UNIVERSITÀ DEGLI STUDI DI MILANO

DIPARTIMENTO DI INFORMATICA

Corso di Laurea in

Informatica Musicale

Scripting KSP per la gestione e modifica di pattern percussivi MIDI in tempo reale

RELATORE

Prof. Goffredo Haus

CORRELATORE

Prof. Simone Coen

TESI DI LAUREA DI

Raffaella Migliaccio

Matr. 795286

Anno Accademico 2013/2014

Ringrazio la mia famiglia, gli affetti più cari e le OFFicine Musicali Amaronesi
senza il cui aiuto non sarei potuta diventare la persona che sono.

Ringrazio il prof. Simone Coen
per il supporto e la disponibiltà che ha dimostrato nel periodo di stage.

Ringrazio Milano
per avermi fatto conoscere molti amici e musicisti
e avermi fatto riflettere sui pregi e difetti di vivere in un piccolo paese,
quale quello in cui io sono cresciuta.

Indice

1 Stato dell'arte ed obiettivi della tesi **1**
1.1 Chocolate Audio 1
1.2 Lo stato dell'arte 2
1.3 Obiettivi del lavoro 5

2 Tecnologie e strumenti software **7**
2.1 Musical Instrument Digital Interface 7
2.2 Kontakt Script Processor 9
2.3 Strumenti di supporto 11

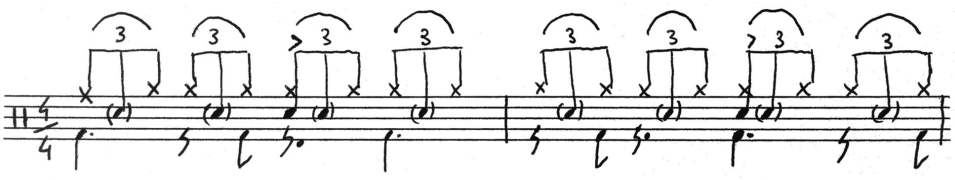
3 Progetto e sviluppo dell'applicazione **15**
3.1 Produzione dei pattern 15
3.2 Strutture dati per la rappresentazione dei pattern 19
3.3 Elaborazione dei pattern 20
3.4 Riproduzione dei pattern 27
3.5 Esportazione dei dati 30

4 Conclusioni e sviluppi futuri **33**
4.1 Il risultato conseguito 33
4.2 Nuove funzionalità 34

Bibliografia **37**

CAPITOLO 1

Stato dell'arte ed obiettivi della tesi



In questo capitolo, dopo aver descritto brevemente l'azienda presso cui si è svolto lo stage, presentiamo alcuni prodotti che costituiscono lo stato dell'arte del settore e hanno fornito lo spunto iniziale per il nostro lavoro. La sezione finale di questo capitolo descrive quali sono stati gli obiettivi che ci siamo preposti.

1.1 Chocolate Audio

Figura 1.1: il logo di Chocolate Audio.

Chocolate Audio [1] è un'azienda nata nel 2003 e si occupa di sound design e produzione di librerie di campioni. Guidata da Simone Coen, ha lavorato prevalentemente nell'ambito dei servizi business to business, ovvero il cosiddetto "conto terzi", collaborando con aziende del settore quali: *Native Instruments*, *Ableton*, *Steinberg*, *Cakewalk*, *Scarbee*, *Arobas* e altre.

Pur avendo come impegno principale il lavoro per conto di altre aziende, dal 2006 Chocolate Audio ha inaugurato una sua linea prodotti, prevalentemente basata su Native Instruments (di seguito NI) *Kontakt*, che vende direttamente al pubblico tramite la propria presenza online.

Chocolate Audio si è occupata prevalentemente di strumenti acustici o elettroacustici ed è particolarmente apprezzata per le sue produzioni in ambito di suoni percussivi e batteristici in particolare.

Nell'ottica di lavorare sempre di più nell'affermazione del marchio presso il pubblico di produttori, DJ e musicisti, Chocolate Audio sta sviluppando una serie di "motori" per strumenti basati su NI Kontakt, in particolare facenti capo ai suoni di natura percussiva.

1.2 Lo stato dell'arte

Prima di iniziare la progettazione e lo sviluppo del prodotto, abbiamo svolto una indagine su quello che la concorrenza offre rispetto alle funzionalità che ci eravamo proposti di implementare, partendo dal presupposto che volevamo progettare del software per il prodotto NI Kontakt.

Ecco tre tra i migliori prodotti attualmente sul mercato.

1.2.1 Apple Logic Pro X Drummer

Drummer di Apple è inserito all'interno di *Logic Pro X* [7] (e di *GarageBand* [3]) e in quanto tale non ha la struttura di un plugin, ma è parte integrantedell'applicazione.

La gestione dei pattern è molto interattiva, come risulta dalla Figura 1.2.

Figura 1.2: Logic Pro X, di Apple.

Tramite tale interfaccia si può selezionare:

- il genere musicale;
- il batterista, possibilità che costituisce una sorta di personificazione della tecnologia in ambito musicale;
- la "song" intesa come macro-struttura di riferimento;

- il pattern, secondo una logica bidimensionale e continua (rappresentata dall'elemento di interfaccia dato dal punto giallo nel riquadro nero) agendo su un asse di complessità (da "simple" a "complex") e su uno di intensità (da "soft" a "loud");
- si possono accendere e spegnere selettivamente i singoli pezzi di batteria, o forzare delle variazioni;
- il checkbox "follow" offre la possibilità di imporre a Drummer di "seguire" il ritmo di un'altra traccia (basso, drum, loop, etc.) per quanto riguarda le figurazioni di kick e snare, questa caratteristica è resa possibile dal fatto che Drummer è integrato in Logic.

1.2.2 Toontrack EZDrummer

EZDrummer [2] è un prodotto che è sul mercato da diversi anni ed è stato aggiornato alla versione 2 da diversi mesi. Si tratta di un "instrument plugin" dedicato; in quanto tale non ha di per sé tutte le limitazioni imposte da Kontakt e dal motore di scripting.

La sezione per la gestione dei Pattern MIDI è strutturata in modo simile a un sequencer MIDI di una *Digital Audio Wordkstation* (DAW). Dal browser della libreria di pattern (parte in alto nella Figura 1.3), si possono trascinare nell'apposita zona i diversi pattern e costruire così la propria canzone.

Figura 1.3: EZDrummer 2, di Toontrack.

Esistono due funzioni interessanti in EZDrummer relative alla gestione pattern:

- la possibilità di suonare un ritmo su un pulsante “tap” e fare in modo che il motore di ricerca restituisca i Pattern che più somigliano a ciò che abbiamo suonato,
- la presenza di un “song creator” ovvero di un algoritmo che dati i pattern caricati, li concatena in modo appropriato rispetto agli stilemi classici della forma canzone.

Rispetto agli obiettivi che ci siamo posti per questo elaborato, EZDrummer si presenta comunque in modo molto rigido in quanto possiede una collezione non molto coerente di Pattern e soprattutto la procedura di costruzione della composizione è molto strutturata e non consente interazioni vere e proprie in tempo reale, obiettivo che invece abbiamo ritenuto importante per il nostro lavoro in quanto, oltre a consentire maggiore espressività al musicista, costituisce un importante lato “giocoso” del fare musica.

1.2.3 NI Studio Drummer

Ci soffermeremo di più su *Studio Drummer* [10] che è una libreria per NI Kontakt, lo stesso prodotto per il quale abbiamo scelto di sviluppare la nostra soluzione.

Native Instruments [8] è leader mondiale di hardware e software per la produzione di musica basata sull’uso del computer. L’azienda si pone l’obiettivo di immettere nel mercato prodotti innovativi e completi in grado di soddisfare ogni tipo di esigenza.

Kontakt (nella versione 5) [6] è uno dei più importanti campionatori e player attualmente in circolazione. Dispone di una libreria con più di mille strumenti e circa 43 GB di campioni. Inoltre, tale prodotto mette a disposizione un motore di scripting, denominato *Kontakt Script Processor* (KSP), che consente di eseguire script in un linguaggio proprietario (deniminato *Kontakt Scripting Language*), i quali permettono di personalizzare in modo versatile ed evoluto il comportamento degli strumenti riprodotti da Kontakt.

La libreria esistente che più si avvicina all’idea che abbiamo voluto implentare è *Studio Drummer*. Questo consiste in una sorta di batterista virtuale che, entro certi limiti, può aiutare il produttore/compositore nella fase di pre-produzione o produzione di brani musicali.

Di tale libreria, qui descriviamo solo la sezione che più ci interessa: quella della gestione dei ritmi di batteria.

Come si vede nella figura 1.4, *Studio Drummer* ha molti ritmi, circa tremila, divisi per genere (funk, rock, jazz, pop, ecc.). Dispone di due funzioni principali per la manipolazione degli stessi:

Quantizzazione permette di aggiustare le “precisione” del ritmo secondo una griglia di figure ritmiche (ottavi, sedicesimi, trentaduesimi).

Swing permettere di aggiungere dello swing al ritmo, dando un "gusto jazz" anche ad un ritmo rock.

Figura 1.4: la sezione di gestione dei ritmi di *Studio Drummer*

1.3 Obiettivi del lavoro

L'idea che ci ha mosso a sviluppare la soluzione proposta in questa tesi è stata la volontà di scrivere un software di supporto a chi si occupa di musica, occupandoci nello specifico di ritmi e dello strumento che più li rappresenta: la batteria.

Osservando *Studio Drummer*, abbiamo pensato di creare uno strumento simile, ma migliorarlo con quello che secondo noi mancava e cioè una classificazione più accurata delle varie parti ritmiche che costituiscono una canzone, dando quindi la possibilità all'utente di disporle come preferisce, pur seguendo una logica musicale.

1.3.1 Classificazione delle parti ritmiche di una canzone

Definiamo nel seguito con precisione i nomi che useremo nella tesi per i **tipi** di parti ritmiche di una canzone:

Intro Parte iniziale della canzone.

Groove Definito genericamente *ritmo principale*, il termine groove indica il ritmo predominante della canzone nelle varie parti che la costituiscono (strofe, ritornelli, etc).

Fill È un passaggio che viene eseguito, ad esempio, tra le varie porzioni di groove, oppure tra un'intro e un groove. Serve per evidenziare una cambiamento nella canzone, o più semplicemenete per dare una sfumatura alla composizione.

Outro Parte conclusiva della canzone.

In seguito utilizzeremo **pattern** per indicare una parte ritmica che appartiene a uno di questi tipi. Inoltre, definiremo con il termine **song** una collezione di pattern che risultino coerenti dal punto di vista musicale e possano essere abbinati in una composizione musicale.

Osserviamo che, all'interno di una composizione musicale, è possibile avere più groove e fill che si alternano, mentre avremo generalmente un solo intro e un solo outro.

1.3.2 La nostra soluzione

Nonostante nella nostra analisi delle soluzioni esistenti abbiamo incontrato ottimi prodotti, molti di essi sono risultati limitati sia che si avvantagino di essere embedded in una *DAW* (Digital Audio Workstation, software usato nella produzione musicale), sia che funzionino indipendentemente da essa.

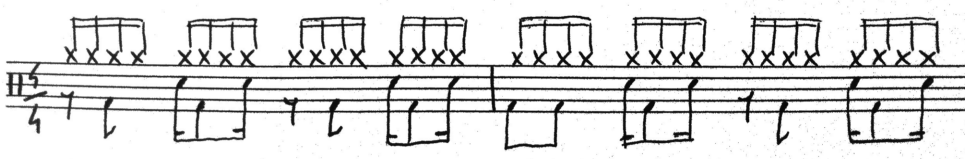
Sebbene *Studio Drummer* sia tra i rappresentanti dello stato dell'arte esso consente solo di scegliere tra sequenze pre-assemblate di intro, groove, fill e outro, e le stesse non possono assolutamente essere organizzate in un flusso di riproduzione.

Dato che Chocolate Audio ha una grande esperienza nella produzione di materiale audio abbiamo deciso di approfondire la conoscenza di KSP per sviluppare una soluzione innovativa che permettesse di superare i limiti dei prodotti presenti sul mercato.

Il punto di forza del nostro software risiede nella possibilità di offrire all'utente di personalizzare la propria composizione ritmica scegliendo sia i pattern da utilizzare che come assemblarli.

CAPITOLO 2

Tecnologie e strumenti software



Questo capitolo richiama le tecnologie principali e gli strumenti software più importanti che abbiamo adoperato per conseguire gli obiettivi che ci siamo dati. In particolare: nella prima Sezione esponiamo gli elementi dello standard MIDI che saranno utili a comprendere la trattazione dei dati presentata nel seguito, nella seconda Sezione descriviamo NI Kontakt ed il suo linguaggio di programmazione, mettendone in luce alcuni limiti; in fine, nell'ultima Sezione descriviamo alcuni strumenti di supporto allo sviluppo e le motivazioni che ci hanno indotti a sceglierli.

2.1 Musical Instrument Digital Interface

2.1.1 Lo standard

Il Musical Instrument Digital Interface (MIDI) standard, nato negli anni '80, comprende sia una specificia hardware, che un protocollo di comunicazione per i dispositivi musicali elettronici, come tastiere, computer ecc.. I dati MIDI possono essere sia trasmessi durante le performance, che essere memorizzati per l'editing o playback successivo.

In questo paragarafo vogliamo, pur senza pretese di completezza, esporre alcune caratteristiche base del MIDI [15] che saranno utili nell'esposizione del nostro lavoro.

I dati MIDI viaggiano da un dispositivo all'altro, sotto forma di **messaggi MIDI**. I differenti tipi di messaggi MIDI sono distinti dal primo byte del messaggio stesso noto come **byte di stato**, il quale definisce la tipologia di messaggio, seguito dai cosiddetti **byte di dati** che contengono

i dati veri e propri. I messaggi MIDI più comuni sono composti dal byte di stato più due byte di dati e sono detti **short message** e sono utilizzati per trasmettere informazioni sulle note da riprodurre; lo standard prevede messaggi più lunghi, adoperati usualmente per veicolare informazioni accessorie come il metro della misura, il nome della composizione, etc..

2.1.2 Short message

Descriviamo qui gli short message MIDI che sono pertinenti al nostro lavoro, specificando per brevità solo il significato dei byte di dati:

NOTE_ON

- primo byte: **pitch** (altezza) della nota da suonare,
- secondo byte: **velocity** (intensità) della nota da suonare.

NOTE_OFF

- primo byte: pitch della nota da silenziare,
- secondo byte: velocity della nota da silenziare, usualmente 0.

Osserviamo che un NOTE_ON con velocity pari a 0 è interpretato come un NOTE_OFF.

CONTROL_CHANGE

- primo byte: **numero** del controllo,
- secondo byte: **valore** del controllo.

I messaggi di CONTROL_CHANGE servono, tipicamente, a trasmettere informazioni di controllo, come nel nostro caso, il grado di apertura o chiusura del *charleston*.

2.1.3 Tick, risoluzione e BPM

Una caratteristica centrale del protoccollo MIDI è che i dati sono trasmessi tra i dispositivi in tempo reale, in un "flusso". Essi non contengono di per sé informazioni di temporizzazione, ogni messaggio viene elaborato al suo arrivo, dunque si presume esso giunga nel momento giusto.

I messaggi MIDI possono essere memorizzati da un sequencer (usualmente è un software in esecuzione su un computer) che li immagazzina come **eventi MIDI**, ossia messaggi MIDI con in aggiunta un'informazione di temporizzazione, che specifica *quando* è stato originato il messaggio, e per cui viene utilizzata una unità di misura denominata **tick**.

Osserviamo che il tick non è una misura assoluta di tempo (espressa in frazioni di secondo), ma una misura relativa alla **risoluzione** che è appunto il numero di tick contenuti in un quarto. Per calcolare la durata t in secondi di un tick, è necessario pertanto conoscere i *Beat Per Minute* (**BPM**), ossia il numero di quarti per minuto. Se la risoluzione vale R, allora:

$$t = \frac{60}{R \times BPM}\ s$$

2.2 Kontakt Script Processor

Una delle caratteristiche che rende NI Kontakt molto interessante per il Sound Design moderno è che il suo comportamento è in parte programmabile, è cioè possibile personalizzare in modo sufficientemente approfondito le funzionalità che rende disponibili.

Con questo intendiamo dire che esso comprende un processore, denominato *Kontakt Script Processor* in grado di interpretare degli script scritti in un linguaggio proprietario denominato *Kontakt Scripting Language*.

Gli script permettono di creare molto più di semplici "plugin" in quanto possono accedere a diverse funzionalità interne di Kontakt quali, ad esempio:

- gestione dei campioni audio,
- gestione di eventi e file MIDI,
- riproduzione di note (tramite i campioni),
- controllo dei parametri audio (filtri, forme d'onda, etc.),
- definizione e gestione di elementi di interfaccia utente.

Nonostante questo motore di scripting offra grosse potenzialità, a volte risulta complesso sfruttarle a pieno a causa di alcune limitazioni insite nel linguaggio stesso e nell'ambiente di programmazione.

Diamo di seguito una illustrazione di massima delle limitazioni caratteristiche del linguaggio:

- i *tipi di dato* si limitano ad interi e stringhe, nonché vettori monodimensionali dei medesimi;
- lo *scoping delle variabili* è esclusivamente globale, non esistono variabili locali, nonostante il linguaggio permetta una forte concorrenza: nella documentazione non viene specificato se l'accesso in memoria a tali variabili sia atomico o meno.

- le *espressioni* si limitano alle usuali espressioni aritmetiche per gli interi, a quelle logiche e, per le stringhe, alla sola concatenazione e comparazione (limitatamente all'uguaglianza);
- le *strutture di controllo* comprendono: la sequenza, la selezione (secondo il costrutto binario `if`/`else`, ed un costrutto ennario `select`/`case`, simile allo `switch` di C) e l'iterazione (col solo costrutto `while` e senza i costrutti di alterazione del flusso `break` o `continue`);
- è possibile definire *funzioni*, ma non è previsto il passaggio di parametri, o la restituzione di valori: tutta la comunicazione avviene tramite l'uso di variabili globali;
- non è possibile *modularizzare* il codice (suddividendolo in moduli che limitino la visibilità delle variabili), è si possibile avere fino a cinque script diversi in esecuzione, ma la comunicazione tra essi avviene con una sorta di meccanismo di memoria condivisa molto primitivo;

Il codice è organizzato in **callback**, una sorta di gestori di interruzione (*interrupt handler* nel senso dei sistemi operativi [20]), la cui esecuzione viene invocata dal motore in modo asincrono in corrispondenza di eventi esterni (come ad esempio eventi MIDI, o di interfaccia utente), o in modo periodico. Mancando però totalmente costrutti di gestione della concorrenza e non essendoci parola nella documentazione circa l'atomicità dell'accesso alle variabili, che ricordiamo sono solo globali [1], questa organizzazione del codice rende impossibile avere certezze su come evitare, ad esempio, condizioni di corsa (*race conditions*).

L'esecuzione parallela tra callback è gestita molto malamente in quanto c'è una sorta di mescolanza tra l'approccio cooperativo e quello *preemptive*: un callback può cedere il controllo (approccio cooperativo) invocando la funzione `wait` (che ne sospende l'esecuzione per un tempo specificato), ma se non lo fa entro un tempo prefissato, il controllo gli viene sottratto (approccio preemptive) e mai più restituito! Il comportamento appena descritto avviene, ad esempio, in presenza di cicli con un elevato numero di iterati (dell'ordine delle decine di migliaia) che non contengono chiamate alla funzione `wait`: in questo caso il motore ne interrompe l'esecuzione segnalando un errore (vedi Figura 2.1), ma senza fornire alcun modo per gestirlo. Ecco il motivo per cui, abbiamo visto nel manuale, che i cicli sono pieni di chiamate `wait(1)`, le quali non avrebbero altra spiegazione.

Figura 2.1: messaggio d'errore derivante da ciclo con "troppi" iterati.

[1] In realtà esistono variabli dette "polifoniche" che, almeno in contesti molto particolari come i callback relativi agli eventi nota, consentono una protezione dalle condizioni di corsa.

Detto questo, possiamo affermare che un linguaggio con le caratteristiche sopra descritte rende la programmazione alquanto ostica, esponendo il codice ad errori di programmazione molto difficili da individuare e controllare.

Oltre questi difetti insiti nel linguaggio stesso, ci sono delle limitazioni anche nelle funzioni di gestione dell'I/O che rendono macchinoso adoperare dati e informazioni esterne.

In ultima analisi dobbiamo dire che le limitazioni più eclatanti riguardano proprio la gestione dei file MIDI e questo riteniamo sia inammissibile per un software musicale. Esso permette di accedere ad un solo file MIDI per volta (esiste un solo buffer, denominato "MIDI object" dove è possibile caricare un solo file per volta). Inoltre non sono gestiti tutti gli eventi previsti dallo standard MIDI, ma soltanto gli short message; risulta quindi impossibile leggere dai file MIDI informazioni fondamentali quali il tempo, o il metro della misura.

Se la gestione dei file MIDI è limitata, quella di file di altro formato (ad esempio testo) è del tutto assente. Si possono salvare e caricare vettori (secondo un formato semplice ma proprietario), ma non si possono leggere file di nessun altro formato standard, come ad esempio file di configurazione (come i file `.ini` comuni in Windows o i `,properties` di Java).

2.3 Strumenti di supporto

2.3.1 Il preprocessore C

Il preprocessore C viene utilizzato automaticamente dal compilatore C per trasformare il codice sorgente prima di passare alla compilazione [24]. È anche detto macro processore perché consente di scrivere delle macro che sono delle "abbreviazioni" per porzioni di codice più lunghe.

Ad esempio, le seguenti righe di codice C:

```
#define GREET "ciao"
#define PRINTLN( X ) printf( "%s\n", X )

PRINTLN( GREET );
```

vengono tradotte dal preprocessore nel codice:

```
printf( "%s\n", "ciao" );
```

Nonostante il preprocessore viene utilizzato solitamente con il C, è possibile con qualche accortezza usarlo anche con il Kontakt Scripting Language.

Le caratteristiche del preprocessore che utilizzeremo sono:

Inclusione di file Abbiamo suddiviso il codice in vari file secondo le funzionilità logiche che dovevamo implementare e quindi usato la direttiva `#include` per ottenere da tali file lo script complessivo.

Compilazione condizionale Siccome le dichiarazioni di variabile possono comparire solo nel callback `init` e noi abbiamo diviso il codice in più file, è necessario all'atto di compilazione raccoglierle tutte insieme (inserendole nell'`init`). Per fare questo utilizziamo le direttive di compilazione condizionale `#ifdef`, `#else` e `#endif` per specificare quale porzione di codice debba comparire all'interno del callback `init`.

Commenti A causa di alcune incompatibilità, per poter introdurre testo senza vincoli nei commenti degli script li abbiamo inclusi in commenti stile C (racchiusi cioè tra `/*` e `*/`).

Macro di tipo "oggetto" Con questa definizione intendiamo delle macro, definite tramite la direttiva `#define`, usate come abbreviazioni di costanti. Ad esempio: il tipo di parte correntemente in esecuzione è un numero intero, ma nel codice usiamo le macro `INTRO`, `GROOVE`, `FILL` e `OUTRO` (invece dei valori interi cui corrispondono) per aumentare la leggibilità del codice. Per aggirare le differenze tra C e KSP in merito al preprocessore, definiamo una macro che vale `#`, questo carattere in C è "riservato" al preprocessore (non può essere dunque usato nella definizione delle macro) ma in KSP esso rappresenta l'operatore logico `diverso`.

Macro di tipo "funzione" Con questa definizione intendiamo delle macro, definite tramite la direttiva `#define`, usate come "approssimazione" di `funzioni con parametri` (che sono assenti in KSP). Si tratta anche in questo caso di una soluzione sintattica che ci aiuta moltissimo a migliorare la leggibilità e stesura del codice. Ad esempio: usiamo una serie di queste macro per definire e disporre gli elementi di interfaccia grafica (operazione che richiede un numero elevato e ripetitivo di chiamate a funzioni KSP).

Fusione delle linee Il preprocessore consente per ragioni di leggibilità di spezzare le linee di codice nel testo sorgente, provvedendo a concatenarle in un unica linea prima di procedere alla compilazione vera e propria; questo per il C va bene, ma purtroppo non per KSP, dove invece ogni linea può contenere al più una singola istruzione. Per questa ragione useremo uno stratagemma (basato su una ulteriore sostituzione ad opera di un tool diverso dal preprocessore) che consenta da un lato di dividere le macro più complesse su più linee, ma dall'altro di ripristinare la divisione delle medesime in linee (dopo la fusione effettuata dal preprocessore).

Infine possiamo dire che nonostante il preprocessore C sia uno strumento molto datato e da noi utilizzato fuori del suo ambito di utilizzo, ci ha consentito di ottenere enormi benefici nella

stesura del codice, permettendoci di modularizzarlo sia suddividendolo in file che utilizzando le macro e di aumentarne di gran lunga la leggibilità.

Nonostante sul mercato ci siano strumenti con funzionalità analoghe e, se vogliamo, più specifiche per KSP, abbiamo scelto di utilizzare il preprocessore per la sua stabilità e diffusione (esso è infatti uno strumento correntemente manutenuto e disponibile, con funzionalità sostanzialmente invariate da moltissimi anni), inoltre è uno strumento ben noto a chiunque abbia programmato in C.

2.3.2 Make

Lo strumento di costruzione `make` è un software in grado di effettuare in modo automatico i passi di un processo di compilazione che devono essere eseguiti.

Nel nostro caso dobbiamo infatti applicare una serie di passi ai file sorgente in cui abbiamo suddiviso il codice al fine di ottenere lo script finale. Nello specifico dobbiamo invocare il preprocessore e lo strumento che abbiamo usato per spezzare su più linee quelle fuse dal preprocessore.

Le ragioni per cui abbiamo deciso di utilizzare questo tool sono simili a quelle per cui utilizziamo il preprocessore: anche in questo caso si tratta di uno strumento estremamente diffuso, stabile e ben noto.

2.3.3 Java

Java [14] è un linguaggio orientato agli oggetti, originariamente rilasciato da Sun nel 1995, ed è ad oggi uno dei linguaggi più diffusi, in particolare nel mondo delle applicazioni web.

Tra le caratteristiche principali di questo linguaggio ci sono: la sua elevata portabilità (`Java` viene eseguito da un interprete, denominato `Java Virtual Machine`, il quale è disponibile sostanzialmente per tutti i sistemi operativi), il supporto nativo per la concorrenza ed il multithreading, la gestione automatica della memoria dinamica e una ricchissima dotazione di API.

In particolare, nell'ambito della tesi, abbiamo scelto di utilizzare Java proprio per il supporto offerto dalle sue API per la gestione del MIDI. Il pacchetto `javax.sound.midi` [13] consente di leggere e scrivere file multitraccia in formato MIDI standard e di analizzarne e modificarne il contenuto con estrema facilità.

2.3.4 Editor

Il *KScript Editor* [5], sviluppato da Nils Liberg, è uno degli editor più utilizzati fra i programmatori che usano il *Kontakt Scripting Language*. Oltre le usuali funzionalità di colorazione della sintassi e navigazione del codice, esso offre la possibilità di ampliare la sintassi originale del linguaggio offrendo costrutti più avanzati come ad esempio i cicli for, o un limitato supporto delle funzioni con parametri.

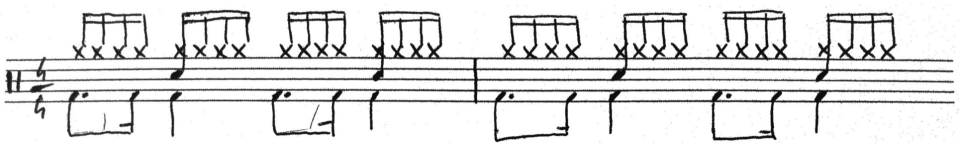
Dopo una prima analisi, abbiamo però deciso di non utilizzare tale editor. Principalmente perchè non volevamo legare il nostro codice ad uno strumento del quale non avessimo precise garanzie di stabilità e supporto. Secondariamente abbiamo preferito usare strumenti più *general purpose*, specie negli editor, dove non abbiamo trovato conveniente utilizzare un editor specifico per il solo *Kontakt Scripting Language*.

La nostra scelta si è orientata su *Sublime Text* [11], un editor molto diffuso per programmatori, che supporta un gran numero di linguaggi di programmazione, anche grazie alla disponibilità di un vasto numero di plugin, uno dei quali [4] specificatamente sviluppato, sempre da Nils Liberg, per KSP.

Rispetto alle estensioni del linguaggio offerte da *KScript Editor* abbiamo preferito, come discusso nelle sezioni precedenti, l'uso del preprocessore C, in quanto decisamente più diffuso e mantenuto.

CAPITOLO 3

Progetto e sviluppo dell'applicazione



In questo capitolo descriviamo l'applicazione che abbiamo sviluppato durante il periodo di stage. La suddivisione della narrazione in sezioni ricalca il flusso logico che l'informazione musicale, ossia i pattern, seguono dalla loro *produzione* alla rappresentazione tramite le *strutture dati* offerte dal linguaggio di programmazione, all'*elaborazione* che essi subiscono prima di essere *riprodotti* e quindi *esportati* per l'uso all'esterno del software qui descritto.

3.1 Produzione dei pattern

Come discusso nella Sezione 1.3.1, abbiamo l'esigenza di registrare diversi pattern per ciascun tipo al fine di raccogliere il materiale che costituisce la song. Questo processo è descritto nelle due Sezioni seguenti.

Nella produzione dei pattern è determinante una classificazione dei generi musicali che sia riconoscibile in modo naturale dall'utente [16]. Per questa ragione, prima di affrontare la fase di registrazione, è molto importante fare un'analisi che riguardi l'obiettivo della produzione e il range di diversità che questa dovrà coprire nel prodotto finito, limitando il numero di generi, sottogeneri, tempi, BPM a una selezione basata sul gusto, le richieste di mercato e la moda del momento.

3.1.1 Registrazione

I pattern possono essere generati in diversi modi:

- usando un editor all'interno di una DAW;
- tramite programmazione di "basso livello";
- tramite esecuzione con tastiera MIDI;
- tramite esecuzione con pad MIDI specifici per batteristi (detti *e-drums*).

In letteratura [23] [22], troviamo studi che consentono di aiutarci nella creazione di algoritmi efficaci nell'umanizzazione di pattern elaborati al computer.

Visto che uno degli obiettivi principali del lavoro è l'autenticità del materiale prodotto, abbiamo scelto l'ultima opzione, chiamando un batterista a eseguire i pattern su un kit e-drums.

I kit tipicamente utilizzati (per diffusione e per qualità fisica dei pad) sono quelli prodotti da Roland [9]: i modelli attualmente di punta sono la batteria TD15 e TD30 [12], riprodotta nella Figura 3.1.

Figura 3.1: Roland TD-30KV

Questi kit consentono un controllo fine delle caratteristiche esecutive, un'ottima resa dinamica e in particolare il charleston ha un controllo continuo e con interazione uomo-macchina molto simile a quella di un strumento reale [25].

La registrazione avviene in modo molto simile a qualsiasi altra produzione, chiedendo al batterista di suonare in maniera consecutiva i diversi pattern che comporranno la song, sfruttando a volte anche l'improvvisazione.

Un altro fattore che generalmente contribuisce alla qualità del risultato è far eseguire i pattern usando gli stessi suoni che poi verranno utilizzati all'interno del software prodotto, in quanto ciò consente al batterista e al produttore di avere una percezione più fedele del risultato finale.

Si è scelta questa tecnica perché garantisce risultati più espressivi da parte del musicista coinvolto, anche se comporta un maggiore dispendio di tempo in fase di post-produzione.

3.1.2 Post-produzione

Quando, per la produzione di musica, si utilizza il computer e lo standard MIDI [17], all'esecuzione del musicista spesso segue una fase del processo creativo denominata **post-produzione** tramite la quale si ottimizza la performance del musicista utilizzando strumenti in grado di manipolare i dati nel formato MIDI, cosa che sarebbe molto complesso fare se si fosse lavorato in ambito puramente audio.

Le modifiche tipiche includono: la trasposizione in semitoni, il ritardo o anticipo temporale e la variazione di altre informazioni MIDI quali la velocity.

In post-produzione, il lavoro si suddivide principalmente in:

- scelta dei pattern (tra quelli registrati);
- eventuale ri-creazione di pattern a partire da frammenti di esecuzioni diverse (qualora il batterista abbia improvvisato, o se dovesse mancare del materiale);
- filtraggio di eventi MIDI generati dal kit e-drums (per Roland tipicamente si parla di messaggi relativi alla posizione di impatto della bacchetta sul rullante, che costituiscono informazioni che noi non intendiamo usare nel nostro lavoro);
- trasformazione degli eventi MIDI: mappatura delle note MIDI secondo quanto richiesto dal plugin di destinazione [1], estensione del range di alcuni eventi [2];
- correzione musicale mediante trasformazione: si può, a seconda dei casi, applicare una pre-quantizzazione a pattern interi o loro parti per renderli più fluidi (è importante che passando da un pattern all'altro il "feel" umano rimanga intatto e costante).

3.1.3 Formato dei file

Al termine della post-produzione vengono prodotte, per ciascun song:

- un file multitraccia MIDI in cui ciascuna traccia corrisponde a un pattern;
- un file di testo contenente: il metro, il BPM e il numero di pattern per tipo.

Dopo aver raccolto il materiale descritto nella Sezione precedente abbiamo dovuto individuare un formato adatto all'elaborazione successiva che, per ciascuna song, oltre ai veri e propri eventi MIDI, ci consentisse di memorizzare:

- il numero e tipo dei vari pattern;

[1] Per esempio Roland mappa il charleston chiuso sulla nota F#1, mentre uno strumento diverso potrebbe associare lo stesso suono ad un'altra nota.

[2] Alcuni kit e-drums limitano il range del messaggio MIDI relativo all'apertura del charleston ad un massimo di 90, mentre noi vogliamo l'intero intervallo possibile di valori che si estende fino a 127.

- il metro della misura e il BPM.

È fuori discussione che tali informazioni non possano essere memorizzate nello script stesso. Questo costringerebbe infatti a modificare il codice ogni qual volta vengano modificati i file MIDI su cui opera, legando il codice sorgente dello script al suo input in modo non accettabile.

Date le limitazioni di KSP che non consentono di leggere agevolmente file di configurazione (testuali, o di altro formato) e di manipolare più di un file MIDI alla volta, abbiamo deciso di rappresentare le suddette informazioni in un unico file MIDI multitraccia.

Poiché, inoltre KSP consente di leggere solo short message abbiamo dovuto usare una serie di stratagemmi per rappresentare tramite essi le informazioni a noi necessarie.

Riguardo alla necessità di avere più pattern per tipo, abbiamo investigato diverse soluzioni possibili. Alcune erano molto pratiche dal punto di vista della stesura del codice, ma del tutto impratiche nella fase di postproduzione. La soluzione finale a cui siamo arrivati, che costituisce un ragionevole compromesso è la seguente.

Ogni song è memorizzata in un singolo file multitraccia in cui ogni traccia rappresenta un pattern di uno specifico tipo. Per dividere fra di loro i vari tipi (intro, groove, fill, outro), non potendo far uso di alcuna meta informazione, abbiamo usato una traccia ad hoc, da noi denominata **empty track** la quale è posizionata alla fine di ogni gruppo di tracce di un determinato tipo e funge da separatore.

La empty track è costituita dai seguenti eventi MIDI (qui descritti tramite il codice Java necessario a generarli):

```
new ShortMessage( NOTE_ON, 127, 127 );
new ShortMessage( NOTE_OFF, 127, 0 );
```

Riguardo al metro della misura, necessario per l'elaborazione successiva, lo standar MIDI prevede che questo e altri dati come il BPM, il nome della traccia…, siano memorizzati sotto forma di meta eventi in quanto non sono fondamentali per il MIDI in se stesso.

Non potendo, di nuovo per via dei limiti di KSP, usare la codifica standard MIDI, abbiamo deciso di rappresentare tali informazioni con short message inseriti in una traccia apposita da noi chiamata **meta track** che abbiamo posizionato come prima traccia del file.

La meta track presenta i seguenti eventi MIDI (di nuovo, descritti tramite il codice Java necessario a generarli):

```
new ShortMessage( NOTE_ON, numeratoreMetro, denominatoreMetro );
new ShortMessage( NOTE_ON, bpm / 128, bpm % 128 );
new ShortMessage( NOTE_OFF, numeratoreMetro, 0 );
new ShortMessage( NOTE_OFF, bpm / 128, 0 );
```

Lo strumento di supporto “InserisciTracceSpeciali”

Al fine di semplificare la fase di post-produzione abbiamo realizzato un programma Java che, dato in ingresso il materiale descritto nella Sezione 3.1, produce il file MIDI nel formato descritto nella Sezione precedente che può essere quindi direttamente utilizzato dal nostro script KSP.

3.2 Strutture dati per la rappresentazione dei pattern

Dopo una serie di sperimentazioni e prendendo in considerazione i limiti del linguaggio, abbiamo convenuto che il modo più consono per la rappresentazione di dati strutturati come gli eventi MIDI è l’utilizzo di vettori paralleli, ossia di una serie di vettori (uno per ciascun tipo di dato), in cui le informazioni relative al medesimo evento sono memorizzate allo stesso indice.

La prima operazione che effettuiamo è quindi il caricamento del file MIDI multitraccia all’interno di Kontakt; questo avviene popolando una serie di array paralleli il cui nome inzia per `%song`. Oltre le usuali informazioni in merito agli eventi MIDI, avremo bisogno in seguito anche di un altro dato e cioè, per ciascun NOTE ON, la posizione del suo NOTE OFF corrispondente. Nel dettaglio, per l’`i`-esimo evento i vettori conterranno:

- `%song_pos[ i ]`: la posizione in tick,
- `%song_cmd[ i ]`: il comando,
- `%song_byte_one[ i ]`: il primo byte dello *short message*,
- `%song_byte_two[ i ]`: il secondo byte dello *short message*,
- `%song_nod[ i ]`: se l’`i`-esimo evento è un NOTE ON, il valore `j - i`, dove `j` è l’indice del NOTE OFF corrispondente, altrimenti -1.

Poichè intendiamo caricare consecutivamente gli eventi di tutti i pattern, abbiamo bisogno di un modo per tener traccia della posizione iniziale di ciascun pattern; a tale scopo usiamo l’array `%patterns_offsets` tale per cui `%patterns_offsets[ p ]` è l’indice nei vettori `%song` del `p`-esimo pattern. Inoltre, per tener traccia di dove sono collocati i vari tipi, adopereremo l’array `%kinds_offsets` tale per cui `%kinds_offsets[ k ]` è l’indice in `%patterns_offsets` del primo pattern del tipo `k`-esimo (i tipi sono numerati da 0 e nel seguente ordine: intro, outro, groove e fill).

Tali vettori e le loro relazioni sono esemplificate dalla figura 3.2

Ad esempio gli eventi relativi al sesto fill occuperanno le posizioni dei vettori `%song` che vanno dall’indice `inizio` a `fine` (compresi), calcolati come segue:

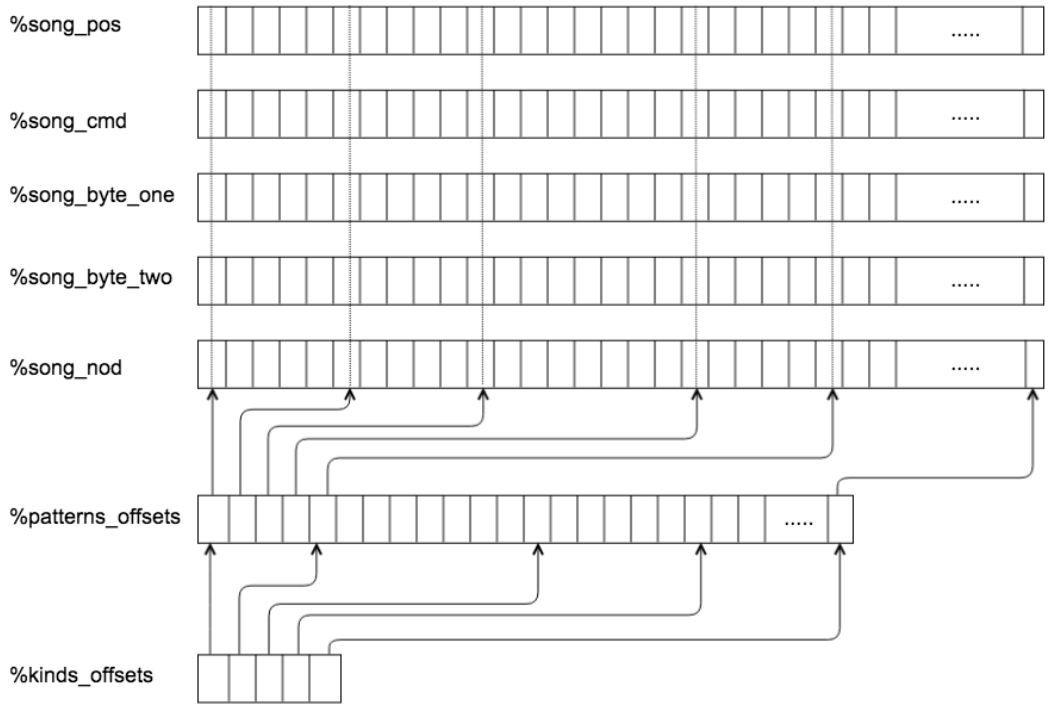


Figura 3.2: rappresentazione visuale dei vettori che contengono la song.

```
k := 3 { 3 corrisponde al tipo fill }
p := %kinds_offsets[ k ] + 5 { il sesto fill ha indice 5 }
inizio := %patterns_offsets[ p ]
fine := %patterns_offsets[ p + 1 ] - 1
```

3.3 Elaborazione dei pattern

Prima di riprodurre un pattern, potremmo voler modificare alcune sue caratterische; a tale scopo abbiamo deciso di mantenere all'interno dei vettori `%song` i dati *originali*, e di introdurre una serie di vettori paralleli `%play` in cui verrà di volta in volta copiato, o fuso, uno o più pattern da trasformare e riprodurre.

3.3.1 Fusione di groove e fill/outro

Quando si passa da un pattern a un altro di diverso tipo, è a volte necessario "fondere" le parti piuttosto che suonarle una in sequenza all'altra.

Ad esempio, quando un batterista sta suonando un groove e vuole passare ad un fill, lo "incastra" nel punto giusto, senza aspettare che sia prima finito il groove. Stesso discorso se il passaggio avviene da groove ad outro.

Facciamo un esempio per capire meglio il punto: usiamo come riferimento la coppia "groove - fill" e scegliamo come metro un 4/4, un groove di 16/4, cioè 4 battute, e un fill di 6/4, cioè una battuta e mezza, come in Figura 3.3.

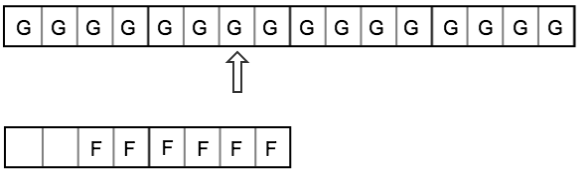


Figura 3.3: groove e fill da fondere, come presenti nei vettori `%song`.

Il batterista, dopo aver eseguito il terzo quarto della seconda battuta (indicato dalla freccia) vuole inserire il fill. L'operazione di fusione può essere descritta come segue:

- si copiano le `n` misure di groove gia interamente suonate (nell'esempio, `n := 1`),
- il fill viene traslato in avanti di `n` misure,
- al posto dei quarti vuoti del fill si copiano i corrispondenti del groove (nell'esempio, i primi due),
- si riempiono i quarti restanti (fino alla lunghezza originale del groove) copiando i quarti del groove (nell'esempio, gli ultimi 4);

questa sequenza di operazioni riempie i vettori `%play` come in Figura 3.4.

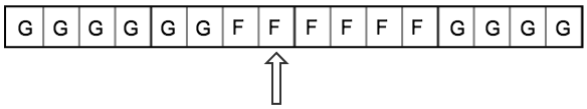


Figura 3.4: risultato della fusione, presente nei vettori `%play`.

A questo punto il batterista procede suonando il quarto quarto della seconda misura (indicato dalla freccia). Osserviamo che la fusione ha riguardato anche i quarti precedenti a quelli già eseguiti, questo al fine di consentire maggiore coerenza in fase di esportazione dei dati, come sarà descritto in seguito.

Per gli outro il discorso è lo stesso, con la differenza che al termine dei medesimi, la composizione sarà finita e quindi non sarà necesserio considerare gli eventuali quarti rimanenti del groove.

3.3.2 Trasformazioni

Una volta popolati i vettori `%play` con i pattern da riprodurre il software consente di applicare delle trasformazioni che riguardano il tempo e l'intensità. Queste trasformazioni alterano direttamente i valori dei vettori `%play`; le descriviamo qui in seguito.

Quantizzazione

Spesso un ritmo che risulta essere *a tempo* e musicalmente corretto per l'orecchio umano, non lo è per la precisione metronomica della macchina, ecco che entra in gioco la **quantizzazione**.

Abbiamo deciso di inserire questa funzionalità nel software, piuttosto che attuare una quantizzazione nella fase di post-produzione, perché avendo utilizzato un batterista umano per registare i pattern abbiamo voluto preservare il più possibile il timing e il "feel" della sua esecuzione.

Precisamente, quantizzare un pattern significa allineare tutti gli eventi (modificandone i *tick*) rispetto ad una griglia di riferimento, nel nostro caso la griglia corrisponde a una tra le seguenti figure musicali: ottavo, ottavo terzinato, sedicesimo, sedicesimo terzinato, trantaduesimo, trentaduesimo terzinato (queste figure sono dette *risoluzione di quantizzazione*).

Per ogni evento MIDI consideriamo il tick t a cui avviene e calcoliamo la differenza Δt corrispondente allo spostamento necessario a riallinearlo alla griglia scelta come segue:

$$\Delta t = \begin{cases} -(t \bmod q)\alpha & \text{se} \quad t \bmod q < q/2 \\ (q - t \bmod q)\alpha & \text{altrimenti} \end{cases}$$

dove $0 \leq \alpha \leq 1$ è un parametro che descrive quanto rigorosa deve essere la quantizzazione: 0 corrisponde al caso in cui le note non verranno spostate e 1 al caso in cui verranno portate esattamente sulla griglia.

La porzione di codice che realizza il calcolo di Δt è la seguente:

```
$old_pos := %play_pos[ $play_idx ]
if ( $old_pos mod $quant_resol <= $quant_resol / 2 )
  $pos_offset :=
    - ( $old_pos mod $quant_resol )
      * 1000 * $quantize_selector / 100000
else
  $pos_offset :=
    ( $quant_resol - ( $old_pos mod $quant_resol ) )
```

```
      * 1000 * $quantize_selector / 100000
end if
```

dove: `$pos_offset` corridponde a Δt, `$quant_resol` a q e adoperiamo lo stratagemma di moltiplicare per `1000 * $quantize_selector / 100000` per approssimare il calcolo del prodotto per α, stratagemma reso necessario dal fatto che il linguaggio supporta solo tipi interi.

Al fine di quantizzare soltanto la posizione delle note e non la loro durata, quando applichiamo tale traslazione ad un evento NOTE ON, sposteremo dello stesso Δt anche l'evento NOTE OFF corrispondente che reperiremo usando il vettore `%play_nod` descritto in precedenza.

Come esemplificato dalla Figura 3.5, se t è pari a 150, con la risoluzione pari a 240 (cioè un sedicesimo) α pari a 1, Δt assumerà valore 90.

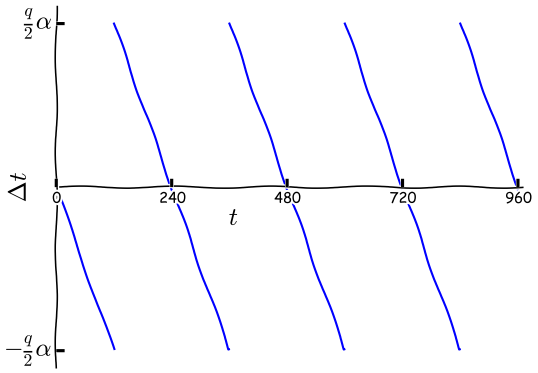


Figura 3.5: spostamento Δt per la *quantizzazione* in funzione di t.

Swing

Ai musicisti è ben noto che suonando un tempo binario come terzinato gli si conferisce un gusto afro-americano, altrimenti detto **swing** [3], che è la denominazione usata nei software musicali.

Consideriamo più concretamente un quarto composto da quattro sedicesimi, possiamo ottenere lo swing spostando in avanti i sedicesimi pari di ogni ottavo. Se descriviamo la misura di questo

[3] il termine *swing* è molto generale e ha più significati, con esso possiamo riferirci a:

- un ritmo sincopato che è caratteristica comune della musica jazz;
- uno stile musicale conosciuto come "swing jazz";
- una serie di balli che includono il Lindy Hop, West Coast Swing ecc..

Nel nostro caso, lo considereremo nella prima accezzione.

spostamento in termini della percentuale c di tempo all'interno dell'ottavo assegnata a ciascun sedicesimo si danno ad esempio i seguenti due casi:

- se c è pari al 50% i sedicesimi restano al loro posto in quanto occupano ciascuno la metà di un ottavo,
- se aumentiamo c al 66%, il secondo e quarto sedicesimo cadranno nella posizione del terzo e sesto sedicesimo terzinato,
- per valori diversi di c si possono comunque ottenere effetti interessanti.

In generale, considerando una griglia di riferimento corrispondente a uno tra i seguenti valori: trentaduesimo, sedicesimo, ottavo; è possibile calcolare lo spostamento Δt da applicare all'evento al tick t secondo la seguente formula:

$$\Delta t = \begin{cases} \left(\frac{s}{2} - t \bmod s\right) + s\alpha & \text{se} \quad \left|t \bmod s - \frac{s}{2}\right| \leq s\alpha \\ 0 & \text{altrimenti} \end{cases}$$

dove $0 \leq \alpha \leq 2/10$ è un parametro che descrive quanto maggiore deve essere lo swing: 0 corrisponde al caso in cui abbiamo meno swing e 2/10 al caso in cui abbiamo maggior swing. Il parametro α è determianto a partire dalla percentuale di swing c che, come discusso in precedenza, assume valori in $[50, 70]$ tramite la relazione $\alpha = (c - 50)/100$.

Riportiamo anche in questo caso la porzione di codice che realizza il calcolo di Δt:

```
$swing_delta := ( $swing_resol *
  ( $swing_selector - 50 ) * 1000 ) / 100000
if ( abs( $old_pos mod $swing_resol - $swing_resol / 2 ) <= $swing_delta )
  $pos_offset :=
    ( $swing_resol / 2 - ( $old_pos mod $swing_resol ) ) + $swing_delta
else
  $pos_offset := 0
end if
```

dove: `$pos_offset` corridponde a Δt, `$swing_resol` a s e calcoliamo `$swing_delta` che corridponde approssimativamente a $s\alpha$ di nuovo usando lo stratagemma `( ... * 1000 ) / 100000` reso sempre necessario dal fatto che il linguaggio supporta solo tipi interi.

Anche nel caso dello swing, modificheremo soltanto la posizione delle note e non la loro durata, applicando la stessa traslazione Δt ad ogni NOTE ON e NOTE OFF corrispondente.

Come esemplificato dalla figura 3.6, se t è pari a 720, con la risoluzione pari a 480 e c pari al 66%, ossia α pari a 1/6, Δt è pari a 80 che infatti sposta 720 a 800 che è la posizione del sesto sedicesimo terzinato.

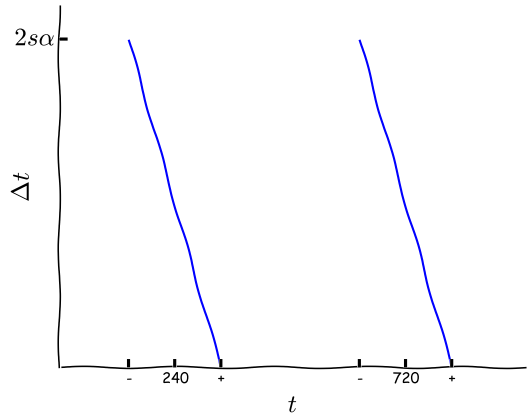


Figura 3.6: spostamento Δt per lo *swing* in funzione di t.

Tempo

Questa trasformazione serve a modificare il tempo della traccia, dando la possibilità all'utente di scegliere se raddopiarlo o dimezzarlo, approssimando quelli che in letteratura vengono definiti "double time feel" e "half time feel" [16].

Osserviamo che essa agisce alterando i *tick* (raddoppiandone o dimezzandone il valore) e non modificando il BPM della traccia.

Anche in questo caso, il limite d'avere a disposizione solo tipi interi ci obbliga al codice poco naturale:

```
%play_pos[ $play_idx ] :=
   (  %play_pos[ $play_idx ] * $stretch_amount  ) / 2
```

dove `$stretch_amount` ha valori 4, 2, o 1 invece dei più ovvi 2, 1 e 1/2.

Intensità

Può essere necessario modificare l'intensità e quindi il volume e timbro risultante delle note che sono suonate; questo, in un evento MIDI, corrisponde a modificare il parametro *velocity* che, secondo lo standard, può assumere valori nell'intervallo $[0, 127]$.

Nel nostro caso, partendo da un parametro c che assume valori nell'intervallo $[-126, 127]$, determineremo un nuovo intervallo di variazione per la velocity pari a $[v_{\min}, v_{\max}]$ secondo la seguente formula:

$$v_{\min} = \max\{1, c\}$$
$$v_{\max} = \min\{c + 127, 127\}$$

che è esemplificata dalla Figura 3.7.

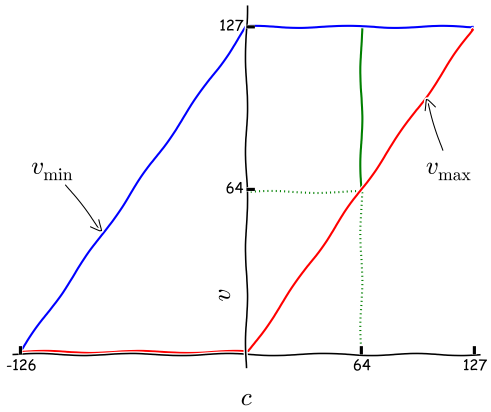


Figura 3.7: grafico di $v_{\min}$ e $v_{\max}$ in funzione del controllo c.

Ad esempio, per un valore di c pari a 64, si determina il nuovo intervallo di variazione $[64, 127]$.

Osserviamo che l'effetto udibile di valori estremi di c è l'appiattimento della dinamica del pezzo, con la conseguente perdita degli accenti.

Una volta deterimanto il nuovo intervallo di variazione, la velocity di v di un evento viene modificata (con una trasformazione lineare) nella nuova velocity v' come:

$$v' = (v - 1)\frac{v_{\max} - v_{\min}}{126} + v_{\min}$$

Di nuovo, secondo l'esempio precedente, la Figura 3.8 riporta la trasformazione dell'intervallo $[0, 127]$ in $[64, 127]$ corrispondente al valore 64 di c:

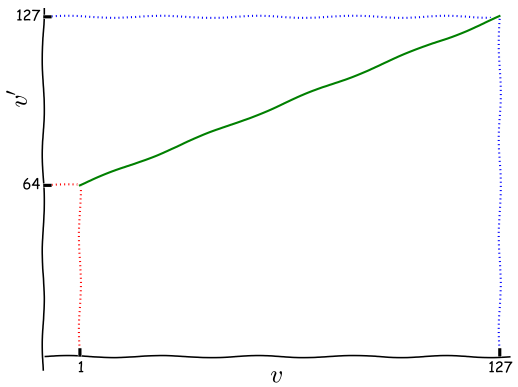


Figura 3.8: esempio di trasfomazione della velocità, quando c vale 64.

3.4 Riproduzione dei pattern

Il punto di forza del nostro software è la possibilità che offre all'utente di comporre le sequenze ritmiche a suo piacimento, è perciò naturale aspettarsi che la riproduzione dei pattern segua una logica non banale. Essa può essere descritta efficacemente dal diagramma in Figura 3.9.

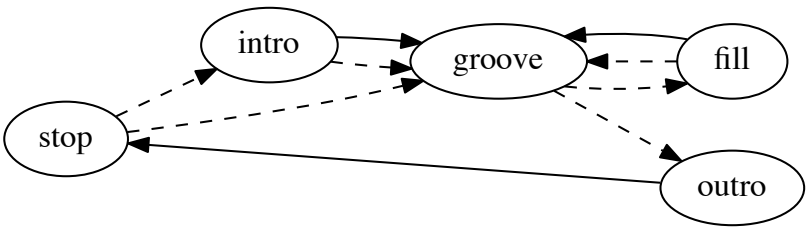


Figura 3.9: diagramma di stato della riproduzione.

I possibili stati corrispondono al tipo di pattern in riproduzione, o a *stop*. Non tutte le transizioni di stato sono possibili, questo a causa della struttura della parte ritmica in una composizione musicale: nel diagramma indichiamo con una linea tratteggiata quelle che l'utente può *causare tramite l'interfaccia utente* (a meno di quelle verso lo stato di stop, che sono sempre possibili), vicerversa con una linea continua quelle che risultano *necessarie* per via della struttra ritmica (ed avvengono al termine del pattern).

Le questioni principali che abbiamo affrontato nell'implementazione della riproduzione dei pattern sono stati i seguenti:

- la riproduzione degli eventi MIDI con la corretta temporizzazione,
- la gestione delle transizioni di stato,

che tratteremo in dettaglio nelle Sezioni seguenti.

3.4.1 Temporizzazione

Una volta che il pattern è stato preparato nei vettori `%play`, per riprodurlo è necessario impartire a KSP una serie di comandi di NOTE ON e NOTE OFF in corrispondenza dei *tick* indicati in `%play_pos`.

Una funzionalità essenziale offerta da KSP è il callback **listener** che viene invocato periodicamente da Kontakt allo specificato intervallo di tick; osserviamo che il legame tra i tick ed il tempo (in secondi) è gestito da Kontakt in due modi sostanzialmente diversi:

- in funzione dei BPM impostati nel software stesso (vedi Figura 3.10) ,
- oppure in funzione della temporizzazione derivante dalla DAW in cui viene utilizzato.

Figura 3.10: temporizzazione gestita dal “master” di Kontakt.

Per riprodurre il pattern, procediamo nel modo seguente:

- suddividiamo il tempo della riproduzione in beat,
- configuriamo il listener in modo che venga invocato una volta per beat,
- riproduciamo gli eventi all’interno del beat corrente tramite un ciclo.

Per gestire la temporizzazione all’interno del ciclo, per ogni coppia di eventi consecutivi, consideriamo la loro distanza in tick `$diff` e effettuiamo la chiamata:

```
wait( tick_to_ms( $diff ) )
```

dove `wait` è una funzione che attende lo specificato numero di microsecondi e `tick_to_ms` è una funzione che converte una durata in tick nella corrispondente durata in microsecondi in accordo alla logica di temporizzazione discussa in precedenza.

3.4.2 Transizioni di stato

Da un punto di vista musicale, le transizioni tra pattern non possono avvenire in momenti arbitrari, ma il buon senso suggerisce che accadano solo in occasione del passaggio:

- tra un beat ed il successivo,
- tra una misura e la successiva;

sebbene dal punto di vista musicale i cambi alla misura risultino più sensati (perché, ad esempio, riproducono un pattern nella sua completezza), per consentire una maggiore libertà durante la riproduzione e dare maggiore responsività all’interfaccia utente, abbiamo considerato anche i cambi al beat.

Osserviamo che, come discusso in precedenza, alcune transizioni sono originate dall’interfaccia utente e potrebbero causare pertanto eventi asincroni rispetto al flusso di esecuzione del programma (eventi cioè che possono avvenire in qualunque istante non deciso in anticipo dal programmatore).

D'altro canto, la nostra scelta di riprodurre i pattern beat per beat unita alla decisione di consentire transizioni solo in occasione del beat, o della misura, ci ha permesso una sorta di gestione sincrona.

Più in dettaglio, gli eventi di interfaccia utente sono gestiti (in modo asincrono) dai relativi callback che si limitano ad annotare in opportune variabili le modifiche di stato richieste dall'utente. Periodicamente, ad ogni beat, il codice del listener verifica se è necessario attuare cambiamenti di stato, sia a causa dell'interfaccia utente che delle transizioni necessarie, e li effettua (in modo sincrono) prima di procedere alla riproduzione del beat corrente.

Questo approccio ha di molto semplificato la stesura del codice ed ha risolto una quantità di problemi di conccorrenza e sincronizzazione che sarebbe stato molto arduo trattare dati i limiti imposti da KSP.

3.4.3 Interfaccia utente

L'interfaccia utente (UI) è importante in qualsiasi programma poiché determina l'efficacia e validità pratica degli obiettivi che ci si è preposti.

È fondamentale differenziare, come viene fatto in letteratura, la generica UI dalla Graphical User Interface (GUI) [21]. La seconda è parte integrante della prima, ma ne costituisce l'aspetto più esteriore e superficiale.

Essendo questa una prima implementazione della nostra soluzione, non ci siamo soffermati a creare una vera e propria GUI, ma abbiamo usato gli elementi di GUI offerti da Kontakt, con il solo fine di poter verificare le funzionalità implementate.

Abbiamo, però, avuto l'accortezza di usare elementi di interfaccia che potessero essere customizzati in futuro (non tutti gli elementi di GUI di Kontakt possono esserlo): come si vede dalle Figure 3.11 e 3.12 alcune scelte non sembrano visualmente accattivanti, ma sono le uniche che ci permetteranno di creare una vera e propria GUI.

Abbiamo scelto queste due figure, in quanto rappresentano i due stati principali dell'applicazione, la Figura 3.11 rappresenta lo stato di stop mentre la Figura 3.12 quello di play, in particolare mostra l'interfaccia mentre un groove è in riproduzione.

Vediamo i dettagli dell'interfaccia, sono presenti:

- il *file selector* che consente di scegliere la song da caricare;
- i bottoni che consentono di cambiare lo stato e sono denominati come i tipi (intro, outro, groove e fill), osserviamo che non sono tutti visibili allo stesso tempo per via delle transizioni illustrate nella Figura 3.9;

Figura 3.11: l'interfaccia utente nello stato stop.

- gli *slider* [4], sotto ogni tipo, i quali permettono di scegliere uno tra i pattern disponibili;
- i bottoni che consentono di scegliere se effettuare le *transizioni al beat o alla misura*;
- i bottoni che consentono di scegliere uno tra le tre opzioni offerte dalla *trasformazione del tempo*;
- il bottone *export* che consente di esportare il pattern, eventualmente fuso e trasformato, in esecuzione;
- gli *slider* che permettono di controllare i paramentri delle trasformazioni *quantizzazione*, *swing*, e *intesità*;
- il menù che consente di scegliere la risoluzione della quantizzazione e swing;
- alcuni elementi informativi tra i quali spicca quello (posizionato sopra l'etichetta swing) che permette di seguire il beat in esecuzione all'interno della misura.

3.5 Esportazione dei dati

Una volta che l'utente ha composto e trasformato i pattern a suo piacimento nei vettori `%play` (ed ha verificato tramite la riproduzione del medesimo le sue scelte), il programma mette a disposizione una funzionalità di esportazione che consente di utilizzarne il contenuto al di fuori dello stesso.

[4]Questo è uno di quei casi in cui avremmo preferito usare un'altro elemento di interfaccia, chiamato *knob*, ma abbiamo optato per uno slider a causa della sua capacità di essere personalizzato graficamente in una fase successiva.

Figura 3.12: l'interfaccia utente nello stato groove.

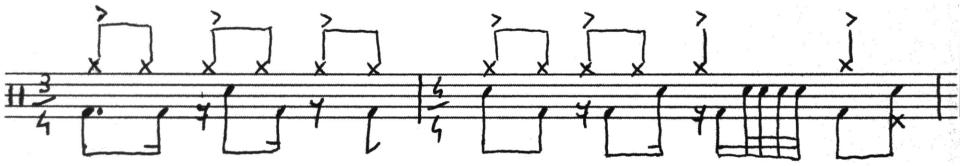
Osserviamo che la fusione e tutte le operazioni di trasformazione sono state progettate in modo che i vettori `%play` presentino un contenuto coerente con la composizione ritmica: se ad esempio un utente ha effettuato un cambio al beat tra groove e fill perdendo alcuni beat del fill, nei vettori `%play` (e quindi nell'esportazione), troverà invece tutti i beat del fill alla loro posizione corretta.

Da un punto di vista più implementativo, l'esportazione avviene secondo i passi seguenti:

- dapprima, usando i dati dei vettori `%play` viene popolato il "MIDI object" che è il buffer in cui Kontakt memorizza i dati MIDI;
- una volta pronto il "MIDI object", viene attivato uno degli elementi dell'interfaccia grafica al quale è attribuito un comportamento *drag and drop* che, se trascinato, produrrà appunto l'esportazione dei dati dal buffer alla DAW, o ad un file nel sistema ospite.

CAPITOLO 4

Conclusioni e sviluppi futuri



Concludiamo la tesi con uno sguardo d'insieme sui risultati ottenuti e con una proposta per i possibili sviluppi futuri.

4.1 Il risultato conseguito

Il nostro obiettivo è stato progettare e sviluppare un software di supporto alla creazione della sezione ritmica di una composizione musicale tenendo ben presenti le esigenze e le competenze del musicista.

La materia prima su cui tale applicazione lavora sono i **pattern**, ossia le parti ritmiche. Il primo passo del nostro lavoro è quindi stato quello di selezionare e categorizzare i generi musicali in modo opportuno, nonché produrre i relativi pattern con l'ausilio di un batterista umano, in modo da preservare il *timing* e il *feel* del musicista scelto, per offrire la massima autenticità possibile all'utente dell'applicazione.

Abbiamo suddividiso i pattern in **tipi** seguendo la classica struttura ritmica di una canzone: *intro*, *groove*, *fill* e *outro*. Abbiamo raccolto in **song** i pattern che si prestassero ad essere eseguiti nella medesima composizione rispettando una logica di genere e gusto. Abbiamo quindi costruito una interfaccia utente che rende possibile scegliere un song, selezionarne i pattern per tipo e comporli nel rispetto della struttura ritmica che abbiamo definito. In questo modo l'utente sceglie con semplicità collezioni di pattern che risultano già ben abbinati, ma ha la libertà di comporli a suo piacimento.

Con lo scopo di preservare l'*autenticità* dell'esecuzione del musicista autore dei pattern li abbiamo memorizzati dopo un processo di post-produzione il meno invasivo possibile. D'altro canto, per aumentare la possibilità di personalizzazione, abbiamo dotato il software di funzioni che trasformano i pattern effettuando dei cambiamenti a livello di tempo (quantizzazione e swing) e a livello di intensità. Il software adotta un simile compromesso tra la fedeltà ai pattern originali (che comprende la garanzia di andare "a tempo") e la libertà espressiva dell'utente consentendogli di cambiare tra un pattern e l'altro solo al cadere del prossimo beat (unità di tempo della misura) o alla misura.

In fine, considerando che questa applicazione potrebbe essere integrata in un contesto più ampio, assieme ad altri strumenti dello stesso tipo, e il suo sviluppo sarà soggetto ad una prosecuzione (con particolare riguardo all'interfaccia utente), uno dei requisiti di progetto è stato quello di modularizzare ed organizzare il codice in modo da facilitare questi passi successivi. Questa esigenza è stata soddisfatta, tra l'altro, tramite l'utilizzo di strumenti di supporto allo sviluppo che hanno consentito di scrivere il codice in modo ben organizzato, modulare e documentato.

4.2 Nuove funzionalità

La raccolta del materiale audio, la progettazione e sviluppo dell'applicazione, nonché le fasi di validazione del lavoro svolto hanno comportato un notevole sforzo e hanno richiesto molto tempo. Questo anche in funzione delle criticità del linguaggio di programmazione offerto dallo strumento che abbiamo scelto e alla scarsezza della documentazione che, in alcuni casi, hanno reso necessarie molteplici iterazioni di sviluppo prima di poter conseguire i risultati che ci eravamo prefissati.

Durante lo svolgimento dello stage, sono emerse una serie di possibili migliorie da apportare al software che non è stato possibile attuare nel tempo a disposizione; per questa ragione, ne diamo una breve descrizione nel seguito con l'intento di tracciare una possibile strada per gli sviluppi futuri.

4.2.1 Gestione a più dimensioni dei pattern

Chiunque ascolti un batterista alle prime armi ed un professionista improvvisare su uno stesso ritmo, comprenderebbe immediatamente cosa intendiamo nel definire più "complessa" l'esecuzione del secondo rispetto a quella del primo: essa avrà più sfumature, più variazioni ritmiche, più colore.

Ci piacerebbe che il nostro software consentisse all'utente, dato un pattern, di variarne la complessità (nel senso dell'esempio precedente). Per di più, ci piacerebbe consentire di variare

in modo indipendente la complessità di “gruppi” di pezzi della batteria, come ad esempio: il rullante, la cassa, i piatti e via discorrendo.

Similmente, come ora il nostro software consente di variare l’intensità complessiva di un pattern, ci piacerebbe che anche questo parametro potesse essere modificato in modo indipendente per i vari gruppi di pezzi.

In qualche senso, vorremmo implementare una variazione a “tre dimensioni” delle caratteristiche di un pattern dove:

- la prima dimensione rappresenti i diversi gruppi;
- la seconda dimensione le possibili complessità;
- la terza dimensione i vari gradi di intensità.

Il caso delle percussioni

La struttura a più dimensioni diventa di particolare interesse nel caso delle percussioni propriamente dette come, ad esempio, bonghi, congas, timbales, tamburelli, triangoli.

Se infatti nella batteria può essere considerato caso abbastanza raro il voler modificare complessità ed intensità per gruppi di pezzi (comunque suonati da un solo musicista), in un’accompagnamento basato su percussioni (suonate in genere da diversi musicisti) è viceversa molto comune voler trattare ciascuno strumento in modo indipendente.

Una gestione a più dimensioni dei pattern potrebbe essere quindi di grande interesse qualora fosse relizzata in modo sufficientemente generale da adattarsi al contempo sia al caso della batteria, che a quello delle percussioni.

4.2.2 Trasformazioni orientate ai pattern percussivi

Il nostro software si occupa esclusivamente di pattern percussivi, ma al momento le sue funzioni di trasformazione non tengono conto di questa specificità così marcata.

Ad esempio, la quantizzazione come descritta nella Sezione 3.3.2.1, agisce in modo omogeneo su tutti gli eventi e in presenza di un *flam* [1] “schiaccerebbe” i due colpi da cui è costituito in uno solo, facendone di fatto perdere la sfumatura.

[1] Un *flam* consiste in due colpi singoli suonati da mani alterne: il primo colpo meno intenso, seguito da un colpo primario più forte della mano opposta. I due colpi sono suonati quasi simultaneamente e sono intesi suonare come un sol colpo.

Ci piacerebbe investigare come riconoscere i *flam*, o simili casi particolari, in modo automatico (o grazie ad una speciale "marcatura" degli eventi effettuata in fase di post-produzione) in modo da poter sviluppare funzioni di trasformazione più specifiche.

Secondo l'esempio precedente, potremmo sviluppare delle funzioni di trasformazione temporale che offrano:

- una forma di quantizzazione che rispetti tali casi (non alterando il tempo degli eventi coinvolti);
- una sorta di "semplificazione" che, viceversa, sia in grado di eliminarli.

4.2.3 Interfaccia utente

Osservando il risultato finale dell'applicazione, è evidente che l'interfaccia utente che abbiamo sviluppato è quanto meno minimale. Come già detto, dati i vincoli temporali, ci siamo infatti limitati a mettere a disposizione dell'utente, con chiarezza e semplicità d'uso, tutte le funzionalità implementate dal software, pur senza pretesa di farlo con un aspetto accattivante.

D'altro canto, se vogliamo che questa applicazione si possa rivolgere ad un pubblico più ampio, diventando magari un prodotto commerciale, sarà necessario intraprendere uno studio più approfondito dell'interazione con l'utente, ad esempio secondo quanto suggerito in [18]. Dopo tale studio, si potrebbe procedere ad implementare elementi di interfaccia personalizzati, pur nei limiti imposti da NI Kontakt che non consente un completo controllo della GUI.

Bibliografia

[1] Chocolate audio. http://http://www.chocolateaudio.com/.

[2] Ezdrummer 2. http://www.toontrack.com/product/ezdrummer-2/.

[3] Garageband. https://www.apple.com/mac/garageband/.

[4] Ksp plugin for sublime text. http://goo.gl/78L7bb.

[5] Kscript editor. http://www.nilsliberg.se/ksp/.

[6] Kontakt5. http://www.native-instruments.com/products/komplete/drums/studio-drummer/.

[7] Logic pro x. https://www.apple.com/logic-pro/.

[8] Native instruments. http://www.native-instruments.com/.

[9] Roland. http://www.rolandus.com/.

[10] Studio drummer. http://www.native-instruments.com/en/products/komplete/drums/studio-drummer/.

[11] Sublime text editor. http://www.sublimetext.com/.

[12] Td-30kv. http://www.rolandus.com/products/details/1206/483.

[13] Javax.sound.midi. http://docs.oracle.com/javase/7/docs/api/javax/sound/midi/package-summary.html.

[14] K. Arnold, J. Gosling, and D. Holmes. *Il linguaggio Java. Manuale ufficiale*. Pearson, 2006.

[15] MIDI Manufacturers Association. *Complete MIDI 1.0 Detailed Specification*. 1999/2008.

[16] M. Berry and J. Gianni. *The Drummer's Bible: How to Play Every Drum Style from Afro-Cuban to Zydeco*. See Sharp Press, 2004.

[17] D. Clackett. *Handbook of MIDI Sequencing*. PC Publishing, 1996.

[18] S. Holland, K. Wilkie, P. Mulholland, and A. Seago. *Music and Human-Computer Interaction*. Springer Series on Cultural Computing. Springer London, 2013.

[19] Nicki, Marinic and Adam Hanley. *KSP Reference Manual, v5.3*. 2013.

[20] A. Silberschatz, P.B. Galvin, G. Gagne, V. Marra, and P. Codara. *Sistemi operativi. Concetti ed esempi*. Pearson, 2009.

[21] Joel Spolsky. *User interface design for programmers*. Apress Series. Apress, 2001.

[22] Ryan Stables, Cham Athwal, and Rob Cade. Drum pattern humanization using a recursive bayesian framework. In *Audio Engineering Society Convention 133*. Oct 2012.

[23] Ryan Stables, Jamie Bullock, and Ian Williams. Perceptually relevant models for articulation in synthesised drum patterns. In *Audio Engineering Society Convention 131*. Oct 2011.

[24] Richard M. Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Paramount, CA, 2009.

[25] Stephen J. Welburn and Mark D. Plumbley. Rendering audio using expressive midi. In *Audio Engineering Society Convention 127*. Oct 2009.

I groove riportati all'inizio dei capitolo sono, nell'ordine:

- *Rosanna*, scritta da David Paich ed eseguita dal batterista **Jeff Porcaro** della band americana *Toto*; è il pezzo di apertura (e primo singolo) dell'album *Toto IV* del 1982.
- *Funky Drummer*, scritta da James Brown e la sua band, eseguita dal batterista **Clyde Stubblefield**; comparsa come singolo nel 1969, è uno dei groove più campionati della musica Pop.
- *Africa*, scritta da David Paich e **Jeff Porcaro**, eseguita sempre dalla band *Toto*; è uno degli altri singoli dell'album *Toto IV*.
- *The Sound of Muzak*, scritta da Steven Wilson ed eseguita dal batterista **Gavin Harrison** della band inglese *Porcupine Tree*; è uno dei brani dell'album *In Absentia* del 2002.