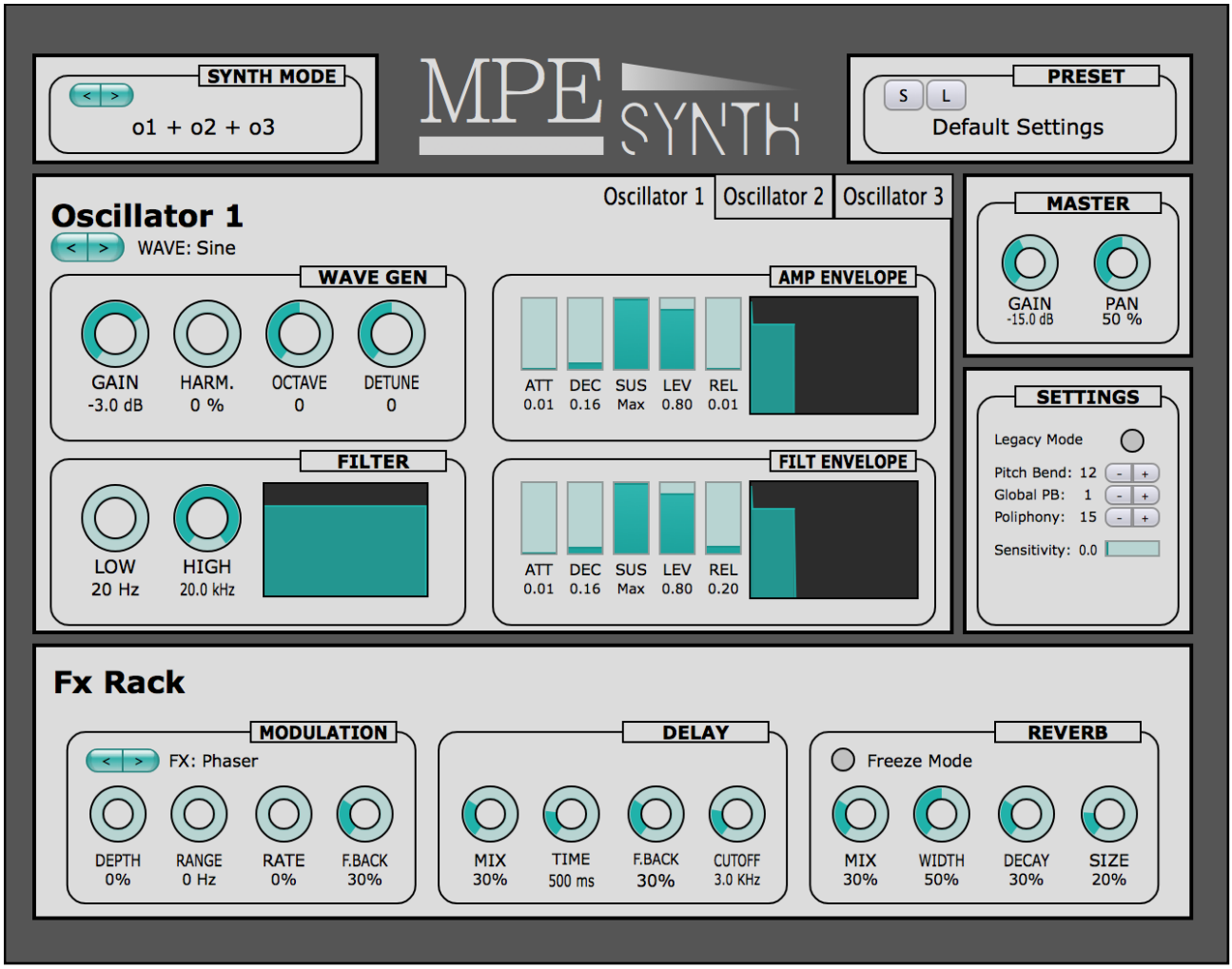
```
void pluginAudioProcessor::processBlock (AudioSampleBuffer& buffer, MidiBuffer&
midiMessages)
{
    for (int i = getNumInputChannels(); i < getNumOutputChannels(); ++i)
        buffer.clear (i, 0, buffer.getNumSamples());

    for (int channel = 0; channel < getNumOutputChannels(); ++channel)
    {
        float* channelData = buffer.getWritePointer (channel);

        for(int s = 0; s < buffer.getNumSamples(); s++){
            channelData[s] = (float)rand()/(float)RAND_MAX - 1.0f;
        }
    }
}
```

Da queste poche righe di codice è possibile notare un primo ciclo dove il buffer audio viene pulito, ossia tutti i valori vengono messi a 0. Il secondo ciclo invece riempie tutti i canali audio di output (channel = 0 è il canale sinistro, 1 è il destro, getNumOutputChannels() considera altri eventuali canali) con un buffer riempito per ogni suo campione da un numero casuale compreso tra -1.0 e 1.0.

4. MPE Synth

Viene ora affrontata in dettaglio il capitolo riguardante il caso di studio. Il plugin sviluppato consiste in uno strumento virtuale con tecnologia MPE. L'idea progettuale è di creare un sintetizzatore con 3 generatori di forme d'onda indipendenti, ciascuno in grado di controllare un filtro, un inviluppo di ampiezza e un inviluppo di filtro, uniti con una precisa modalità di sintesi. Il suono generato verrà poi trasmesso ad una catena di effetti per ottenere il timbro in uscita.

Il plugin è stato compilato secondo gli standard VST e Audio Units, a 64 bit, e si presenta al caricamento nell'host come visualizzato in figura.



4.1 Le Classi

Anzitutto, occorre precisare che il codice è stato suddiviso in diverse classi, in modo che la struttura risulti più ordinata e di facile comprensione. Scritte in C++ attraverso il framework JUCE, ogni classe rappresenta in genere un oggetto, di cui verranno create istanze nelle classi fondamentali. Per un'ulteriore organizzazione e facilità di lettura del codice, le classi predisposte all'interfaccia grafica sono state separate dalle rispettive classi di processing dell'audio.

Oltre alle classi fondamentali di JUCE, ossia AudioProccessor e AudioProcessorEditor, descritte nel paragrafo precedente, nello sviluppo di plugin MPE viene introdotta una nuova classe, che eredita da MPESynthesizerVoice: la classe, in questo caso specifico denominata Synth, contiene la generazione delle voci dello strumento, ossia include al suo interno tutte le istanze delle classi audio, ad eccezione dell'AudioProcessor e delle classi effetti. Di conseguenza, in essa viene generato gran parte di quello che sarà il timbro in uscita.

Nel progetto, le altre classi per la generazione del timbro sonoro:

- Oscillator - dove viene selezionata una forma d'onda, gestita da alcuni parametri;
- Filter - un semplice filtro passa banda, formato dall'accoppiamento di un passa alto ed un passa basso;
- Envelope - contiene l'inviluppo, utilizzabile sia come inviluppo di ampiezza, che come inviluppo del filtro;
- MpeReverb - un riverbero sviluppato da JUCE modificato per un miglior controllo dei parametri;
- MpeDelay - la classe per gli effetti di eco;
- MpeModultation - un contenitore di effetti di modulazione (phaser, chorus…).

Per meglio organizzare l'interfaccia grafica sono state create due ulteriori classi che organizzano la GUI:

- Container: contiene le classi che si occupano del timbro del suono, ossia un Oscillator, due Envelope e un Filter;
- FXRack: contiene le classi effetto.

Vi sono ulteriori classi di progetto, create per migliorare l'esperienza con il plugin:

- MpeManager - controlla tutti i parametri globali della classe MPE, e permette il passaggio da MPE a Legacy Mode, ossia la modalità standard comune a tutti i controller;
- SynthMode - agisce sulla modalità di accoppiamento delle forme d'onda;
- Preset - per salvare e caricare configurazioni stabilite;
- MasterGain - che controlla volume e pan generale.

Nei paragrafi successivi verranno descritti in modo più approfondito i metodi e i parametri delle classi che meritano maggiore attenzione.

4.1.1 La classe Oscillator

La funzione principale della classe Oscillator è consentire la definizione delle caratteristiche della forma d'onda. Nel plugin ci saranno tre istanze della classe Oscillator: questo significa che l'utente potrà definire fino a tre forme d'onda distinte tra loro, che verranno successivamente unite a seconda della modalità di sintesi selezionata.

Occorre sottolineare che questo modulo non produce suono, ma è parte fondamentale di quello che sarà il timbro del segnale di uscita dello strumento.

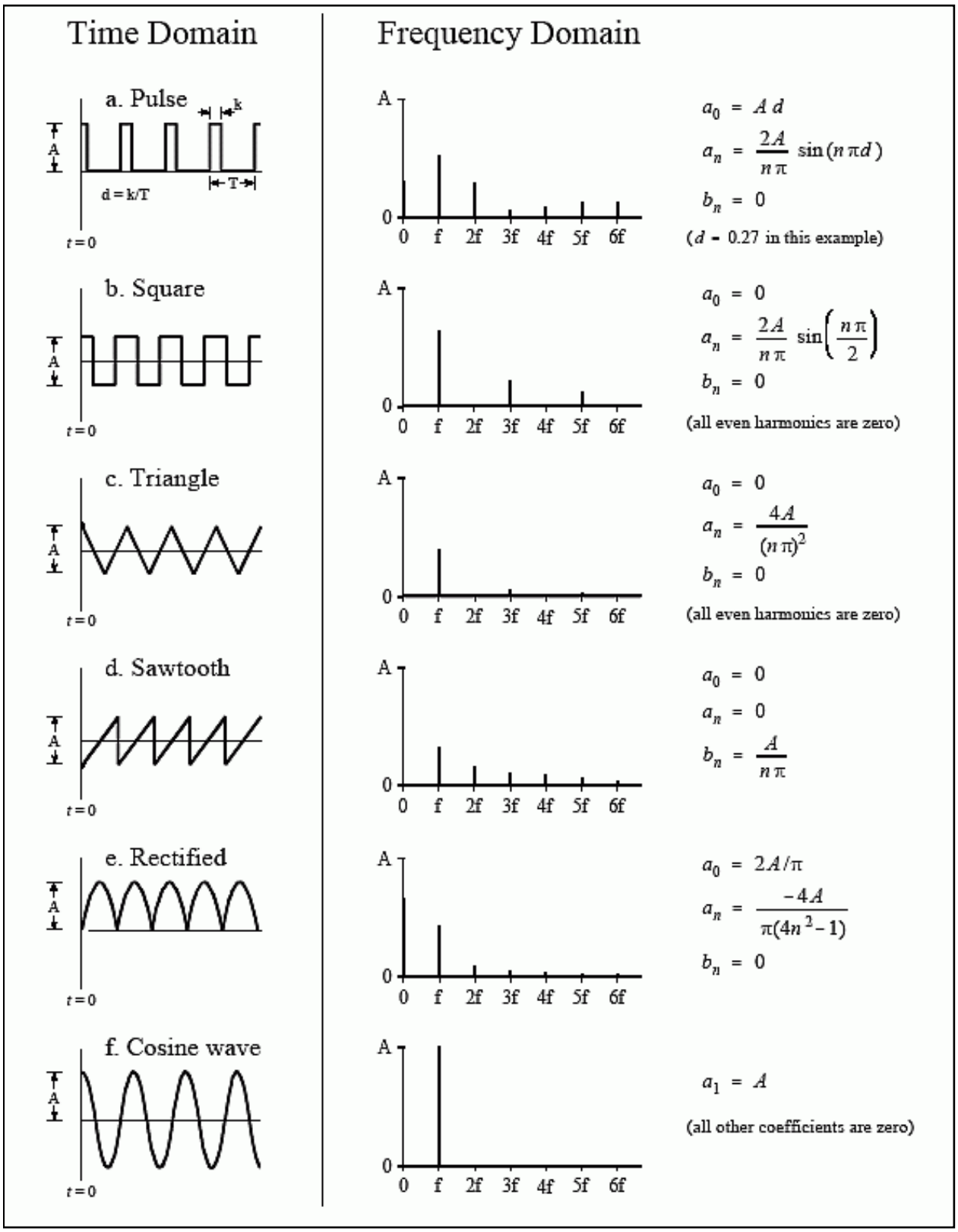
I valori dei parametri della classe non possono essere modificati in modo efficace durante l'esecuzione in tempo reale, a meno che l'Oscillator non venga supportato da un'adeguata interfaccia grafica. Il discorso delle interfacce grafiche vale per qualsiasi modulo che verrà trattato, e come detto in precedenza, si è preferito separare le GUI dalle rispettive classi audio. Perciò ogni discorso relativo alla grafica sarà rimandato nei prossimi paragrafi.

Le forme d'onda prodotte in questa classe possono contenere armoniche di frequenze molto alte, oltre il limite dell'udibile, che potrebbero però presentarsi come repliche nel segnale, dando origine al fastidioso fenomeno dell'aliasing; occorre quindi introdurre un filtro passa basso (a frequenza 18-20 kHz) per attenuare questo problema, senza che il timbro del sintetizzatore venga alterato.

Nella classe, in particolare, vengono definiti i seguenti parametri:

- **Wave**, ossia la forma d'onda selezionata. Possono essere scelte forme d'onda classiche, come la sinusoide, l'onda quadra, il triangolo e il dente di sega, forme d'onda più complesse o white noise. Tutte le equazioni delle forme d'onda (ad eccezione del white noise, che si tratta di una sequenza randomica di valori) vengono scritte per serie di Fourier, e inserite in delle Look-Up

Tables per essere lette. La tabella successiva mostra le principali forme d'onda nel tempo e nella frequenza, e come possono essere approssimate attraverso il teorema di Fourier.



Le Look-Up Tables altro non sono che strutture dati, usate per sostituire operazioni di calcolo che richiedono un buon uso della CPU con semplici operazioni di consultazione. In situazioni come

lo sviluppo di plugin, monitorare l'uso della CPU è un'operazione fondamentale: un software di questo tipo, che solitamente è utilizzato in real time, deve essere il più ottimizzato possibile per evitare sovraccarichi o altri inconvenienti che potrebbero compromettere la performance.

Lo script mostra le equazioni delle forme d'onda e come vengono inserite nelle Look-Up Tables.

```
int length = 200000.0;
float* sinArray = new float[length];
float* extraSinArray = new float[length];
float* sawArray = new float[length];
float* sqrArray = new float[length];
float* triArray = new float[length];
float* bassArray = new float[length];
float* plsArray = new float[length];

for(int s = 0; s<length; s++){
      float x = 2.0*M_PI*(s/88200.0);

      sinArray[s] = std::sin(x);

      extraSinArray[s] = std::sin(0.5*x) + std::sin(2.0*x) + std::sin(2.0*x) + std::sin(3.0*x);

      int nHarm = 12;
      float amplitude = 1.0f;
      float dutyCicle = 0.20f;

      for (int n = 1; n < nHarm; n++) {
        if (n%2 == 0) {
              sawArray[s] -= (amplitude/n)*std::sin(x*n); //armonica pari: sottraggo
        } else {
              sawArray[s] += (amplitude/n)*std::sin(x*n); //armonica dispari: sommo
              sqrArray[s] += (2.0f*amplitude/n)*std::sin(x*n);
              plsArray[s] += ((2.0f*amplitude)/n*M_PI)*std::sin(x*n*dutyCicle);
        }
      }

      triArray[s] = std::sin(x) - 0.11*std::sin(3*x) + 0.04*std::sin(5*x) - 0.02*std::sin(7*x)
+ 0.01*std::sin(9*x) - 0.008*std::sin(11*x);

      bassArray[s] = 0.25*std::sin(x) + std::sin(2*x) + 0.50*std::sin(3*x) + 0.1*std::sin(4*x)
+ 0.1*std::sin(5*x)+ 0.05*std::sin(6*x);

}
sinLookupTable = new SM_Wavetable(sinArray, length, 88200.0);
extrasinLookupTable = new SM_Wavetable(extraSinArray, length, 88200.0);
sawLookupTable = new SM_Wavetable(sawArray, length, 88200.0);
squareLookupTable = new SM_Wavetable(sqrArray, length, 88200.0);
triLookupTable  = new SM_Wavetable(triArray, length, 88200.0);
bassLookupTable = new SM_Wavetable(bassArray, length, 88200.0);
pulseLookupTable = new SM_Wavetable(plsArray, length, 88200.0);
```

- **Volume**, determina il contributo energetico dell'onda dell'oscillatore. Assegna il valore di volume tramite una funzione setGain, che sarà chiamata dal processore audio (PluginProcessor) ogni volta che verrà mosso lo slider relativo al volume dell'oscillatore nell'interfaccia grafica.

  Il valore di volume che viene passato alla funzione è in decibel, in un intervallo compreso tra -100 dB e 6.0 dB. Questo valore viene convertito in un valore lineare e assegnato alla funzione. Il parametro consente di mixare l'energia di ogni forma d'onda dei tre oscillatori, consentendo la percezione di un particolare timbro.

```
void Oscillator::setGain(float g) {
      gain.setValue(g);
}
```

- **Harmonics**, si tratta di una sorta di distorsione che aumenta il numero di armoniche nella forma d'onda stabilita. Partendo dalla forma d'onda fondamentale, ossia la sinusoide, per valori alti del parametro Harmonics, l'onda sinusoidale tenderà sempre più ad assomigliare ad un'onda quadra. Il valore di distorsione armonica è assegnato ad una variabile harmDist tramite una funzione setDistortion, mentre il metodo che realizza la distorsione armonica, in base al valore di amount (che nel nostro caso altri non è che harmDist) è la funzione distortion.

```
void setDistortion (float d) {
      harmDist.setValue(d);
}


float distortion( float input, float amount) {

      if (amount == 1.0f) {
        amount = 0.99f;
      }

       amount = std::sin((amount)*(double_Pi/2));

       float k;
       k = (2.0f * amount)/(1.0f - amount);
       float output = (1+k)*(input)/(1+k*std::abs(input));

      return output;
}
```

- **Octave**, modifica la frequenza della nota, trasponendola di una o due ottave, sia gravi che acute. La trasposizione agisce solo per l'oscillatore in considerazione. Attraverso la funzione setOctave viene stabilito il valore di trasposizione, ma è solo con setFrequency che la frequenza viene effettivamente trasposta. Questo perché ad ogni nota premuta viene chiamato setFrequency, di conseguenza per modificarne l'altezza occorre agire sul valore di frequenza passato a questa funzione.
- **Detune**, suddivide l'intervallo di semitono in 100 parti e modifica la frequenza di una nota. Trova utilità nelle operazioni di fine tuining o per creare una sorta di effetto Chorus con le forme d'onda degli altri oscillatori. La funzione fondamentale è setDetune, che non è altro che un'implementazione della teoria di Ellis riguardo al temperamento equabile, ossia che un'ottava di 12 semitoni può essere suddivisa in 1200 intervalli uguali. Anche in questo caso, la frequenza viene effettivamente modificata chiamando setFrequency.
- **Indice di Modulazione**, ossia un parametro ulteriore utile in alcune modalità di sintesi, in particolare la sintesi FM e la sintesi AM. In base quindi al valore che assume la modalità di sintesi (contenuto nella classe SynthMode), questo parametro viene attivato al posto del Detune, per quegli oscillatori la cui forma d'onda sarà una modulante. Il range di valori del parametro è impostato tra 0.0, che equivale a nessuna modulazione, e 10.0. Tale valore viene assegnato dalla funzione setModIndex e ripreso nella classe Synth durante la fase vera e propria di generazione

sonora. Vale la pena sottolineare che dal valore di questo parametro si evince che la modulazione produrrà suoni tanto più armonici quanto l'indice di modulazione si avvicina a valori interi.

```
void Oscillator::setOctave (int o) {
    octave = o;
    setFrequency(frequency);
}

void Oscillator::setDetune (float d) {
    float normDetune = d;
    detune = powf(2, normDetune/1200);
    setFrequency(frequency);
}

void Oscillator::setFrequency(float f) {
    frequency = detune*f*powf(2,octave);
    generator->setPitch(frequency);
    period = 1.0f/frequency;
    samplePeriod = period*sampleRate;
    sampleDutyCicle=samplePeriod*dutyCicle;
}


void Oscillator::setModIndex(float m) {
    modIndex = m;
}
```

4.1.2 La classe Filter

La classe Filter implementa un filtro passa-banda, ottenuto attraverso un filtro passa-basso e un filtro passa-alto. I parametri di questo semplice filtro sono:

- **Low Cut Frequency**, la frequenza di taglio del filtro passa-alto. Essa è variabile in un range di valori che va da 20 Hz a 5000 Hz. Viene settata dal metodo setHighPassFilter e ottenuta da getLowCutFreq.
- **High Cut Frequency**, la frequenza di taglio del filtro passa-basso. Come per la Low Cut Frequency, essa varia da 200 Hz a 20 kHz ed è configurata dal setLowPassFilter e richiamata da getHighCutFreq.

```
void setLowPassFilter(double freq){
     highCutFreq = freq;
}

void setHighPassFilter(double freq){
     lowCutFreq = freq;
}

double getLowCutFreq() {
     return lowCutFreq;
}

double getHighCutFreq() {
     return highCutFreq;
}
```

Una volta determinate le frequenze di taglio per entrambi i filtri, avviene il filtraggio vero e proprio, con la funzione bandPassFilter. Essa prende in ingresso il buffer audio e le due frequenze di taglio: il passa-banda è ottenuto tramite l'utilizzo in cascata del filtro passa-alto in successione al filtro passa-basso.

Il tipo di filtro realizzato è un IIR (Infinite Impulse Response). La caratteristica degli IIR è di essere sistemi dinamici causali la cui risposta all'impulso non è mai nulla al tendere all'infinito del tempo. Il modello matematico di questo tipo di filtro è:

$$y = \lambda \cdot y' + (1 - \lambda) \cdot x$$

dove:

- y è il segnale di uscita;
- y' è il segnale di uscita all'istante precedente;
- x è il segnale in ingresso;
- λ è un coefficiente dato da:

$$e^{-\frac{\pi Fc}{sr}}$$

Il metodo per il filtraggio è stato implementato in questo modo:

```
float bandPassFilter(double lowCutoff, double highCutoff, float sigIn ) {

      float outHPF = 0;
      float outLPF = 0;

      // Calcolo la lambda dei filtri = e^(2pigreco*Fc/Sr)
      float lambdaHP = exp(-double_Pi*lowCutoff/sampleRate);
      float lambdaLP = exp(-double_Pi*highCutoff/sampleRate);

      // Filtro Passabasso
      outLPF = lambdaLP*outLPF_1 + (1-lambdaLP)*sigIn;
      outLPF_1 = outLPF;

      // outputLPF è l'input del Filtro Passaalto
      outHPF = outLPF - inHPF_1 + lambdaHP*outHPF_1;
      inHPF_1 = outLPF;
      outHPF_1 = outHPF;

      return outHPF;
}
```

4.1.3 La classe Envelope

La classe Envelope implementa l'inviluppo attraverso un ADRS. L'inviluppo può essere utilizzato sia per modellare l'ampiezza del segnale in ingresso (amplitude envelope), sia per controllare il filtro (filter envelope) applicato al segnale da istanze della classe Filter.

Prima di entrare nel dettaglio sul funzionamento dell'inviluppo occorre spiegare in modo più approfondito i parametri e le variabili in gioco:

- **Attack**, il tempo che impiega un suono (o un filtro) ad andare dal valore 0 al valore massimo. Il tempo, in questo caso, altro non è che un numero di campioni, rappresentato dalla variabile attSamples: essa viene impostata dall'utente attraverso uno sldier nell'interfaccia grafica, contenuta nel Container, che chiama la funzione setAttack. Questo metodo, oltre che ad aggiornare il valore della variabile, determina anche il ratio (attRatio) del transitorio di attacco: esso rappresenta lo step di incremento del valore di volume (o di frequenza nel caso del filtro) ad ogni ciclo, e permette di arrivare al valore massimo nel tempo stabilito con una percezione sonora lineare.
- **Decay**, il tempo che impiega il suono (o il filtro) a passare dal valore raggiunto durante la fase di Attack al livello di sostegno. Similmente a quanto detto precedentemente per l'Attack, anche il Decay viene controllato dalle variabili decSamples e decRatio e impostato dalla funzione setDecay. A differenza dell'Attack, possiamo affermare che il ratio è negativo in quanto deve rappresentare una diminuzione di volume, o di frequenza nel caso si tratti di un filter envelope.
- **Sustain**, il tempo di sostegno, quando il volume (o la frequenza del filtro) si mantiene costante dopo le due fasi precedenti. Non ha senso definire una variabile susRatio in questo caso perché il livello non varia ad ogni campione ma rimane costante per tutta la durata del tempo di Sustain, perciò troviamo il metodo setSustain che si limita a settare i susSamples. Se questa variabile ha valore massimo (definito in una macro e rappresentato nell'interfaccia grafica dalla scritta "MAX") allora il tempo di sostegno sarà infinito, o meglio, termina nel momento in cui arriva un messaggio di Note Off. Se, invece, il valore di susSamples è minore del valore massimo, il tempo di sostegno sarà della durata indicata dall'interfaccia grafica, senza necessariamente attendere un Note Off: questo permette di simulare in modo più realistico tutti i tipi di strumenti, sia quelli percussivi, che hanno tempi di sostegno definiti e differenti, sia, ad esempio, organi o sintetizzatori, le cui note durano finché non viene rilasciato il tasto.
- **Level**, il livello di volume (o la frequenza) raggiunto al termine del tempo di Decay. Questo parametro è l'unico del modulo Envelope a non essere calcolato in campioni (o comunque in grandezze di tempo), ma assume un range di valori tra 0.0 e 1.0 e viene moltiplicato per il valore massimo del segnale, raggiunto al termine del tempo di attacco. La funzione che definisce il valore di Level, rappresentato dalla variabile susLevel, è setSusLevel. Questo metodo è strettamente connesso con il Decay: il decRatio varia a seconda del livello del segnale che deve raggiungere, perciò è necessario che ogni volta che cambia il valore di susLevel venga

aggiornato anche il ratio del Decay. Il ratio non modifica il tempo di decadimento, ma determina la quantità di segnale diminuita ad ogni campione.

Questo parametro è utile per simulare in modo ulteriormente convincente i diversi strumenti musicali: ad esempio, suoni percussivi hanno un valore di Level molto basso, mentre gli organi hanno un decadimento quasi nullo (susLevel = 1.0). Inoltre permette di creare inviluppi di filtro dinamici nel tempo.

- **Release**, il tempo di rilascio, ossia il tempo impiegato dal segnale a raggiungere il valore 0. Questo parametro interviene nel momento in cui arriva un messaggio di Note Off, oppure quando termina il tempo di Sustain. Come per maggior parte dei parametri, anche il Release è controllato da una variabile relSamples che determina i campioni necessari a raggiungere il valore 0, e da relRatio che anche in questo caso, indica la quantità di segnale ridotta ad ogni campione.

A fini di rappresentazione grafica dell'inviluppo (comunque esclusa da questo modulo), vengono utilizzate le funzioni setTotalEnvelope e getTotalEnvelope.

```
void setAttack(int a){
      attSamples = a;
      attRatio = 1.0/a;
      setTotalEnvelope();
}

void setDecay(int a){
      decSamples = a;
      decRatio = (1.0-susLevel)/a;
      setTotalEnvelope();
}

void setSusLevel(float a){
      susLevel = a;
      setDecay(decSamples);
}

void setSustain(int a) {
      susSamples = a;
      setTotalEnvelope();
}

void setRelease(int a){
      relSamples = a;
      setTotalEnvelope();
}

void setTotalEnvelope() {
      if (susSamples == MAX_ENV_SUS) {
        totEnvelope = INT_MAX;
      } else {
        totEnvelope = attSamples + decSamples + susSamples +relSamples;
      }
}

int getTotalEnvelope() {
      return totEnvelope;
}
```

Una volta definiti i metodi necessari per impostare i parametri dell'inviluppo, occorre chiamare la funzione envelope, che agisce direttamente sul buffer audio: essa, attraverso le variabili isNoteOn e envSamples riconosce in quale momento dell'inviluppo si trova il segnale in ingresso, ed effettua le operazioni necessarie. Il parametro env invece, applica il guadagno al segnale in relazione al valore di inviluppo calcolato.

La funzione setNoteOn, chiamata dalla classe Synth ad ogni NoteOn, imposta la variabile booleana isNoteOn, utile a rappresentare il momento di partenza del Release ogni volta che il Sustain è al massimo, e inizializza altri parametri utili al calcolo.

```
float envelope(float sigIn) {
      if (isNoteOn) {
        if (envSamples < attSamples) {
                // Attacco
                env += attRatio;
        } else if (envSamples< attSamples+decSamples) {
                // Decay
                env -= decRatio;
        } else {
                if (susSamples >= maxEnvSus) {
                } else {
                        if (envSamples >= susSamples || susSamples <= minEnvSus) {
                        setNoteOn(false);
                }
        }

      }
      envSamples++;
      } else {
        //Note off -> release
        if(envStatus){
                envSamples++;
                env-= relRatio;
                if (env <= 0){
                        env = 0;
                        if(envSamples > 0) envStatus = false;
                        envSamples = 0;
                }
        }
      }

      return env*sigIn;

}
```

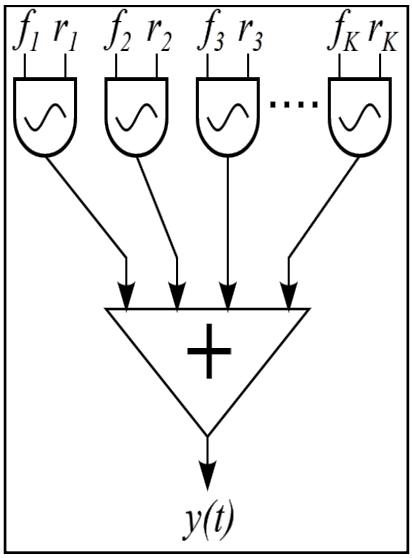
4.1.4 La classe SynthMode

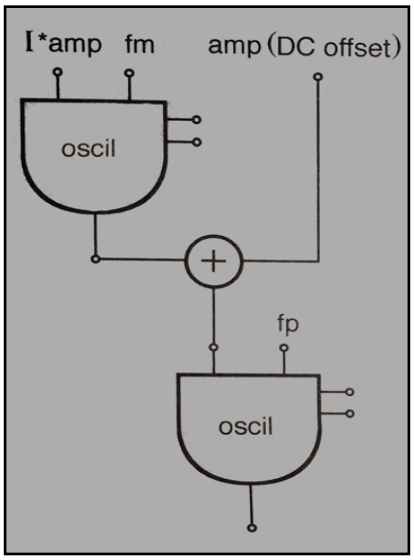
SynthMode è la classe adibita alla selezione della modalità di sintesi, ossia al modo in cui verranno sommati gli oscillatori. Come già detto in precedenza, nel plugin si avranno tre oscillatori con forme d'onda indipendenti, le quali verranno unite secondo quanto specificato dall'istanza della classe SynthMode.

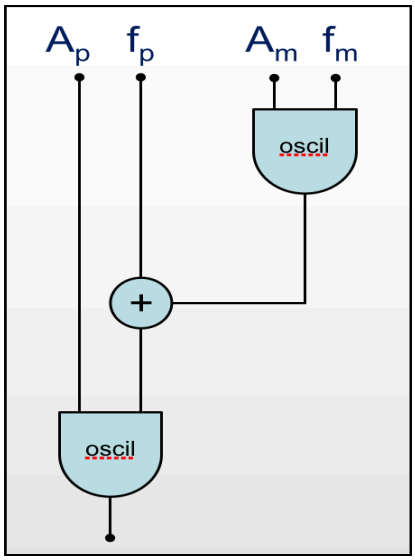
I metodi di sintesi implementati sono sostanzialmente tre: sintesi additiva, sintesi FM, sintesi AM. Data la presenza di tre oscillatori, è consentito l'utilizzo combinato di queste modalità: in particolare, troviamo:

- **Sintesi additiva**: tutte le forme d'onda dei tre oscillatori sono sommate tra loro: o1 + o2 + o3.
- **Sintesi FM 1**: la frequenza della forma d'onda del terzo oscillatore funge da modulante alla frequenza della sintesi additiva dei primi due oscillatori: (o1 + o2) FM o3.
- **Sintesi FM 2**: la frequenza della forma d'onda del terzo oscillatore funge da modulante alla frequenza della forma d'onda del secondo oscillatore, che a sua volta funge da modulante alla frequenza della forma d'onda del primo oscillatore. In altre parole, si tratta di una doppia modulazione in frequenza: (o1 FM o2) FM o3.
- **Sintesi AM 1**: l'ampiezza della forma d'onda del terzo oscillatore modula l'ampiezza del risultato della sintesi additiva tra i primi due oscillatori: (o1 + o2) AM o3.
- **Sintesi AM 2**: l'ampiezza della forma d'onda del terzo oscillatore modula l'ampiezza della forma d'onda del secondo oscillatore, che a sua volta modula l'ampiezza della forma d'onda del primo oscillatore. Si tratta di una doppia modulazione in ampiezza: (o1 AM o2) AM o3.
- **Sintesi AMFM**: la frequenza della forma d'onda del terzo oscillatore modula la frequenza della forma d'onda del secondo oscillatore, la cui forma d'onda modula l'ampiezza della forma d'onda del primo oscillatore: (o1 AM o2) FM o3.

A seconda della modalità di sintesi selezionata, nella classe Synth verrà richiamata una particolare funzione per ogni situazione; tali metodi verranno discussi successivamente nella spiegazione della classe Synth.







4.1.5 La classe MasterGain

La classe MasterGain agisce sul segnale in uscita del plugin, ossia dopo che è stato generato il timbro nella classe Synth. Si tratta di un semplice modulo che possiede due parametri:

- **Gain**, che determina il livello di uscita generale, è implementato attraverso la moltiplicazione del livello di volume del segnale di uscita.
- **Pan**, determina la posizione del segnale in uscita sul fronte stereo. La realizzazione di questo parametro avviene moltiplicando il segnale in uscita da ogni canale per un opportuno coefficiente, in modo tale da differenziare il livello di volume dei due canali. La somma dei coefficienti deve essere necessariamente 1.0, per mantenere costante l'intensità di uscita; quindi se il coefficiente a = 0.6 per il canale sinistro, per il canale destro si avrà un coefficiente b = 1.0 - a = 0.4.

4.1.6 La classe Synth

Il cuore della generazione sonora è la classe Synth; tutte le istanze dei moduli appena descritti vengono richiamati da questa classe e fatte interagire per creare il timbro finale in uscita dallo strumento.

Ogni istanza di Synth contiene una voce Mpe, di conseguenza, per garantire la polifonia, occorre che nell'AudioProcessor vengano create più voci. Questo fatto permette di comprendere più a fondo il discorso della tecnologia MPE: una voce, che nell'ambito percettivo corrisponde ad una nota, è un oggetto a sé stante, che non è vincolato ad altre voci, pertanto può essere controllato indipendentemente.

In questa classe vengono definite, in array di tre dimensioni per ottimizzare il codice, un'istanza della classe Oscillator, un'istanza della classe Filter e due istanze della classe Envelope, una per l'inviluppo di ampiezza e una per l'inviluppo del filtro. Inoltre troviamo un'istanza di SynthMode per la selezione della modalità di sintesi, e un'istanza di MasterGain, per il livello di uscita e il pan del segnale generale.

Sono definiti diversi metodi, che spesso richiamano le funzioni delle classi incluse. Tra questi metodi si può trovare:

- **noteStarded**: viene richiamata ogni volta che arriva un Note On. Come prima operazione, prende dall'istanza della classe synthMode il valore del parametro che rappresenta la modalità di

sintesi. Questo influisce sulla seconda operazione della funzione, che richiama il setFrequency della classe Oscillator per ognuno dei tre oscillatori: nel caso di sintesi FM, la frequenza dell'oscillatore verrà moltiplicata per l'indice di modulazione del relativo oscillatore (modIndex). Infine attiva gli ADSR di ogni oscillatore richiamando la funzione setNoteOn che abilita la funzione envelope.

- **noteStopped**: contrariamente a noteStarted, questa funzione entra in gioco ogni volta che arriva un messaggio di Note Off; l'unica operazione che svolge riguarda gli ADSR, disabilitando il setNoteOn e permettendo il processing del Release da parte della funzione envelope.
- **notePressureChanged**, **notePitchBendChanged**, **noteTimbreChanged**: sono le tre funzioni fondamentali che garantiscono l'espressività plugin, caratteristica fondamentale per uno strumento MPE. Esse vengono chiamate ogni qual volta che il valore del relativo CC cambia, ed aggiornano i parametri corrispondenti con i nuovi valori.

  In particolare, la Pressure viene controllata dall'asse Z e varia in base alla forza premente che agisce sul tasto; il Pitch Bend è rappresentato dall'asse X, ossia lo spostamento orizzontale del dito sulla superficie di controllo dopo che un tasto è stato premuto; infine il Timbre è la situazione analoga per l'asse Y, nel caso specifico di questo software assegnato al valore della frequenza di cutoff del filtro passa basso.
- **metodi di sintesi**: funzioni apposite per ognuna delle modalità di sintesi. A livello di codice sono piuttosto complicate, in quanto molti sono i parametri in gioco che devono essere assemblati al posto giusto. In questi metodi le istanze delle classi di processing audio vengono unite per formare il timbro della singola voce.
- **getNextSample**: richiama il metodo corretto in base alla modalità di sintesi stabilita, svolge operazioni di ottimizzazione della memoria (ad esempio cancella le note il cui inviluppo è terminato) e prepara il campione audio per essere scritto nel buffer di uscita.
- **renderNextBlock**: riempie i canali di uscita con i campioni ricavati dal metodo getNextSample, applicando il guadagno e il pan generale. Questo è il metodo della classe Synth richiamato dall'AudioProcessor nel processBlock.

4.1.7 Le classi Effetti

Le classi effetti coinvolte nel progetto sono una classe per le modulazioni (MpeModulation), una per l'eco (MpeDelay) e una per il riverbero (MpeReverb). Le istanze di questi oggetti non vengono

create nel Synth, come accadeva per le classi descritte finora, ma sono definite direttamente nell'AudioProcessor. Questo perché ogni effetto non deve essere applicato sul singolo campione, o sulla singola voce, ma deve agire su tutto il buffer audio in uscita, per ottenere un suono gradevole.

Nella classe Modulation viene implementato un Phaser. Il segnale in ingresso a questo tipo di effetto viene sdoppiato: uno rimane invariato rispetto all'originale, mentre l'altro viene fatto passare attraverso filtri All Pass (ossia senza attenuazioni di frequenze), che ad ogni passo sfasano il segnale di 180° in un determinato range di frequenze. Se si ascoltassero i due segnali in uscita singolarmente, non si percepirebbe alcuna differenza, in quanto l'orecchio umano non è sensibile alla fase. Tuttavia, se si sommassero i due segnali, risulterebbero delle interferenze lungo lo spettro sonoro, dovute appunto alle variazioni di fase apportate dai filtri.
La caratteristica del Phaser però è quella di generare un timbro dinamico: questo è possibile aggiungendo un oscillatore LFO che modifica nel tempo lo sfasamento apportato dai filtri, e che di conseguenza fa cambiare il range di frequenze che vengono enfatizzate.
Nel progetto, i parametri che entrano in gioco nel Phaser sono i seguenti:

- **Depth**, che stabilisce la percentuale di segnale effettata. In particolare, il segnale in ingresso al Phaser viene sdoppiato ulteriormente, consentendo di avere la possibilità di miscelare al segnale wet, la parte dry originale. Con una semplice moltiplicazione dettata dal parametro, si stabilisce la percentuale di presenza di un segnale piuttosto che di un altro.
- **Range**, indica l'intervallo di frequenze enfatizzate dal filtro All Pass.
- **Rate**, determina la velocità del cambiamento di fase.
- **Feedback**, ossia la quantità di effetto ricorsivo che agisce sul segnale.

L'algoritmo per processare il segnale con l'effetto di Phaser è stato realizzato in questo modo:

```
float playPhaser( float inSamp ){
      //Calcolo e aggiorno i parametri dell'LFO
      float d  = _dmin + (_dmax-_dmin) * ((sin( _lfoPhase ) +
                                          1.f)/2.f);
      _lfoPhase += _lfoInc;
      if( _lfoPhase >= F_PI * 2.f )
        _lfoPhase -= F_PI * 2.f;

      //Aggiorno i parametri del filtro
      for( int i=0; i<6; i++ )
        _alps[i].Delay( d );

      //Calcolo l'output
      float y =_alps[0].Update(
               _alps[1].Update(
               _alps[2].Update(
               _alps[3].Update(
               _alps[4].Update(
               _alps[5].Update( inSamp + _zm1 * _fb ))))));
      _zm1 = y;

      return inSamp * (1 - _depth) + y * _depth;
    }
```

_alps[i] rappresenta il banco di filtri All Pass, necessari al processing: sono istanze di una sottoclasse privata di Phaser.
Nella classe Modulation si potrebbero implementare altri effetti oltre al Phaser, per poter creare timbriche sempre più differenti. Generalmente, i classici effetti di modulazione sono il Chorus e il Flanger, che a livello di algoritmo sono molto simili al Phaser. L'implementazione di tali effetti potrebbe essere un passo di upgrade del plugin; questo potrebbe inoltre significare anche una revisione dei parametri in gioco, e di conseguenza una modifica dell'interfaccia grafica. Per come è stato strutturato il codice del software, al momento è possibile attivare sul segnale solo un effetto di modulazione.

La catena prosegue con gli effetti di ritardo, implementato nelle classi Delay e Riverbero. La differenza principale tra questi due tipi di effetti sonori riguarda la scala dei tempi resi disponibili: il riverbero riproduce il suono originale con un ritardo minimo, sicuramente di molto inferiore al secondo, mentre il delay può produrre suono anche a distanza di decine di secondi.

I parametri implementati nella classe MpeDelay che gestiscono il ritardo sono:

- **Mix**, che determina la percentuale di volume in uscita del segnale ritardato in relazione a quella del segnale in ingresso. Per mantenere un output di uscita costante, la percentuale di volume d'uscita del segnale non ritardato sommata a quella del segnale ritardato risulterà 100%.
- **Time**, il tempo di ritardo, indica dopo quanti millisecondi si avverte la prima ripetizione. Le successive ripetizioni inizieranno dopo intervalli regolari di tempo, determinate dal valore di questo parametro.
- **Feedback**, ossia il numero di ripetizioni. Trattandosi di un algoritmo ricorsivo, il segnale di uscita viene sdoppiato: uno dei due segnali viene sommato al segnale in ingresso, opportunamente pesato da un coefficiente di attenuazione, mentre l'altro costituita il vero output del sistema. In questo modo è possibile ripetere il segnale ritardato con un effetto di sfumatura nel tempo.
- **Cutoff**, per rendere più realistica l'attenuazione, al termine del circuito, prima di essere sdoppiato, il segnale attraversa un filtro passa basso. Questo parametro gestisce la frequenza di taglio del filtro.

Per quanto concerne il riverbero, i parametri gestiti dalla classe MpeReverb sono:

- **Mix**, parametro che svolge la stessa identica funzione del suo omonimo nella classe MpeDelay. Determina quindi la quantità di segnale wet rispetto alla quantità di segnale dry.
- **Width**, è un parametro che indica in percentuale la quantità di onde sonore riflesse. In altre parole, più questa percentuale è bassa, più il suono verrà assorbito dalle pareti dell'ipotetica stanza, viceversa ci saranno più riflessioni.
- **Decay**, indica la percentuale di decadimento delle onde riflesse. Il suono perde intensità all'aumentare della distanza percorsa e in base alla presenza di ostacoli lungo il percorso. Trattandosi nel caso specifico di suono riflesso, parte dell'energia verrà assorbita dalla parete riflettente. Questo parametro permette di simulare il tempo di decadimento delle onde riflesse, al fine di rendere più accurata l'esperienza sonora.
- **RoomSize**, la dimensione della stanza: più essa è grande, più percepibili saranno le riflessioni; il valore 0% simula un ambiente privo di riflessioni, quale potrebbe essere uno spazio aperto piuttosto che una camera anecoica. Questo parametro permette di verificare il comportamento dello strumento in diverse situazioni ambientali, oltre che a conferire una sorta di tridimensionalità al suono, dovuta alla percezione spaziale.
- **FreezeMode**, se attivo, "congela" lo status temporale del riverbero, che non sarà più soggetto ai fenomeni di decadimento.

```
void processStereo (float* const left, float* const right, const int numSamples) noexcept
    {
        jassert (left != nullptr && right != nullptr);

        for (int i = 0; i < numSamples; ++i)
        {
            const float input = (left[i] + right[i]) * gain;
            float outL = 0, outR = 0;

            const float damp    = damping.getNextValue();
            const float feedbck = feedback.getNextValue();

            for (int j = 0; j < numCombs; ++j)  // accumulate the comb filters in parallel
            {
                outL += comb[0][j].process (input, damp, feedbck);
                outR += comb[1][j].process (input, damp, feedbck);
            }

            for (int j = 0; j < numAllPasses; ++j)  // run the allpass filters in series
            {
                outL = allPass[0][j].process (outL);
                outR = allPass[1][j].process (outR);
            }

            const float dry  = dryGain.getNextValue();
            const float wet1 = wetGain1.getNextValue();
            const float wet2 = wetGain2.getNextValue();

            left[i]  = outL * wet1 + outR * wet2 + left[i]  * dry;
            right[i] = outR * wet1 + outL * wet2 + right[i] * dry;
        }
    }
```

Una considerazione importante va fatta in merito al posizionamento degli effetti lungo la catena: non esiste un modo corretto o un modo errato per posizionare i vari moduli, spesso è una scelta dettata dal gusto personale. Così come è stato fatto per questo software, la prassi comune suggerisce di inserire in catena prima gli effetti di modulazione, quindi quelli di ritardo; tuttavia la sperimentazione di nuove collocazioni degli effetti nella catena può contribuire alla produzione di suoni interessanti dalle timbriche non convenzionali. In questo senso, un aggiornamento del software potrebbe permettere all'utente di sistemare a suo piacimento gli effetti nella catena.

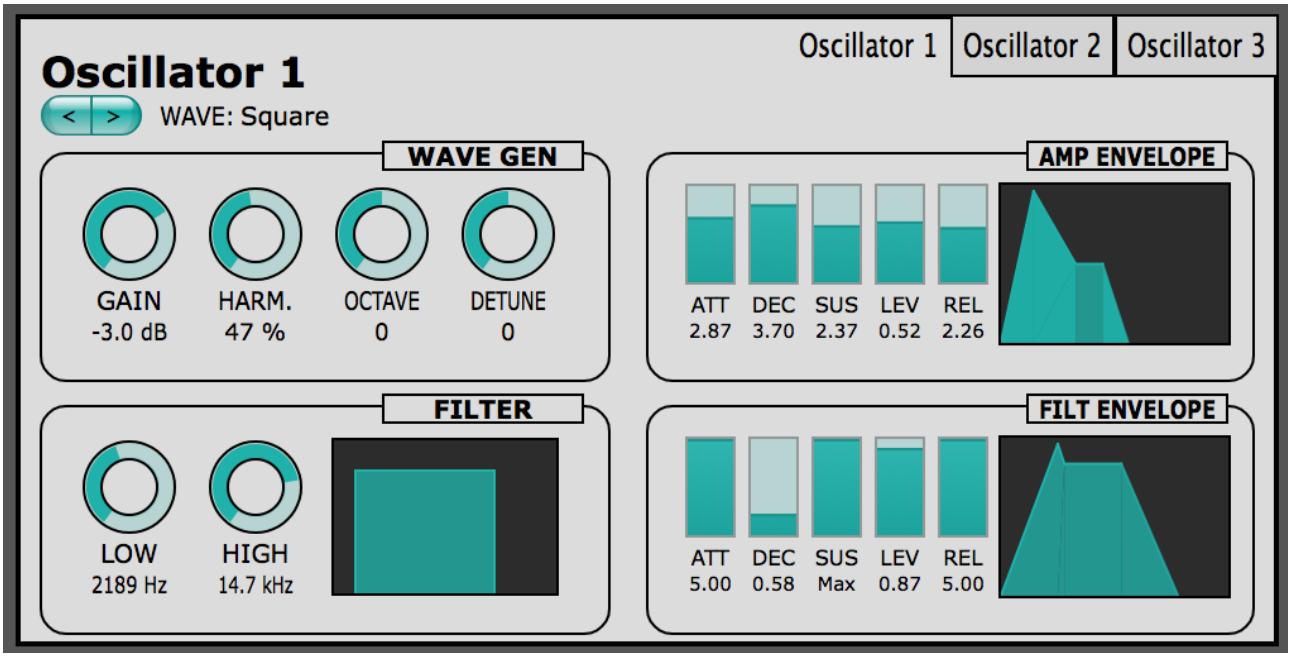
L'applicazione degli effetti avviene, come detto in precedenza, nell'AudioProcessor, all'interno del metodo fondamentale ProcessBlock.
Il segnale in uscita dal Synth passa attraverso le funzioni che richiamano gli effetti. Subito si può notare l'ordine nella catena dalla scrittura procedurale del codice, e come gli effetti agiscono sull'intero buffer e non sulla singola voce.

```
    // prelevo le uscite
    float* chOut[2];

    chOut[0] = buffer.getWritePointer(0);
    if( buffer.getNumChannels() == 2){
        chOut[1] = buffer.getWritePointer(1);
    }

    // Phaser
    if (effects->getEffect() == FxRack::kPhaser) {
        for (int s = 0; s < buffer.getNumSamples(); s++) {
            chOut[0][s] = phaser->playPhaser(chOut[0][s]);
            if( buffer.getNumChannels() == 2){
                chOut[1][s] = phaser->playPhaser(chOut[1][s]);
            }
        }
    }

    // Delay
    for (int s = 0; s < buffer.getNumSamples(); s++) {
        chOut[0][s] = delay->filter(chOut[0][s]);
        if( buffer.getNumChannels() == 2){
            chOut[1][s] = delay->filter(chOut[1][s]);
        }
    }


    // Riverbero
    if (effects->freezeButton->getButtonStatus() == 1) {
        reverb->setFreezeMode(0.99f);
    } else {
        reverb->setFreezeMode(0.0f);
    }


    if( buffer.getNumChannels() == 2){
        reverb->processStereo(chOut[0], chOut[1], buffer.getNumSamples());
    } else {
        reverb->processMono(chOut[0], buffer.getNumSamples());
    }
```

4.1.8 Interfaccia grafica

Le classi finora presentate agiscono solamente sui parametri audio, e non vi è modo per l'utente di modificarne i valori in tempo reale. Per questo motivo, occorre mettere a disposizione un'interfaccia grafica che agisca da tramite tra l'utente e i parametri del plugin.

Il framework JUCE fornisce librerie grafiche, contenenti sia elementi grafici geometrici di base, sia oggetti quali Sliders, Buttons, Labels, tutti editabili sovrascrivendo o ereditando i metodi LookAndFeel, per rendere l'interfaccia grafica originale.



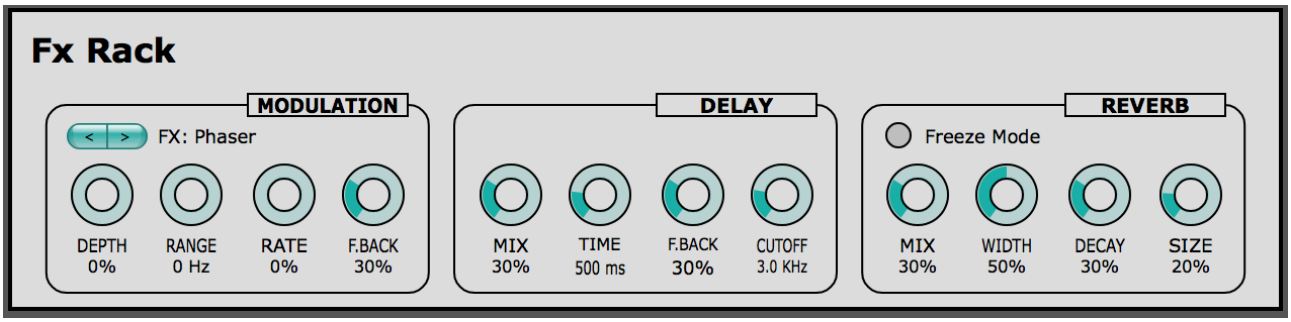
La classe Container contiene le interfacce grafiche dei moduli descritti in precedenza, ossia Oscillator, Filter ed Envelope: troviamo quindi un oscillatore, un filtro, un inviluppo di ampiezza e un inviluppo di filtro.

Dalla figura, si nota subito che tutti i parametri sopra descritti sono gestiti da Sliders di diversa natura (circolari, verticali,…). Ogni elemento grafico ha un suo Listener, che chiama una funzione relativa - sliderValueChanged per gli Sliders o buttonClicked per i Buttons - ogni volta che il valore dell'elemento viene cambiato. Attraverso queste funzioni è possibile notificare i cambiamenti di valore, non solo ad esempio alle Labels che mostrano tale valore, ma anche alle variabili vere e proprie che agiscono sull'audio.

Di carattere puramente grafico sono invece le semplici rappresentazioni grafiche di filtro e inviluppi: il rettangolo nero del grafico è stato suddiviso in modo equo un determinato numero di volte (ad esempio, quello dell'inviluppo prende il valore della funzione getTotalEnvelope), in modo

da ottenere un grafico proporzionato. Il grafico si aggiorna ogni volta che viene mosso uno Slider dei parametri rappresentati: è quindi logico aspettarsi nello sliderValueChanged una funzione repaint, tipica di JUCE per ridisegnare tutta o parte dell'interfaccia grafica. Il metodo repaint chiamato in sliderValueChanged risulta essere un modo intelligente di aggiornare la grafica, in quanto viene processato solo quando strettamente necessario; un modo più dispendioso in termini di memoria e con un risultato visivo non sempre ideale sarebbe aggiornare la grafica ad intervalli regolari di tempo.

La classe FxRack è la corrispondente interfaccia grafica per i moduli effetto, ossia MpeModulation, MpeDelay e MpeReverb. Riprende tutte le caratteristiche elencate nella classe Container.



4.1.9 La classe MpeManager

Nella maggior parte dei casi, i dispositivi MPE possono essere impostati via software per configurare parametri hardware. Ad esempio, il Pitch Bend eseguito nello spazio di un'ottava di una Seaboard può essere associato ad un incremento maggiore di 12 semitoni. Per questo motivo, e per evitare inconvenienti in fase di performance, occorre che sia software che hardware siano impostati allo stesso modo. Nel progetto realizzato non è possibile comunicare con l'hardware per modificare le impostazioni attraverso il plugin, viceversa è possibile configurare il software in modo identico alla device. La classe MpeManager si occupa appunto di impostare questi parametri. Essi sono Pitch Bend per canale, Pitch Bend Globale, polifonia delle voci.

Inoltre, sempre in questa classe, sono stati aggiunti un parametro di sensibilità per il Timbre e un pulsante per il passaggio alla Legacy Mode, ossia la modalità di fabbrica di un controller MIDI comune. In questa situazione il plugin perderà le caratteristiche espressive tipiche dell'architettura MPE, ma potrà essere suonato da qualsiasi tipo di piattaforma MIDI.

4.1.10 Salvataggio e caricamento

Per completezza del software, si è voluto permettere all'utente di salvare e caricare configurazioni personali dello strumento. La classe adibita al caricamento e al salvataggio è la classe Preset: essa si compone di due pulsanti, Save e Load, e una Label che mostra a video il nome del preset caricato. Il salvataggio dei parametri è supportato dalla creazione di un file XML, dove i parametri di ogni classe richiamata nell'AudioProcessor vengono salvati con il valore corrispondente. Nella funzione di salvataggio deve essere specificato anche il nome del parametro, in modo tale che il file XML risulti ordinato, leggibile e che non venga fatta confusione di indici in fase di caricamento.

```
XmlElement* MpeProject_j4AudioProcessor::saveParametersToXML(){
    XmlElement* xmlState = new XmlElement("mpesynth");

    xmlState->setAttribute("version","1");
    XmlElement* xmlProgram = new XmlElement("program");
    xmlProgram->setAttribute("value", "init");

    for(int k=0; k<kNumParameters;k++){
        XmlElement* xmlParam  = new XmlElement("param");
        xmlParam->setAttribute ("name", parametersNames[k]);
        xmlParam->setAttribute ("index", k);
        xmlParam->setAttribute ("value", getParameter(k));
        xmlProgram->addChildElement(xmlParam);
    }
    xmlState->addChildElement(xmlProgram);

    String folder =
File::getSpecialLocation(File::userApplicationDataDirectory).getFullPathName() + "/MPE Presets";
    File dataFolder = File(folder);

    if(!dataFolder.exists()) dataFolder.createDirectory();

    xmlState->writeToFile(File(dataFolder.getFullPathName() + "/Last Saved.xml"),"","",80);

    return xmlState;
}

void MpeProject_j4AudioProcessor::loadParametersFromXML(XmlElement* xmlState){

    String folder =
File::getSpecialLocation(File::userApplicationDataDirectory).getFullPathName() + "/MPE Presets";
    File dataFolder = File(folder);

    if(!dataFolder.exists()) dataFolder.createDirectory();

    xmlState->writeToFile(File(dataFolder.getFullPathName() + "/Last Read.xml"),"","",80);

    if(!xmlState->hasTagName("mpesynth")){
        return;
    }

    XmlElement* xmlProgram = xmlState->getChildByName("program");
    if(!xmlProgram){
        return;
    }

    XmlElement* xmlParam  = xmlProgram->getFirstChildElement();

    while(xmlParam != 0){
        if(xmlParam->hasTagName("param")){
            int k = xmlParam->getIntAttribute("index");
            float value = (float)xmlParam->getDoubleAttribute("value");
            setParameter(k, value);
        }
        xmlParam = xmlParam->getNextElement();
    }
}
```

Inoltre, il software dispone di un metodo di salvataggio e caricamento automatico richiamato ad ogni salvataggio o apertura di un progetto in una Digital Audio Workstation.

4.2 AudioProcessor

Come già spiegato nei capitoli precedenti, AudioProecssor è la classe cardine di JUCE che ha la funzione principale di mandare in output sulla scheda audio il buffer di uscita, al termine di tutte le operazioni necessarie alla creazione del timbro.

Il metodo fondamentale, ossia processBlock, per prima cosa riconosce la configurazione del sintetizzatore, ossia se è in MPE o in Legacy mode, quindi inizializza i parametri del processore audio con la funzione initSynth.

L'inizializzazione del sintetizzatore avviene ad ogni cambio di modalità: come si può evincere dal codice, la Legacy Mode non prevede l'utilizzo delle zone, ed è stata programmata perché venga disattivata ogni volta che viene creata una zona, al fine di evitare conflitti di modalità.

Dallo script nella pagina successiva, si nota che instrMpe è un istanza di MPEInstrument, che può essere vista come un'estensione della classe MPESynthesizer (synthMPE), con ulteriori funzioni tra cui la Legacy Mode. Il costruttore di MPESynthesizer permette di istanziare un MPEInstrument per mettere in relazione le due classi.

La modalità MPE viene configurata a partire dalla definizione di una zona: MPEZone è la classe creata a tale scopo e zonaMPE ne è un'istanza. Nel costruttore vengono impostati alcuni valori "(1, 15, 12, 2)" che rappresentano rispettivamente "(canale master di zona, polifonia, Pitch Bend per canale, Pitch Bend globale)", ripresi nella classe MpeManager. Per semplicità di utilizzo, viene solamente definita una zona con canale master di zona sul canale MIDI 1 e con canali nota dedicati alla zona sui canali MIDI 2 - 16.

A questo punto la zona creata viene aggiunta al gestore di zone layoutMPE che viene a sua volta assegnata al synthMPE, di modo che le voci lavorino correttamente sui canali stabiliti. Questa istanza tornerebbe molto utile quando vengono definiti più MPESynthesizer o MPEZone, che non è il caso preso in discussione da questo progetto.

Infine, per ogni canale voce definito nella zona, viene istanziata una MPESynthesizerVoice, cioè la voce contenente il timbro configurato nel Synth.

```
void MpeProject_j4AudioProcessor::initSynth() {

    legacyStatus = manager->legacyButton->getButtonStatus();

    if (instrMPE.legacyMode == false) {
        MPEZone zonaMPE(1, 15, 12, 2);
        layoutMPE.addZone(zonaMPE);
        synthMPE->setZoneLayout(layoutMPE);
    } else {
        instrMPE.enableLegacyMode();
        instrMPE.setTimbreTrackingMode(MPEInstrument::allNotesOnChannel);
        instrMPE.setPressureTrackingMode(MPEInstrument::allNotesOnChannel);
        instrMPE.setPitchbendTrackingMode(MPEInstrument::allNotesOnChannel);
    }

    synthMPE->setVoiceStealingEnabled (false);

    for (int i = 0; i < 15; ++i) {

        synthMPE->addVoice (voiceMPE = new Synth());

        if (instrMPE.legacyMode == false) {
            voiceMPE->setLegacyMode(false);
        } else {
            voiceMPE->setLegacyMode(true);
        }

        voiceMPE->setSensitivity(manager->filterSensitivity->getValue());

    }

    mpeLayoutChanged = true;

}
```

Dopo aver compiuto queste azioni di inizializzazione, il processBlock prepara lo streaming in uscita svuotando i canali audio non utilizzati, per evitare inconvenienti disturbi a livello sonoro, e riempiendo successivamente quelli adoperati recuperando i dati provenienti in uscita dal Synth.

```
for (int i = getNumInputChannels(); i < getNumOutputChannels(); ++i)
        buffer.clear (i, 0, buffer.getNumSamples());

    // qui avviene la “magia”: passando il blocco di messaggi MIDI, l’MPESynthesizer capisce, in
base al canale MIDI di ogni messaggio, a quali voci passare gli eventi relativi a quel canale, se
ci sono note-on fa partire il suono, se ci sono note-off fa smettere il suono, etc. Chiamando i
metodi che realizzati nella classe che eredita da MPESynthesiserVoice

    synthMPE->renderNextBlock (buffer, midiMessages, 0, buffer.getNumSamples());
```

Le operazioni successive sono inviare il segnale alla catena di effetti, come già precedentemente spiegato. Al termine della lettura procedurale del metodo processBlock viene prodotto il segnale audio vero e proprio, percepibile dalla scheda audio del computer.

L’AudioProcessor, nel caso di studio, non solo si occupa di produrre in uscita lo streaming audio, ma anche di collegare la GUI ai parametri audio reali rappresentati. Ad ogni classe grafica vengono associati i metodi setParamIndex e getParameter: l’AudioProcessor crea un proprio array di parametri globali, a cui mette in relazione i singoli parametri delle interfacce. È importante che tutti i parametri vengano definiti in modo univoco: se, come in questo caso, si ha la classe Oscillator con

5 parametri, e vengono create 3 istanze di questa classe, si dovranno avere nell'AudioProcessor 15 parametri con nome univoco, per evitare incomprensioni nel momento in cui vanno notificati i cambiamenti o vengono caricati determinati preset.

Il codice seguente mostra l'associazione tra parametri grafici di un'istanza (delle tre) della classe Container con l'array di parametri proprio dell'AudioProcessor:

```
panel[0] = new Container(ampEnvParams, filtEnvParams);
panel[0]->setProcessor(this);

panel[0]->setParamIndex(Container::kWave, kWave1);
panel[0]->setParamIndex(Container::kGain, kGain1);
panel[0]->setParamIndex(Container::kDistortion, kDistortion1);
panel[0]->setParamIndex(Container::kOctave, kOctave1);
panel[0]->setParamIndex(Container::kDetune, kDetune1);
panel[0]->setParamIndex(Container::kModIndex, kModIndex1);

panel[0]->setParamIndex(Container::kAttackA, kAttAmp1);
panel[0]->setParamIndex(Container::kDecayA, kDecAmp1);
panel[0]->setParamIndex(Container::kSustainA, kSusAmp1);
panel[0]->setParamIndex(Container::kReleaseA, kRelAmp1);
panel[0]->setParamIndex(Container::kSusLevelA, kLevAmp1);

panel[0]->setParamIndex(Container::kAttackF, kAttFilter1);
panel[0]->setParamIndex(Container::kDecayF, kDecFilter1);
panel[0]->setParamIndex(Container::kSustainF, kSusFilter1);
panel[0]->setParamIndex(Container::kReleaseF, kRelFilter1);
panel[0]->setParamIndex(Container::kSusLevelF, kLevFilter1);

panel[0]->setParamIndex(Container::kLowFreq, kLowFreq1);
panel[0]->setParamIndex(Container::kHighFreq, kHighFreq1);

for (int i=0; i< panel[0]->getNumParameters(); i++)
      addParameter(panel[0]->getParameter(i));
```

Attraverso setParameter, viene settato il nuovo valore che assume il parametro, riconosciuto in base all'indice globale di AudioProcessor, e notificato il tutto alla relativa interfaccia grafica. Nell'esempio, viene riportato il settaggio di una nuova forma d'onda in un oscillatore:

```
void MpeProject_j4AudioProcessor::setParameter (int parameterIndex, float newValue){

      switch (parameterIndex) {

        case kWave1:
        case kWave2:
        case kWave3:
        {
                int waveValue = roundFloatToInt(panel[parameterIndex-kWave1]
                        ->paramArray[Container::kWave]->range.convertFrom0to1(newValue));
                panel[parameterIndex-kWave1]->OscilSlider[0]->setValue(waveValue);
                for (int v=0; v < synthMPE->getNumVoices(); v++) {
                        ((Synth*)synthMPE->getVoice(v))->oscillators[parameterIndex-kWave1]
                                ->setWave(waveValue);
                }
         }
        break;

      }
}
```

4.2.1 AudioProcessorEditor

Si conclude il discorso relativo al software con una descrizione di AudioProcessorEditor. Si è parlato di questa classe come il luogo principale dove costruire l'interfaccia grafica del plugin, tuttavia nel progetto sono state definite ulteriori classi di interfacce grafiche (su tutte Container e FxRack). Di conseguenza, basta che le loro istanze siano contenute in AudioProcessorEditor per essere visualizzate. AudioProcessorEditor quindi si occuperà solamente di posizionare nello spazio definito i vari moduli di interfacce grafiche, di gestire il colore di sfondo e di assegnare le dimensioni in pixel del plugin.

5. Conclusioni e sviluppi futuri

Il software realizzato già presenta buone prestazioni e stabilità, e rispetta le funzionalità base di un plugin con specifiche MPE, su tutte la possibilità di controllo totale dei parametri di una nota e di agire sulle tre dimensioni. L'obiettivo del progetto sicuramente è stato raggiunto: JUCE e Xcode sono state ottime piattaforme per la realizzazione, sia per la preparazione dell'ambiente dell'ambiente di sviluppo, configurato correttamente in automatico, sia per l'organizzazione e la gestione delle classi, consentendo da subito la possibilità di compilare in modo funzionante il software.

L'ambito di applicazione del software certamente risulta essere quello di produzione musicale e, attraverso il salvataggio e caricamento di preset, e quindi la possibilità di recuperare un timbro creato dall'utente in breve tempo, di utilizzo in contesti live.

Attualmente, il plugin è disponibile come VST e come Audio Unit ed è stato testato su Mac OS X 10.11 e Mac OS X 10.12, con risultati positivi sulle seguenti DAW: Logic Pro X, Vienna Ensemble Pro, REAPER e Garageband. Non vi sono configurazioni che mandano in crash questi software, ma non è detto che in altri host non si presenti qualche bug: ogni workstation, infatti, ha un proprio specifico ordine di chiamata delle funzioni di interazione tra host e plugin, quindi si potrebbero riscontrare situazioni in cui un vengono generati errori di caricamento. Un lavoro più accurato e valido consisterebbe nel testare il software realizzato almeno sulle DAW più diffuse, come Cubase o Digital Performer, per avere un discreto grado di certezza che funzioni correttamente per la maggior parte di un ipotetico insieme di utenti. Inoltre, il file di progetto di JUCE è stato configurato anche per Windows, in particolare sulle ultime versioni dell'ambiente di sviluppo Visual Studio, in modo che possa essere facilmente compilato per ottenere una dll apposita per il sistema operativo di Microsoft.

Tuttavia, il plugin non può definirsi completamente ultimato, in particolare per due motivi.

- **Aggiunta di funzionalità**: il plugin è stato pensato per essere in continua evoluzione; come già accennato in alcuni paragrafi precedenti, molte potrebbero essere in futuro le implementazioni di classi, piuttosto che particolari metodi di processing. Basti pensare alla quantità innumerevole di classi effetto che possono essere integrate: allo stato attuale è presente solo un Phaser, ma nulla vieta di aggiungere un Chorus, un Flanger, un Harmonizer, una Distorsione, e così via. Questa potrebbe essere una prima implementazione, dato che il plugin già predispone uno Slider pensato per il cambio di effetto; un po' più complesso risulterebbe l'aggiunta di un numero di effetti a discrezione dell'utente, o del posizionamento di un effetto in un punto particolare della catena, in

quanto tale operazione comporterebbe anche un adeguamento dell'interfaccia grafica.

Altra semplice operazione consisterebbe nell'aggiunta di LookUp Tables di forme d'onda per la generazione del suono: la serie di Fourier permette di approssimare qualsiasi segnale periodico, per cui è possibile generare forme d'onda personali con sonorità particolari. Ad esempio, nel plugin si trovano le forme d'onda ExtraSine e Instr. Bass, che rispettivamente rappresentano una sinusoide modificata e un timbro simile a un basso elettrico. Inoltre, pesando opportunamente i coefficienti delle armoniche delle forme d'onda classiche è possibile differenziare il contributo energetico a diverse frequenze, per cui con una stessa serie di Fourier vengono generati timbri differenti, ad esempio una Sawtooth 1 con molta presenza di armoniche basse, e quindi con un suono più pieno, una Sawtooth 2 con grande contributo alle alte, ottenendo un suono più acido, una Sawtooth 3 senza alcune armoniche, ecc… Questo aspetto riguarda perlopiù l'ambito del sound design, ossia della programmazione timbrica di uno strumento musicale.

Ultima semplice aggiunta riguarda i parametri delle varie classi: al momento sono stati modellati un massimo di cinque parametri nei moduli Oscillator, Envelope e MpeReverb. Ovviamente nella generazione sonora molte più variabili entrano in gioco, di conseguenza tutte sono parametrizzabili e controllabili. Rifacendosi al discorso precedente, potrebbe essere interessante, nella classe Oscillator, aggiungere un numero di slider che controllano il contributo energetico di ogni armonica di una forma d'onda, consentendo all'utente di generare la propria forma d'onda senza che venga per forza scritta in una tabella in fase di pre-compilazione e che non venga più modificata. L'aggiunta di parametri diventa un'operazione più delicata al momento dell'inserimento degli stessi in una GUI, perché se da un lato è vero che più controlli permettono una gestione maggiore del timbro, dall'altro possono creare confusione soprattutto agli utenti meno esperti, che non saprebbero più come gestire decine di Slider. Inoltre, occorre non dimenticare di aggiornare i vari array di parametri indicizzati perché la GUI interagisca con la giusta variabile audio e perché non si creino inesattezze nelle operazioni di salvataggio e caricamento.

Un'operazione più complicata invece potrebbe riguardare l'assegnamento dei parametri a MIDI CC di default, o tramite un'interfaccia di gestione, in modo da permettere la manipolazione real time attraverso i controller.

- **Ottimizzazione del codice**: spesso a questo passaggio non viene data troppa importanza, in qualsiasi ambito di applicazione. Invece è fondamentale che il software esegua le operazioni richieste con il minore impiego di risorse, e che non occupi troppo spazio in memoria; trattandosi poi, in questo caso specifico, un plugin utilizzabile per suonare in real time, l'ottimizzazione

risulta essere vitale. L'ottimizzazione del codice non è un'operazione semplice da portare a termine: trovare una soluzione che migliori le prestazioni non è un'azione immediata o comunque sempre fattibile. Esempi possono riguardare le operazioni algebriche nelle equazioni: divisioni, elevamenti a potenza, logaritmi richiedono un maggiore dispendio di CPU rispetto alle moltiplicazioni o alle somme, per cui è consigliabile evitare questi calcoli laddove possibile, soprattutto se ripetuti spesso; una variabile o delle macro in questi casi risultano molto utili.

Inoltre è importante che tutti i puntatori agli oggetti creati vengano distrutti se non sono più adoperati, in quanto alleggeriscono il carico di memoria utilizzata. Inizialmente sembrerebbero operazioni banali e inutili, ma quando il software inizia ad avere una discreta dimensione, un codice ottimizzato farà la differenza in termini di prestazioni.

Nel caso in esame, si è cercato di sviluppare il plugin cercando di ottimizzare il più possibile, tuttavia in futuro, anche in base all'aggiunta di nuove funzionalità, alcuni metodi o alcuni calcoli potrebbero necessitare di essere riscritti.

In conclusione, si può affermare che lo scopo principale del progetto è stato raggiunto con risultati piuttosto soddisfacenti, tenendo comunque in considerazione le possibilità di sviluppo sopra elencate affinché il software possa essere effettivamente utilizzato anche livello professionale.

6. Bibliografia e sitografia

Libri e manuali:

- C. Adam, G. Lengeling, M. Sapp, C. Johanson, E. Eagan, L. Haken, K. McMillen, R. Jones, G. Bevin, A. Gaynes, R. Linn, R. Lamb, B. Supper, J.B: Thiebaut: “Multidimensional Polyphonic Expression”, The MIDI Manufactures Association, Dec. 2015.
- M. Malcangi: "Informatica Applicata al Suono per la Comunicazione Musicale – Musical Digital Audio: Teoria e pratica", Maggioli Editore, Milano, 2008.
- R. Bianchini, A. Cipriani: “Il suono virtuale”, Edizioni ConTempoNet, 2001.
- J. Proakis, D. Manolakis: “Digital signal processing – Principles, algorithms, and applications”, Patience Hall International Editions, 1996.
- A Cipriani, M. Giri: “Musica Elettronica e Sound Design – Teoria e pratica con Max e MSP Volume 1”, Edizioni ConTempoNet, 2009.
- A Cipriani, M. Giri: “Musica Elettronica e Sound Design – Teoria e pratica con Max Volume 2”, Edizioni ConTempoNet, 2013.
- A. Frova: “Fisica nella musica”, Zanichelli, 1999.
- R. Spagnolo: “Manuale di acustica applicata”, CittàStudiEdizioni, 2008.
- M. Robinson: “Getting started with JUCE”, Packt Publishing, 2013.

Siti Web:

- dal sito web di JUCE: https://www.juce.com/
  - https://www.juce.com/doc/classAudioProcessor
  - https://www.juce.com/doc/classMPEInstrument
  - https://www.juce.com/doc/classMPESynthesiser
  - https://www.juce.com/doc/classMPESynthesiserVoice
  - https://www.juce.com/doc/classLabel
  - https://www.juce.com/doc/classButton
  - https://www.juce.com/doc/classSlider
  - https://www.juce.com/doc/classLookAndFeel
  - https://www.juce.com/doc/classReverb
- dal sito web di Roger Linn Design: http://www.rogerlinndesign.com/
  - http://www.rogerlinndesign.com/mpe.html

- da MusicDSp: http://musicdsp.org/
  - http://musicdsp.org/files/phaser.cpp
- dal sito web di Suono Elettronico: http://www.suonoelettronico.com/
  - http://www.suonoelettronico.com/introduzione.htm
  - http://www.suonoelettronico.com/cap_iii_2_oscillat.htm
  - http://www.suonoelettronico.com/cap_iii_2_2_rumore.htm
  - http://www.suonoelettronico.com/cap_iii_2_3_inviluppo.htm
  - http://www.suonoelettronico.com/cap_iii_3_filtri.htm
- dal sito web MIDI Association: https://www.midi.org/
  - https://www.midi.org/specifications/item/the-midi-1-0-specification
  - https://www.midi.org/specifications/category/reference-tables
- da DSPGuide: http://www.dspguide.com/
  - http://www.dspguide.com/ch13/4.htm
- da http://www.ludovico.net/:
  - http://www.ludovico.net/students_soundsynth.php
  - http://www.ludovico.net/students_midi.php