Conservatorio di Musica “Licinio Refice” di Frosinone
Dipartimento di Nuove Tecnologie e Linguaggi Musicali
Diploma Accademico di I Livello in Discipline Musicali

Tesi di laurea in Musica Elettronica

Live Coding:
Costruzione di un sistema per l’improvvisazione algoritmica

Relatore: Prof. Alessandro Cipriani

Candidato: Riccardo Ancona
Matricola: T10974

Anno Accademico 2019-2020

Indice

Introduzione | 1

Capitolo I: Contesto storico-sociologico
- 1.1 Definizione di Algorave | 3
- 1.2 I precursori del live coding | 6
- 1.3 Breve storia del live coding | 12
- 1.4 Il network globale del live coding | 16
- 1.5 Hacking culture, Do It Yourself e Open-Source | 20

Capitolo II: La pratica del live coding: implicazioni performative
- 2.1 L'improvvisazione algoritmica | 25
- 2.2 Il codice come partitura dinamica | 30
- 2.3 Logiche e potenzialità della scrittura algoritmica | 32
- 2.4 Aspetti semiotici e audiovisivi del live coding | 38

Capitolo III: Definizione di un sistema di live coding
- 3.1 Software e librerie per il live coding | 42
- 3.2 Parametri per la scelta di un linguaggio di programmazione | 46
- 3.3 TidalCycles: architettura, sintassi e pattern design | 52
- 3.4 Haskell: vantaggi di un linguaggio funzionale puro | 57
- 3.5 SuperCollider come server audio per il live coding | 59
- 3.6 Max/MSP come server audio per il live coding | 63

Capitolo IV: Implementazioni di tecniche compositive e processamenti elettroacustici
- 4.1 Tecniche di sintesi e processamento | 67
- 4.2 Tecniche stocastiche | 85
- 4.3 Tecniche ritmiche e contrappuntistiche | 95

Conclusioni | 100

Bibliografia | 102

Introduzione

Il live coding è una forma di improvvisazione che consiste nella scrittura di algoritmi in tempo reale.
In sostanziale continuità con la tradizione della ricerca nel campo della musica algoritmica, la pratica del live coding è emersa negli ultimi venti anni grazie al lavoro di programmatori informatici, compositori e performer. La disponibilità in forma gratuita e libera degli strumenti di programmazione per il live coding ha permesso lo sviluppo di una sottocultura in grado di generare un contesto socio-culturale specifico.
Benché il live coding sia da qualche anno oggetto di studi accademici, le ricerche sugli aspetti musicologici e tecnologici sono ancora all'inizio. A partire da quanto sinora esplorato dalla letteratura, questi tesi ha lo scopo di indagare le implicazioni compositive e performative proprie del live coding, tentando di tracciare un possibile percorso di sviluppo di un ambiente per l'improvvisazione algoritmica. All'analisi storica e musicologica è affiancato dunque uno studio delle attuali architetture informatiche, al quale segue la costruzione di un sistema specifico in grado di applicare tecniche compositive tradizionali e processamenti elettroacustici in tempo reale.
Il primo capitolo si occupa di inquadrare il contesto storico-sociologico che ha portato allo sviluppo delle attuali forme di improvvisazione algoritmica, dapprima descrivendo la nascita della cultura degli Algorave (sez. 1.1), dunque ricostruendo le influenze ideative provenienti dai precursori della *computer music* (sez. 1.2); a partire da tali prospettive si è delineata una breve storia del live coding (sez. 1.3), sia sul piano tecnologico, sia su quello sociologico, sino a giungere all'attuale conformazione associazionistica e non-centralizzata, i cui aspetti etnografici e discografici sono discussi nella sez. 1.4. È stato proficuo indagare quali fossero i principi politici e ideologici propri della comunità dei *live coders* (sez. 1.5), con particolare enfasi sulle modalità di divulgazione e condivisione della conoscenza proprie dell'Open Source.
L'indagine sulle proprietà musicologiche intrinseche dell'improvvisazione algoritmica è oggetto di studio nel secondo capitolo. In particolare sono state prese in esame le implicazioni derivanti dalle specificità tecniche di questa

pratica sul piano temporale di una performance (sez. 2.1) e le relazioni tra scrittura musicale e scrittura algoritmica (sez. 2.2); conseguentemente sono state analizzate le potenzialità inscritte nella codifica del linguaggio musicale mediante i linguaggi di programmazione (sez. 2.3). Una riflessione sulla multimedialità del live coding e i suoi effetti percettivi sul pubblico è presente nella sez. 2.4.

Il terzo capitolo ha lo scopo di comparare gli attuali linguaggi di programmazione musicale in tempo reale (sez. 3.1) e riflettere sui parametri determinanti nella scelta di un design specifico (sez. 3.2). In luce di quanto rilevato, si è proposta una possibile architettura composta da molteplici linguaggi utilizzati in sinergia (TidalCycles, Haskell, SuperCollider, Max/MSP), il cui funzionamento è descritto dalla sez. 3.3 alla 3.6. A una spiegazione teoretica sono affiancati esempi scritti in tali linguaggi.

Una dimostrazione delle possibilità inscritte nel sistema così definito costituisce il quarto e ultimo capitolo. La sez. 4.1 raccoglie una serie di algoritmi di sintesi e processamento scritti in linguaggio SuperCollider negli ultimi tre anni, mentre una messa in pratica delle tecniche stocastiche è illustrata nella sez. 4.2. La tesi si conclude con l'implementazione di alcune tecniche ritmico-contrappuntistiche in linguaggio Haskell (sez. 4.3). Quanto mostrato nell'ultimo capitolo rappresenta un esempio di personalizzazione di un ambiente di live coding, le cui funzionalità possono essere definite dal compositore elettroacustico. Piuttosto che una trattazione esaustiva del campo di possibilità del live coding, si tratta dunque di fornire degli strumenti, affinché ciascun compositore possa intraprendere il live coding con la prospettiva di poter costruire un ambiente specifico e direzionato verso le proprie esigenze espressive.

Tutti gli algoritmi mostrati nell'ultimo capitolo sono presenti sul mio profilo GitHub (Olbos, 2020) e possono essere utilizzati e modificati liberamente.

CAPITOLO I: Contesto storico-sociologico

1.1 Definizione di Algorave

Il termine "Algorave" è un neologismo sincratico composto da "algorithmic" e "rave". Esso designa un evento musicale in cui la musica viene generata in tempo reale attraverso linguaggi di programmazione. La presenza del sostantivo "rave" specifica che un Algorave, a differenza di una più generica "live coding performance", fa riferimento al concetto di festa libera e auto-organizzata in cui i partecipanti ballano musica elettronica.
Il Criminal Justice and Public Order Act (1994), decreto emesso dal parlamento inglese per rendere illegali i *rave parties*, definisce un rave come un raduno di almeno venti persone in cui "la musica includa suoni interamente o prevalentemente caratterizzati dall'emissione di una successione di impulsi ripetitivi".
Ironicamente, il sito algorave.com descrive l'Algorave come "costituito da suoni interamente o prevalentemente caratterizzati dall'emissione di una successione di *condizionali* ripetitivi", laddove il termine "condizionali" indica una funzionalità presente in molti linguaggi di programmazione.
Sebbene le connessioni con la rave culture siano esplicite e imprescindibili per l'Algorave, l'estetica sonora non è necessariamente connessa alla musica techno. Durante un festival di Algorave si può osservare, oltre alla presenza di moltissimi generi di club music, un'estesa varietà stilistica che può comprendere *ambient music*, *extreme computer music*, *noise*, musica elettroacustica, *chiptune*, e talvolta perfino sincretismi tra musica algoritmica e musiche a essa lontanissime, come ad esempio le musiche folkloriche. Difatti la relazione tra rave music e Algorave è legata a similarità ideologiche, filosofiche e di contesto socio-culturale, più che a una sostanziale convergenza estetica.
Inoltre Algorave non indica un genere musicale e non rappresenta di per sé un descrittore estetico, benché l'utilizzo di linguaggi musicali di programmazione in tempo reale costituisca una base metodologica condivisa in grado di generare coordinate stilistiche comuni.

Se comparati a strumenti musicali elettronici più convenzionali, quali ad esempio samplers, sintetizzatori e groove boxes, i linguaggi di programmazione permettono maggiore personalizzazione e possono perciò dare luogo a una maggiore differenziazione estetica. Tuttavia le specificità dell'approccio algoritmico in tempo reale – tra cui l'uso di variabili condizionali e stocastiche, la facilità nell'usare strutture metriche complesse e un generale approccio talvolta puramente matematico ai parametri del suono – determinano delle affinità potenzialmente sufficienti a far emergere una dimensione estetica condivisa e specifica di questa modalità esecutiva.

Algorave indica allo stesso tempo l'evento e la comunità che lo organizza e ne fa esperienza.
L'utente GitHub "*yaxu*", riconducibile ad Alex McLean (McLean, n.d.), ha scritto delle linee guida per l'organizzazione di un Algorave, nelle quali è specificato che "Algorave non è un brand protetto o un franchise", quanto piuttosto "una community" (yaxu, 2017).
Il testo auspica che la cultura Algorave sia interpretata come libera, non gerarchica, non istituzionalizzata, tollerante delle diversità, locale e globale al contempo. Prescrive inoltre che la maggioranza delle performance in un Algorave siano basate sulla generazione di musica algoritmica in tempo reale, e che i processi algoritmici, generalmente rappresentati come testi, siano resi visibili al pubblico tramite proiezioni.
Il primo evento musicale a essere definito esplicitamente come Algorave ebbe luogo a Londra il 17 marzo 2012 (Resident Advisor, 2012). Il web magazine Wired (Cheshire, 2013) riporta che tale espressione fu scherzosamente coniata da Alex McLean e Nick Collins nel tardo 2011. Il termine, forse per la sua icasticità, si è diffuso notevolmente ed è divenuto il referente di un insieme di pratiche performative al confine tra musica algoritmica, *club music,* hacking e *geek culture.*
L'appello allo sviluppo di un network globale di Algorave è stato diffuso da TOPLAP (Temporal Organization for the Purity of Live Algorithm Programming), un collettivo nato nel 2004 per l'organizzazione, la

promozione e la didattica dell'improvvisazione musicale mediante programmazione algoritmica in tempo reale.
In pochi anni il concetto di Algorave si è diffuso in tutto il mondo e al momento il sito algorave.com elenca 275 eventi svoltisi in 29 nazioni.
In molti casi gli eventi sono trasmessi in streaming sul web. Sono frequentemente organizzati eventi del tutto virtuali, talvolta all'interno di ambienti tridimensionali in realtà virtuale e/o aumentata, nei quali gli utenti possono esplorare uno scenario mentre prendono parte alla manifestazione.
L'aspetto virtuale è formalmente ed esteticamente compatibile con le pratiche dell'Algorave e viene considerato un elemento strutturale dell'espressione di tale sottocultura. Questa caratteristica ha contribuito alla diffusione su scala globale del live coding, in parte grazie a un festival annuale in streaming, nel quale musicisti da tutto il mondo trasmettono a turno le proprie performance in un flusso ininterrotto.
Di anno in anno il festival è cresciuto per durata e numeri, fino ad arrivare, nell'edizione 2020 chiamata Eulerroom Equinox (CLiC, 2020), a includere circa centottanta esibizioni nell'arco di quattro giorni consecutivi.

Negli ultimi anni, l'Algorave e più in generale l'improvvisazione in live coding, sono divenuti oggetto di studio da parte della comunità accademica, che ne ha analizzato gli aspetti tecnici, estetici, filosofici e sociologici.
Alcuni creatori dei linguaggi di programmazione per il live coding, tra cui Alex McLean, Thor Magnusson e Julian Rohruhber, sono ricercatori accademici associati a istituzioni e hanno documentato lo sviluppo delle proprie ricerche (si vedano ad esempio Rohrhuber, de Campo, Wiesers, 2005, McLean, 2011, McLean, Dean, 2018b, Magnusson, 2011).
L'interesse verso i potenziali sviluppi tecnologici, filosofici e artistici del live coding ha portato nel 2015 alla creazione dell'International Conference on Live Coding, riunione annuale che oltre a comprendere conferenze sugli aspetti teorici e tecnici, include anche concerti e l'organizzazione di un Algorave (ICLC, 2020).

1.2 I precursori del live coding

Al fine di delineare una storia del live coding, è necessario anzitutto inquadrare i paradigmi di pensiero che hanno permesso l'emersione delle condizioni storiche e materiali per lo sviluppo di questa pratica musicale. In tal senso è rilevante rintracciare brevemente quali concetti fondamentali per il live coding siano stati introdotti dai precursori della computer music e della musica algoritmica.

Il live coding, infatti, tanto sul piano tecnologico quanto su quello del pensiero compositivo-improvvisativo, si pone in una sostanziale linea di continuità con la ricerca sulla musica elettronica portata avanti da studi radiofonici, università e centri di ricerca industriali a partire dagli anni '50 del XX secolo.

Benché dunque si possa parlare di ricerca specificamente indirizzata all'improvvisazione mediante linguaggi di programmazione musicale in tempo reale soltanto a partire dagli anni '80, una parte piuttosto consistente delle premesse tecniche e concettuali è racchiusa negli scritti e nelle opere di compositori-ricercatori quali Pierre Barbaud, Lejaren Hiller, Herbert Brün, James Tenney, Gottfried Michael Koenig, Iannis Xenakis, Laurie Spiegel e Pietro Grossi.

Nel 1959 Pierre Barbaud fondò il Groupe de musique algorithmique de Paris (GMAP), usando esplicitamente il termine "musica algoritmica" per la prima volta. Barbaud voleva superare il concetto romantico di "ispirazione" del compositore applicando un approccio razionale mediante l'uso del calcolatore elettronico.

Il GMAP comprendeva anche Janine Charbonnier e Roger Blanchard, coi quali Barbaud compose delle opere collettive, tra cui *Souvenirs entomologiques* (1959) e *Factorielle 7* (1960). Quest'ultima era costituita da 5040 variazioni (5040 = 7!) di una serie di altezze ottenute mediante un algoritmo (Andreatta, 2013).

Dal momento che il GMAP non era in possesso di un dispositivo di conversione digitale-analogica, i parametri delle composizioni erano ottenuti

da calcoli del computer ed erano successivamente trascritti sotto forma di spartito per strumenti acustici. Si osserva l'interesse del gruppo verso la sperimentazione delle possibilità combinatorie dei parametri musicali come principio di automatizzazione del processo compositivo (Robert, 2018).

Parallelamente alle ricerche del GMAP, alla University of Illinois at Urbana-Champaign, Lejaren Hiller e Leonard Issacson composero *Illiac Suite* (1957), un quartetto d'archi la cui partitura era stata programmata mediante la generazione di numeri interi casuali secondo il metodo di Monte Carlo. Un elenco di regole estratte da studi sulla percezione e da tecniche compositive tradizionali veniva applicato all'output del generatore casuale, in modo da creare selettivamente sequenze di note musicali.
Nello sviluppo delle sue prime composizioni, Hiller fu fortemente influenzato dalla teoria dell'informazione, per la quale i dati erano analizzati in termini di possibilità comunicative; perciò Hiller concepiva una sequenza totalmente casuale come la più ricca in termini informazionali – senza tuttavia che i concetti di informazione e comunicazione fossero relazionati alla produzione di significato.
Egli dunque intravedeva la possibilità di concepire la composizione come "un compromesso tra caos e monotonia" (Hiller, 1959, p.110), nel quale la selezione di elementi ridondanti da parte del compositore costituisce l'aspetto creativo in grado di dare senso.
Benché le posizioni di Hiller fossero state accolte in maniera ostile da buona parte della comunità accademica, nel 1958 fu in grado di fondare alla University of Illinois l'Experimental Music Studio, un centro di ricerca che vide la presenza di altri compositori quali Herbert Brün, James Tenney, Kenneth Gaburo e Salvatore Martirano (Battisti, n.d.).
Per molti anni Hiller proseguirà la propria ricerca sulla composizione algoritmica, pubblicando dettagliate descrizioni degli algoritmi implementati (Hiller, 1981). L'importanza storica di Hiller per il live coding è quella di aver introdotto un paradigma di pensiero per l'uso di sequenze pseudo-casuali controllate da un insieme di regole predefinite dal compositore.

Anche James Tenney lavorò in questa direzione, arrivando a definire la sua musica "stocastica" secondo un'accezione personale: "[la musica stocastica] richiede un processo casuale vincolato. Sono i vincoli a dare la forma al pezzo." (Dennehy, 2008, p.84).
Affiancato a uno studio approfondito della percezione degli eventi musicali nel tempo (Belet, 2008), il processo compositivo di Tenney si concentra sull'aspetto macro-formale e timbrico-morfologico, delegando le variazioni parametriche micro-formali a procedure pseudo-casuali e feedback statistici (Polansky et al., 2011).
La possibilità di gestire i micro-movimenti con processi automatici non-deterministici rappresenta oggi uno degli elementi più importanti per la gestione di un'improvvisazione in live coding, poiché consente all'esecutore di concentrarsi su livelli temporali più ampi, pur mantenendo movimento e complessità micro-formale.

Herbert Brün, autore di rivoluzionari software di sintesi del suono programmati in linguaggio FORTRAN, s'interessò alla composizione algoritmica con l'intento di produrre nuove forme sonore. Influenzato, così come Hiller, dalla teoria dell'informazione, creò il concetto di *anticomunicazione*:

> «L'anticomunicazione è più facilmente osservabile […] se frammenti ben noti di un sistema linguistico sono composti in un ambiente contestuale in cui, per quanto ci provino, non riescono a significare ciò che hanno sempre significato, e invece, iniziano a mostrare tracce d'integrazione in un altro sistema linguistico, nel quale […] un giorno potrebbero significare quello che non hanno mai significato ed essere nuovamente comunicativi» (2004 in Kowalkowski, 2008, p. 238).

Per Brün dunque l'impiego di un linguaggio musicale già codificato e stratificato dalla tradizione non rappresenta un'opportunità comunicativa, ma gli elementi sintattici in esso contenuti possono essere contestualizzati attraverso nuovi sistemi compositivi.

Il pensiero algoritmico di Brün è un pensiero della costante riconfigurazione del passato alla ricerca dell'inaspettato. Il processo compositivo, dunque, non parte da premesse deterministiche, ma è costruito col fine di generare un campo di possibilità aperto.
Per Brün la ricerca di nuovi linguaggi è un aspetto politico intrinseco al ruolo del compositore (Smith e Smith, 1979).
Il tema dell'imprevedibilità dell'output di un algoritmo è un argomento di rilievo nel contesto del live coding, al punto che Cox (2015) ha teorizzato la possibilità di concepire l'incertezza improvvisativa come uno strumento politico in grado di rendere all'esecutore una comprensione profonda dei meccanismi ontologico-epistemologici della cultura computazionale. In quest'ottica il paradigma compositivo di Brün costituisce uno strumento concettuale proficuo per l'ideazione e il design di un proprio sistema di composizione o improvvisazione algoritmica.

Gottfried Michael Koenig lavorò alla WDR di Colonia tra il 1954 e il 1964; successivamente si trasferì in Olanda e divenne direttore dell'Institute of Sonology. In merito alla costruzione formale propose, in opposizione al metodo analitico delle forme tradizionali, la costruzione di modelli algoritmici costituiti da una serie di regole matematiche definite dal compositore.
Il design del modello matematico è continuamente messo alla prova dall'esperienza percettiva e corretto al fine di raggiungere equilibri coerenti con la cognizione umana. Il metodo compositivo consiste dunque in una costante costruzione di prototipi e varianti di un algoritmo. Con questo proposito, Koenig categorizzò una serie di strategie possibili per il design di un modello compositivo: interpolazione, estrapolazione, principio cronologico-associativo, struttura a blocchi (Koenig, 2018).
Una trattazione approfondita di tali strategie esula dagli scopi di questa tesi; tuttavia vale la pena denotare la centralità attribuita al processo matematico come principio generatore della forma musicale.
I processi matematici nella musica di Koenig si basano, come precedentemente osservato in altri compositori menzionati, sul controllo e la limitazione di segnali casuali. Egli sviluppò numerose tecniche di controllo

mediante l'uso di serie, gruppi, rapporti di peso statistico dei valori e maschere di tendenza.
In molti linguaggi di live coding contemporanei è possibile trovare funzioni che producono comportamenti analoghi a quelli implementati da Koenig.

Lo stesso interesse sulle strutture casuali, o per meglio dire stocastiche, seppur declinate con tutt'altra concezione, si riscontra nelle opere e negli scritti di Iannis Xenakis. Egli certamente non necessita di presentazioni biografiche; i suoi innumerevoli contributi alla musica algoritmica ed elettroacustica sono stati ampiamente studiati (si veda ad esempio Solomos ed., 2001).
Xenakis infatti ha sostenuto un cambio di paradigma dal determinismo causale classico al probabilismo statistico, applicando i principi della stocastica ai parametri sonori. Ancor prima di avere accesso a un calcolatore elettronico, elaborò tecniche basate su leggi di distribuzione, catene di Markov, spazi vettoriali e sieves (Xenakis, 1992). L'influenza di un tale rigore matematico applicato a una dimensione estetica e poetica personale costituisce una fonte d'ispirazione indiscutibile per la comunità del live coding.
Molte delle sue tecniche sono state successivamente implementate in linguaggi di programmazione in tempo reale – in particolare si veda l'applicazione su SuperCollider della Dynamic Stochastic Synthesis da parte di Luque (2009).
Nel capitolo 4 di questa tesi si trovano alcune applicazioni degli oscillatori stocastici da me programmate per TidalCycles, oltre a un'implementazione in Haskell per la scrittura in tempo reale di strutture ritmiche ottenute mediante la tecnica dei sieves.

Laurie Spiegel ha lavorato ai Bell Laboratories con Max Mathews, dove ha perfezionato il sistema GROOVE, un programma per il controllo algoritmico di sintetizzatori analogici. È inoltre la sviluppatrice di Music Mouse, il primo software di composizione algoritmica a fare uso di un mouse.

La sua ricerca sul design di strumenti e interfacce musicali elettroniche ha influenzato intere generazioni di sviluppatori; TidalCycles, il software di live coding impiegato nella parte pratica di questa tesi, è stato progettato da Alex McLean ispirandosi alle ricerche di Spiegel in merito ai pattern musicali (McLean e Wiggins, 2009, TidalCycles n.d. a).
Secondo Spiegel un pattern è una sequenza di informazioni che può essere soggetta a trasformazioni; può essere dunque reiterabile o ricombinabile attraverso una serie di operazioni quali trasposizione, inversione, rotazione, sfasamento, riscalamento, interpolazione, estrapolazione, frammentazione, sostituzione interparametrica, combinazione, sequenziamento e ripetizione (Spiegel, 1981). Queste tecniche consentono di applicare, su vari livelli temporali e a qualunque parametro, trasformazioni contrappuntistiche, il cui controllo in tempo reale, se gestito tramite un linguaggio opportunamente programmato, risulta agevole ed economico in termini di quantità di operazioni da svolgere per ottenere elaborazioni complesse dei materiali.

Pietro Grossi è stato un pioniere della computer music italiana. Al CNR di Pisa sperimentò la possibilità di modificare in tempo reale altezze e durate di un sintetizzatore digitale controllato mediante il software DCMP da lui sviluppato. Successivamente ottenne i fondi per la costruzione di un sintetizzatore più potente, il TAU2, per il quale scrisse programmi.
Questi sistemi erano in grado di calcolare in tempo reale tutte le operazioni, permettendo al programmatore di modificare il codice durante l'esecuzione senza interrompere il flusso audio (Mori, 2010). Grossi cominciò a concepire la possibilità di una musica automatica e potenzialmente infinita e priva di ripetizioni. Mori (2015b) ha osservato come le intuizioni di Grossi abbiano largamente anticipato lo sviluppo del live coding.
Grossi inoltre nel 1970 fu il primo a realizzare un esperimento di telematica musicale, trasmettendo segnali tra Rimini e Pisa (Giomi, 2017); questa sperimentazione precede di molti anni uno dei caratteri principali della cultura del live coding, vale a dire lo streaming in tempo reale di un'esibizione musicale. Successivamente Grossi immaginò la possibilità di una *Home Art,* un'arte "creata da e per se stessi", "estemporanea", "effimera" (Grossi, n.d.),

prodotta con facilità mediante i personal computer e potenzialmente condivisibile in maniera libera tra gli utenti. Questa speculazione si muove in una direzione analoga alla concezione open source del live coding contemporaneo.

1.3 Breve storia del live coding

L'avvento dei "microcomputers" negli anni '80 permise lo sviluppo materiale dei primi esperimenti di computer music dal vivo: grazie alla portabilità, il costo relativamente basso e la possibilità di affrancarsi dai grandi computer situati nei centri di ricerca universitari, ricercatori e amatori poterono cominciare a sperimentare in maniera indipendente tecniche per la programmazione musicale. La maggioranza dei linguaggi di programmazione dell'epoca erano compilati e non consentivano il controllo in tempo reale.
I primi tentativi di live coding vero e proprio furono realizzati in FORTH, un linguaggio di programmazione imperativo che permetteva l'esecuzione interattiva.

Tra il 1978 e il 1983, nella Bay area di San Francisco un gruppo di docenti e studenti del Mills College, tra cui John Bischoff, Jim Horton, Tim Perkis, David Behrman, Paul DeMarinis e Rich Gold, fondarono The League of Automatic Music Composers, un collettivo per la sperimentazione di computer music collaborativa automatica (Boutwell, 2009). Ispirato dalle utopie tecnologiche della Silicon Valley, dagli esperimenti di musica cibernetica di David Tudor e dalla *free improvisation*, il gruppo costruì un network di numerosi microcomputer KIM tra loro interconnessi, in grado di performare musica 8 bit. Questo sistema doveva essere programmato prima dell'esibizione e non c'erano possibilità di modifica durante l'esecuzione. Tuttavia i microcomputer KIM trasmettevano dati tra di loro, mutando il proprio comportamento secondo circuiti di feedback programmati.
Dalla League of Automatic Music Composers nacque nel 1986 il gruppo "The Hub", formato da Bischoff, Horton e Perkis; il loro scopo era inventare

nuove tecnologie per il live electronics con i computer. Le performance di The Hub avevano un approccio più installativo che performativo. Infatti, dal momento che non era ancora possibile modificare il codice in tempo reale, i computer erano disposti in uno spazio espositivo che il pubblico poteva esplorare liberamente (Bischoff e Brown, n.d.).
Nel 1991 David P. Anderson e Ron Kuivila crearono il sistema di programmazione Formula (Forth Musical Language), un'estensione del linguaggio FORTH pensata per la composizione algoritmica e l'interazione in tempo reale (Anderson e Kuivila, 1991). Formula non era in grado di modificare il timbro dei suoni, ma poteva gestire altezze, durate e ampiezze organizzando le note in gruppi. Il codice poteva essere modificato in qualunque momento durante l'esecuzione. La prima performance di live coding documentata è un'esibizione di Ron Kuivila allo STEIM di Amsterdam nel 1985, in cui eseguì la sua composizione algoritmica *Water Surface*. Il concerto si concluse dopo mezz'ora con un crash del sistema operativo.

Sorensen (2005) ha osservato che durante gli anni '90, escludendo tentativi di patching in tempo reale con Max/MSP, non vi sono state particolari innovazioni nel campo del live coding. Non è stata infatti rintracciata alcuna documentazione che attesti attività di questo genere nell'ultimo decennio del Novecento.
L'interesse per il coding musicale in tempo reale ha visto il suo sviluppo nei primi anni del 2000. Rispetto agli esperimenti in FORTH degli anni '80, in questo periodo la disponibilità di potenza computazionale era notevolmente aumentata. Inoltre molti linguaggi di programmazione stavano acquisendo maggiore popolarità e il bacino di utenti aveva visto un vertiginoso incremento grazie alla progressiva scolarizzazione digitale, alla commercializzazione di massa dei personal computer e alla diffusione del World Wide Web. Estremamente rilevante per il live coding fu la pubblicazione nel 1996 di SuperCollider, un linguaggio di programmazione orientato agli oggetti, sviluppato da James McCartney ispirandosi al linguaggio Smalltalk (Wilson et al. eds., 2011). SuperCollider è un ambiente

dinamico per la musica algoritmica estremamente versatile, in grado di contenere numerosi livelli di astrazione per il controllo dei parametri musicali.
Nel 2000 McCartney e Julian Rohrhuber lavorarono a un'installazione chiamata *remote control lounge* presso il Telenautik di Amburgo che permetteva al pubblico di "influenzare il suono" mediante SuperCollider da remoto. Rohrhuber era interessato alla modifica del codice in tempo reale e a tal fine scrisse JITLIB (Just In Time Library), una libreria di astrazioni per SuperCollider in grado di facilitare ciò che egli ha definito *just in time programming* (Rohrhuber et al., 2005).
Un sostanziale salto in avanti per lo sviluppo del live coding avvenne nel 2002, quando McCartney decise di rilasciare la terza versione di SuperCollider in forma open source. L'accessibilità e la possibilità di modificare il codice sorgente del programma costituì un sostanziale punto a favore per l'adozione di SuperCollider da parte di numerosi utenti interessati al live coding. Questa nuova versione divise il programma in tre parti fondamentali: scsynth (il server audio), sclang (il linguaggio di programmazione) e scide (l'editor comprendente l'interfaccia utente). Scsynth e sclang comunicano tra loro con messaggi Open Sound Control (OSC); questa separazione tra il server audio e il linguaggio rende possibile il controllo del primo mediante linguaggi di programmazione differenti da sclang.
Una parte sostanziale dei linguaggi di live coding tutt'ora impiegati, tra cui TidalCycles, FoxDot e SonicPi, fa uso del server audio di SuperCollider.

Nel 2000 Alex McLean e Adrian Ward fondarono Slub, un duo inglese di improvvisazione in live coding – successivamente divenuto un trio con l'arrivo di Dave Griffiths – che faceva uso di stilemi, ritmi e timbri provenienti dalla club music. Ognuno dei componenti del gruppo suonava con un proprio sistema algoritmico ed era sincronizzato agli altri tramite protocollo TCP/IP (Collins et al., 2003).
Ward impiegava il suo ambiente di programmazione Pure Events scritto in linguaggio REALbasic, col quale era in grado di associare a ogni evento musicale un frammento di codice richiamabile in tempo reale. McLean aveva creato feedback.pl, un sistema scritto in Perl che variava il proprio codice in

maniera autonoma, innescando dei feedback tra ciò che era scritto dall'esecutore e ciò che variava per opera del sistema stesso (Ward et al., 2004).
Gli Slub furono i primi a proiettare durante le esibizioni le schermate dei computer contenenti i codici.
Nel 2003 Ge Wang e Perry Cook pubblicarono ChucK, un linguaggio di programmazione open source per l'*on-the-fly music programming,* un'espressione che a quel tempo era usata come sinonimo di live coding (Wang e Cook, 2004). ChucK introdusse una metodologia di controllo temporale estremamente precisa, che consentiva di gestire gli eventi alla velocità della frequenza di campionamento (Wang et al., 2015). Analogamente a SuperCollider, esso dispone di *unit generators*, operatori DSP che possono essere combinati per ottenere tecniche di sintesi complesse. A distanza di diciassette anni dalla prima versione, ChucK vanta una vasta comunità di utenti.

Ben presto, grazie alla diffusione di informazioni sul web, l'idea dell'improvvisazione musicale algoritmica in tempo reale cominciò a diffondersi. Benché in quegli anni fosse già ampiamente sviluppato il mercato delle *Digital Audio Workstation*, alcuni tra docenti, studenti, programmatori e amatori preferirono avventurarsi nella sperimentazione algoritmica.
Il 14 febbraio 2004, a seguito di un'improvvisazione collettiva con JITLIB presso il Changing Grammars Symposium alla Hamburg Art Academy (HFBK-Hamburg, 2004), un gruppo di partecipanti fondò TOPLAP, un'organizzazione per la promozione del live coding.
La nascita di TOPLAP sancì la volontà di creare una comunità globale di live coders attraverso l'organizzazione di eventi e lo scambio di informazioni, tutorial, frammenti di codice, file audio e supporti multimediali. Da questo momento in poi, si osserva un'incrementale proliferazione di nuovi linguaggi per il live coding e un aumento quasi esponenziale, di anno in anno, del numero di concerti.

Già nel 2007 TOPLAP ebbe un network sufficientemente sviluppato affinché fosse possibile la formazione di una laptop orchestra composta da quindici live coders diretti da Ge Wang presso la Princeton University (PLOrk, n.d.).

Riassumere gli anni successivi al 2007 risulta complesso a causa della densità di informazioni presenti. Inoltre la distanza temporale non è sufficiente per permettere una storicizzazione degli eventi.
Come già menzionato, la nascita di Algorave nel 2012 ha contribuito alla diffusione capillare dell'improvvisazione musicale algoritmica e ha creato un contesto socio-culturale specifico di tale fenomeno.
Non tutte le esperienze di live coding contemporaneo rientrano nell'Algorave, tuttavia esso rappresenta, assieme a TOPLAP, un punto di riferimento comunicativo per la comunità globale.
La prossima sezione si occupa di analizzare le caratteristiche comunitarie del live coding.

1.4 Il network globale del live coding

La sottocultura del live coding è intrinsecamente legata alla complementarità tra comunità globale e locale. La si può considerare una sottocultura nativa del web, in quanto si è sviluppata attraverso mailing list, forum, chat e streaming. Nondimeno, l'elemento aggregativo locale costituisce una componente estremamente significativa.
Virtualità e fisicità rappresentano, tanto nella pratica performativa, quanto sul piano sociale, un binomio imprescindibile per la comprensione del fenomeno del live coding contemporaneo. Difatti, sebbene il live coding di per sé sia semplicemente un metodo per l'improvvisazione musicale, negli ultimi venti anni attorno a questa pratica si è sviluppata una cultura associazionistica che comprende numerose decine di gruppi locali e migliaia di utenti online. Le cause di una tale configurazione tecno-sociale risalgono probabilmente a una filosofia di condivisione delle conoscenze e dei mezzi tecnologici specifica dell'hacking culture e dell'open source (per un approfondimento su questo

aspetto, si veda sez. 1.5), ma anche all'orientamento comunitario e scarsamente personalistico caratteristico di alcune sottoculture evolutesi a partire dalla rave music. Vale la pena soffermarsi sui mezzi di trasmissione del live coding e sulla sua attuale configurazione sociale nel mondo.

L'uso delle piattaforme web da parte della comunità dei live coders assolve cinque funzioni essenziali: la diffusione di informazioni tecniche, la logistica relativa all'organizzazione di eventi, la collaborazione a distanza, la socialità e la trasmissione di contenuti artistici.
La musica algoritmica improvvisata richiede una vasta area di competenze che interseca teoria musicale e programmazione informatica. Durante lo studio di un determinato linguaggio di programmazione per il live coding, succede frequentemente di non riuscire a trasporre un'idea matematica o musicale in forma di codice; scarsità di documentazione relativa al linguaggio e difficoltà nel comprendere la sintassi e i concetti della programmazione si verificano piuttosto spesso. La possibilità di consultare un forum attivamente frequentato da altri utenti permette di accelerare la curva di apprendimento.
A tal scopo, nel 2015 TOPLAP fondò un canale su Slack, un'applicazione di messaggistica istantanea finalizzata al lavoro collettivo (Toplap, 2015). Il canale TOPLAP conteneva a sua volta decine di sotto-canali, ciascuno specifico di un determinato linguaggio per il live coding. Successivamente, a causa dell'alto numero di utenti (più di 1200), la comunità si è spostata su un'altra applicazione chiamata Lurk (n.d.). Oltre a queste chat, vi sono innumerevoli forum nei quali è possibile chiedere aiuto di fronte a difficoltà nella programmazione. Vi sono anche gruppi di messaggistica istantanea sull'app Telegram in diverse lingue, nei quali si discute dell'organizzazione di workshop ed eventi locali. Queste chat sono un luogo di socialità importante per l'aggregazione della comunità internazionale.
Un'altra funzione svolta dalle tecnologie web è la condivisione di codice. Dal momento che tutte le informazioni musicali sono rappresentate sotto forma di codici, è possibile scambiare e condividere l'intero processo creativo in pochi kilobyte. In questo modo artisti situati in diversi continenti possono collaborare in maniera flessibile e immediata, sviluppando astrazioni musicali

collaborative; lo scambio di codice nato per la progettazione informatica è diventato uno strumento creativo dalle notevoli potenzialità.
Alcuni live coders sono soliti condividere i documenti contenenti algoritmi, routine e progetti in corso d'opera su GitHub, una piattaforma per la condivisione di codici usata da milioni di utenti nella programmazione informatica di ogni genere (si vedano ad es. Guiot, n.d. e Hodnick, 2016).
Tutti gli algoritmi scritti per questa tesi sono disponibili e scaricabili dal mio profilo GitHub (Olbos, n.d.).
La condivisione di informazioni tecniche è attuata anche sfruttando le potenzialità pedagogiche dei mezzi audiovisivi. Su Youtube si possono trovare centinaia di video didattici per il live coding. Inoltre si è recentemente discusso della possibilità di organizzare dei corsi online, seguendo il paradigma nascente dell'apprendimento digitale sperimentato da piattaforme come Kadenze e Khan Academy.
Alex McLean ha realizzato dei video-corsi per l'apprendimento del software TidalCycles, ai quali è possibile accedere con una *club membership* il cui costo è deciso dall'utente in base alla propria disponibilità economica.
Altre forme di diffusione tramite il web si riscontrano su social network come Facebook, Instagram e Reddit, utilizzati per condividere brevi frammenti audiovisivi e per trasmettere informazioni logistiche circa l'organizzazione di eventi online e dal vivo.
Un aspetto estremamente peculiare della comunità del live coding, specie per quanto riguarda l'Algorave, è l'utilizzo delle tecnologie di streaming in tempo reale. Sebbene in questi anni la trasmissione di contenuti audiovisivi tramite piattaforme di streaming quali Youtube e Twitch sia aumentata vertiginosamente, dando vita a una vera e propria *streaming culture*, le modalità con cui Algorave fa uso di queste tecnologie presentano elementi di originalità. Sono infatti regolarmente organizzati festival online ai quali i live coders possono liberamente prendere parte, trasmettendo la propria esibizione.
La compresenza di eventi fisici e virtuali simboleggia la complementarità dei due aspetti nella pratica del live coding. Che si tratti di un concerto dal vivo trasmesso anche virtualmente, o di un'improvvisazione svolta in casa e

realizzata unicamente per il pubblico online, amatori ed esperti hanno egual diritto a occupare fasce orarie che vanno solitamente dai venti ai trenta minuti. Questa sovrapposizione di contesti e di esperienza dei programmatori è concepita al fine di dare luogo a manifestazioni inclusive e ricche di diversità.

A fronte di una tale proliferazione di contenuti virtuali, si osserva la nascita di gruppi locali dedicati a workshop, co-working ed esibizioni dal vivo.
TOPLAP promuove forme di organizzazione non-centralizzate attraverso i Toplap Nodes, associazioni locali autogestite. Al momento esistono trenta Nodes presenti in Argentina, Belgio, Brasile, Canada, Cile, Cina, Colombia, Danimarca, Ecuador, Francia, Germania, Israele, Italia, Giappone, Messico, Olanda, Perù, Regno Unito, Spagna, Taiwan e USA.
Il Node *toplap Italia* è stato fondato nel 2019 a seguito di un workshop di Alexandra Cárdenas presso Tempo Reale (Toplap Italia, n.d.). La creazione di un gruppo nazionale ha facilitato lo scambio di informazioni e ha permesso l'organizzazione di concerti e workshop. Dopo essere entrato in contatto con un gruppo di programmatori che avevano preso parte al workshop di Cárdenas, assieme al collettivo AMEN ho organizzato il primo algorave a Roma, svoltosi allo Spin Time Labs il 28 marzo 2019 (flyer, 2019). Quest'esperienza è stata un'occasione per conoscere live coders provenienti da diverse zone d'Italia.
Altri algorave si sono svolti in Italia negli ultimi due anni grazie a organizzazioni quali Phase (2018) e Umanesimo Artificiale (Primo, 2019).
Mori (2020) ha svolto una ricerca etnografica sulla comunità di live coding inglese. Attraverso esperienze dirette, *netnography* e *autoethnography,* ha analizzato quella che probabilmente è la comunità locale più attiva al mondo per numero di algorave organizzati, documentando conferenze e concerti.
Cárdenas (2018) si è occupata di ricostruire gli sviluppi del live coding in Messico e in India. Comparando le due situazioni, ha osservato in Messico una comunità solida e molto partecipata – in continuo contatto con le molte altre comunità locali sudamericane tramite il collettivo CLiC (n.d.) – mentre,

per quanto riguarda l'India, ha attestato l'emergenza di primi tentativi per la costruzione di un gruppo locale.

Prima di concludere questa sezione, è rilevante osservare la presenza discografica del live coding. Sebbene, per ovvie ragioni estetiche, la musica improvvisativa algoritmica non costituisca un campo d'interesse per il mercato musicale *mainstream*, essa sta trovando spazio in molte piccole etichette discografiche. Nonostante un volume economico pressoché nullo, queste etichette catalizzano la diffusione di stili e linguaggi estetici, diffondendo le musiche prodotte dai live coders al di fuori del contesto di provenienza.
La piattaforma web prediletta per la pubblicazione, lo streaming e la vendita di supporti fisici e merchandising è Bandcamp. Alcune tra le etichette che negli ultimi anni si sono dedicate al live coding sono la berlinese Conditional di Calum Gunn (Conditional, n.d.), Kaer'Uiks (n.d.) di Daniel Glaser, la polacca Outlines (2020), la PC Music (2019) e la Computer Club (n.d.) di Sheffield.
L'etichetta italiana Umanesimo Artificiale (2020) ha pubblicato una raccolta di musica algoritmica corredata da un video per ogni traccia, in cui l'autore spiega il funzionamento del suo sistema e vi improvvisa.
Numerose altre *label* hanno in catalogo lavori prodotti con tecniche di live coding; una discografia completa richiederebbe uno studio dedicato.
Per il momento basti considerare che il fenomeno è crescente, e che in futuro potrebbe innescare riflessioni sulle differenze metodologiche ed estetiche tra esibizioni dal vivo e pubblicazioni "in studio" in relazione a un'arte essenzialmente legata al tempo reale.

1.5 Hacking culture, Do It Yourself e Open-Source

La pratica del live coding si basa su presupposti filosofici e politici strettamente relazionati al dibattito contemporaneo sulle politiche digitali e il lavoro cognitivo.

DIY, hacking culture e open source sono tre princìpi ideologici e pragmatici che interessano vaste aree dell'arte e del *making,* implicando riflessioni sulla proprietà intellettuale, il mercato tecnologico e il concetto di "bene comune" applicato ai beni informazionali.
Sebbene si possa praticare live coding senza porsi interrogativi in merito, la comunità dei live coders ha più volte dimostrato interesse; nelle conferenze, non mancano dibattiti sulle questioni politiche, ambientali e filosofiche. Ancor più rilevante è il fatto che una parte piuttosto sostanziale della comunità applichi nella pratica i princìpi di hacking e condivisione.

Con la sigla *DIY* (*do it yourself*), letteralmente "fai da te", si intende l'auto-costruzione dei propri strumenti e mezzi a partire da risorse quanto più possibile gratuite o riciclate. Di fatto la modalità produttiva DIY è antica quanto l'essere umano, ma essa è divenuta nell'ultimo secolo un principio politico in reazione alla società consumistica e alla privatizzazione dei mezzi di produzione industriale. Già dalla cultura punk, il DIY rappresenta un principio inscindibile dal modo d'essere culturale, laddove il rifiuto per il prodotto pre-fabbricato dal mercato esprime la volontà di percepire se stessi e i propri mezzi al di fuori dei meccanismi economici.
La sostanziale totalità delle sottoculture musicali sorte a partire dalla fine degli anni '60 applica il DIY nelle modalità produttive e distributive, in assenza di finanziamenti da parte di etichette discografiche cosiddette *major*. Con l'evolversi delle tecnologie disponibili in grande scala, la fondazione di etichette discografiche "indipendenti" e di piccoli studi casalinghi per l'autoproduzione è aumentata notevolmente, specie per quanto riguarda la musica elettronica. In questo senso, il live coding introduce una forma ancor più profonda di DIY: la programmazione dei propri strumenti musicali. Dialogando con live coders nei forum online, si evince che l'indipendenza da sintetizzatori e digital audio workstations ha rappresentato per molti uno dei motivi principali per il passaggio alla programmazione musicale algoritmica.
La possibilità di costruire le proprie astrazioni tramite funzioni algoritmiche mette in contatto i live coders con un'ampia comunità di makers, hackers e

sviluppatori indipendenti, che hanno fatto dell'autoproduzione dei propri mezzi un principio politico imprescindibile.

Il termine "hacker" necessita di una disambiguazione. L'immaginario collettivo associa a questa parola lo stereotipo del ragazzo nerd che passa le giornate al computer a compiere sovversive operazioni di pirateria, diffondendo virus, rubando dati sensibili e violando i protocolli di sicurezza di governi e aziende.
A dire il vero quest'immagine si applica a un numero di hackers estremamente ristretto; le azioni sovra-menzionate sono fortemente condannate dalla maggioranza della comunità dell'hacking.
Nel suo senso originario (Coleman, 2016), l'hacking è un'attitudine alla riconfigurazione delle tecnologie, un processo creativo mediante il quale un determinato artefatto tecnologico o software viene adoperato per funzioni non previste dai suoi progettisti.
Nikitina (2012) ha attribuito alla *hacker culture* abilità creative e artigianali essenziali nell'età digitale. Considerata quest'accezione, si può affermare che il live coding racchiuda in sé elementi provenienti dall'hacking. Il paradigma creativo della programmazione musicale in tempo reale contempla la modifica del codice dei software al fine di ottenere nuove funzionalità: è un'estetica della rielaborazione tecnologica, del riciclo di frammenti di codice, ai quali viene data una natura inedita.

La riconfigurazione si basa sull'accessibilità del codice. La comunità di live coding condivide i frutti del proprio lavoro cognitivo attraverso la pubblicazione degli algoritmi su piattaforme quali GitHub.
La pratica della condivisione del codice sorgente dei software ha una storia lunga e complessa, fatta di scontri ideologici tra idee comunitarie e modalità di business (Di Bona e Ockman, eds., 1999).
Le innumerevoli tipologie di licenze applicate ai software vanno dalla totale attribuzione privata alla condivisione incondizionata, passando per modalità intermedie o altre, quali ad esempio il copyleft.

L'Open Source è il principio di collaborazione e condivisione del codice finalizzato allo sviluppo collettivo e alla personalizzazione dei contenuti. Lungi dall'essere unicamente appannaggio di progetti "dal basso" e auto-prodotti, questa modalità è impiegata anche dalle grandi aziende del mercato digitale (Herstatt e Hels, eds., 2015). Open Source inoltre non indica gratuità; esistono infatti software a pagamento per i quali è possibile contribuire alla programmazione.
La prospettiva politica più diffusa nella comunità del live coding prescrive la totale gratuità e libertà di modifica dei linguaggi di programmazione e delle piattaforme utilizzate. Non si osservano software per il live coding a pagamento e per quasi tutti il codice sorgente è liberamente modificabile.
Una volta create le fondamenta di un linguaggio, il processo di miglioramento ed espansione è un atto partecipativo portato avanti dalla comunità. A fronte dell'assenza di un "servizio assistenza", ogni utente può mettere in discussione alcuni elementi del linguaggio e proporre delle modifiche. Le continue migliorie operate dalla collettività possono restituire flessibilità e permettere l'integrazione di nuovi sistemi e protocolli.
Alcuni linguaggi, come ad esempio SuperCollider, continuano a essere aggiornati per decenni dalla comunità di utenti. Se per linguaggi proprietari, vale a dire posseduti da aziende private, la longevità dipende dalla capacità dell'azienda di essere competitiva in un mercato fatto di concorrenza e continue variazioni di domanda e offerta, un linguaggio open source è in grado di sopravvivere al tempo fintanto che esiste una comunità interessata a farne uso.
Talvolta si desidererebbe che un certo software privato avesse una determinata funzionalità, ma essa non viene implementata dall'azienda, in quanto tale funzionalità viene ritenuta irrilevante o non proficua nel rapporto costi/benefici; nel caso dei software open source partecipativi come SuperCollider, esiste un ampio campo dialettico tra utente e sviluppatore, non mediato dagli interessi aziendali.

In sostanza l'etica della condivisione e della partecipazione è una conditio sine qua non del live coding. TOPLAP è apertamente contraria a un

approccio produttivo opaco; è auspicata la maggior trasparenza possibile, non soltanto nella costruzione del sistema, ma nella totalità della pratica musicale. Come affermano Cox e McLean:

> «Dal momento che il software non è definito soltanto dal codice del programma, ma anche dagli altri materiali richiesti al programma per funzionare, i programmatori diventano parte integrante dell'azione. L'atto del coding diviene un prototipo per l'azione in senso più ampio, che include una critica dell'imperativo commerciale dello sviluppo dei software e anche delle relazioni sociali normative a esso associate. Come il codice sorgente, queste relazioni divengono aperte a ulteriori modifiche» (2013, p.63).

Dunque il rifiuto dei meccanismi commerciali insiti nell'uso di software proprietari è soltanto una componente di una concezione politica che vede l'uso delle tecnologie per il live coding come tramite per la trasmissione di principi etici e sociali più ampi.
L'idea che l'appropriazione dei mezzi produttivi della classe creativa e cognitiva costituisca il punto di partenza per un cambiamento economico e sociale è condivisa anche dai pensatori post-operaisti. Terranova (2014) ha proposto "un nuovo *nomos* per il bene comune post-capitalista", "aprendo possibili linee di contaminazione con gli ampi movimenti di programmatori, hackers e makers".
Per quanto si tratti meramente di una forma artistica ed espressiva, il live coding racchiude le potenzialità per essere un frammento culturale determinante per lo sviluppo di nuove forme di solidarietà ed emancipazione della classe cognitiva.

CAPITOLO II: La pratica del live coding: implicazioni performative

2.1 L'improvvisazione algoritmica

Rispetto all'improvvisazione strumentale, il live coding presenta alcune sostanziali differenze tecniche, neurobiologiche e temporali. La relazione con un codice dinamico, modificato in tempo reale sulla base di un feedback umano/macchina attraverso astrazioni linguistiche, costituisce un quadro fenomenologico dell'improvvisazione piuttosto singolare tra le pratiche musicali.
Gli strumentisti interagiscono con oggetti tangibili attraverso meccanismi sensomotori, producendo materialmente e direttamente eventi sonori concepiti solitamente come note musicali. Al contrario un live coder dialoga con un sistema semantico dinamico, la cui struttura varia nel corso del tempo in funzione delle sue scelte. Il live coder non produce singolarmente gli eventi sonori, ma costruisce e trasforma le strutture che organizzano insiemi, gruppi e pattern di eventi (McLean e Wiggins, 2009). Non ci sono correlazioni fisiche e causali tra il movimento corporeo e l'emissione di suono; infatti un live coder può passare anche diversi minuti a scrivere prima di *valutare* il codice, cioè inviare le informazioni sintattiche all'*interprete* che le elabora e le trasforma in eventi sonori.

Sayer (2015) ha analizzato le implicazioni neurobiologiche del live coding, osservando che i processi cognitivi messi in atto durante un'improvvisazione algoritmica divergono sensibilmente da quelli di un'improvvisazione strumentale. Egli ha teorizzato che l'assenza del carico cognitivo derivante dalle azioni motorie – assolto dalla *memoria oggettuale* – renda i live coders più liberi di impiegare le proprie energie sugli aspetti strutturali e concettuali del suono, facendo maggiormente uso della *memoria processuale*. Le abilità manuali dello strumentista messe in atto durante l'improvvisazione si basano sull'acquisizione di memoria muscolare che dà spesso luogo a soluzioni musicali inconsce e fondate sull'abitudine; i live coders sarebbero

maggiormente liberi di fare scelte consapevoli, dal momento che operano mediante interazioni temporali più dilatate e perciò meno soggette a fattori inconsci e a riflessi sedimentati nella memoria:

> "Non appena le catene del "tempo reale" si allentano, la capacità cognitiva liberata può essere utilizzata per un comportamento musicale guidato maggiormente da considerazioni derivanti dalla memoria processuale e dai processi concettuali, con un ridotto senso di automaticità, piuttosto che essere guidato da modalità più meccaniche di espressione musicale" (Sayer, ibid., p. 91).

Paradossalmente, lasciare che un sistema computazionale assolva alla produzione degli eventi sonori, può dare risultati meno "automatici" rispetto alla relazione corporea e diretta con uno strumento musicale convenzionale. Il live coder attua un'ibridazione tra il pensiero computazionale della macchina e quello emotivo-percettivo dell'essere umano, sfruttando la precisione macchinica per assolvere alle operazioni che risultano più complesse per la mente, come calcoli ed esecuzioni temporalmente accurate; nel frattempo, l'umano può impiegare tutte le sue energie cognitive nelle sue più grandi abilità: l'immaginazione, la scelta e la creatività.

Il musicologo cognitivista Andrew Goldman (2019), sulla base di studi fenomenologici, neuroscientifici e psicologici, ha teorizzato l'esistenza di due distinte modalità improvvisative, *incarnata* (*embodied*) e *proposizionale* (*propositional*), ascrivendo il live coding alla seconda categoria:

> "L'improvvisazione proposizionale si caratterizza come un tipo di performance con un rapporto disgiunto tra il movimento e il suo effetto sensoriale, sia temporalmente che in termini di sistematicità del contenuto del feedback, e dall'utilizzo di decisioni discrete piuttosto che continue" (Goldman, ibid., p. 5).

Il live coding sarebbe dunque *disincarnato,* in quanto è assente la relazione tra gesto e suono. Questa caratteristica determina un differente uso della memoria nei processi decisionali dell'improvvisazione, non più legati al movimento come negli strumenti acustici ed elettroacustici, ma puntuali e discreti nel tempo.
Durante una performance di live coding, l'esecutore può soffermarsi ad ascoltare l'output dell'algoritmo, contemplare una serie di possibili sviluppi o modifiche, operare una scelta e modificare il codice. Questa sequenza di operazioni può svolgersi in un dominio temporale piuttosto ampio, dando luogo a modalità d'interazione analoghe a ciò che Vaggione (2001) ha chiamato *feedback azione/percezione*. Egli auspicava un rapporto tra essere umano e sistemi algoritmici che potesse dar luogo a "formalizzazioni non fondative, ma operazionali, locali, e tattiche" (p.56). In questo senso, il live coding è in grado di creare un paradigma di interazione nel quale l'esperienza percettiva umana plasma l'algoritmo durante il suo sviluppo.

Non mancano tuttavia conseguenze limitanti nel carattere temporale disgiunto del live coding. Se comparato a uno strumento musicale convenzionale, un sistema di live coding presenta generalmente maggiori difficoltà nell'improvvisare con altri strumenti. L'assenza di un'interazione immediata costituisce un impedimento nella produzione di frammenti sonori rapidi ed estemporanei.
Un live coder necessita solitamente di un intervallo temporale compreso tra pochi secondi e diversi minuti per elaborare una risposta sonora coerente con un dialogo tra più musicisti. Questo scarto temporale, definito da McLean e Wiggins (ibid.) "latenza idea-codice" (*idea-to-code latency*), può essere invalidante se si è a contatto con linguaggi musicali che presentano formule e stilemi caratterizzati da densa temporalità e varietà. Sarebbe necessario preparare delle astrazioni algoritmiche specifiche per questo genere di dialoghi, a patto che il linguaggio di programmazione utilizzato lo consenta. Sebbene questo limite possa essere superato dal design di linguaggi focalizzati sull'interazione con altri strumenti, permane una differenza di natura

sostanzialmente incommensurabile, talvolta difficilmente compatibile con altre forme musicali, eppure dotata di potenzialità estremamente peculiari.

L'affrancamento dall'interazione incarnata tra gesto e suono permette al live coder di concentrarsi su processi e strutture più ampie dei singoli eventi. Incentrarsi sulla formalizzazione dei processi non va tuttavia inteso in termini di una strutturazione architettonica e deterministica.
Le performance algoritmiche si sviluppano nel tempo attraverso la ricerca di strutture effimere, dinamiche, stabili soltanto parzialmente, continuamente esposte all'errore e all'imprevedibile. Il carattere improvvisativo d'incertezza e immaginazione è un elemento fondante: il live coding sfrutta la creatività computazionale al fine di trovare sinergie sonore inaspettate, seguendo un paradigma epistemologico aperto all'incertezza.
Il compito del live coder è comprendere in quale momento intervenire, modificando l'algoritmo per direzionare lo sviluppo sonoro. In questo senso Cocker (2016, p.6) ha affermato la necessità di "adottare una posizione mediale" nella quale l'intuito viene impiegato per capire "quando smettere e quando riconfermare il controllo". Una pratica improvvisativa di questo genere richiede una concezione temporale non lineare e incentrata sulla creazione:

> "Nel live coding, la produzione di linguaggio non è solo performativa, ma anche *cairologica*, a volte un linguaggio non ancora conosciuto emerge simultaneamente […] alla situazione emergente in cui si sta svolgendo. In questo caso, la produzione del codice non precede l'azione (sotto forma di script pre-configurato) né segue come annotazione o documentazione, ma viene prodotta simultaneamente alla performance stessa" (Cocker, ibid., p.8).

Per Cocker dunque il tempo del live coding è il tempo del *kairós,* termine usato dagli antichi greci per definire "il tempo opportuno", una modalità percettiva di ordine qualitativo. Contrapponendosi a una visione lineare e cartesiana, la percezione del tempo cairologico esprime un rapporto con

l'algoritmo incentrato sul feedback azione/percezione in una prospettiva improvvisativa. Dunque, sebbene legato al tempo in maniera meno vincolata rispetto all'improvvisazione strumentale, il live coding permette di elaborare una modalità di ascolto e di scelta estremamente inserita nella dimensione temporale.

In alcuni casi la percezione della necessità di modificare il codice è più veloce della capacità di elaborarne tutte le articolazioni sintattiche. Per questo i linguaggi di programmazione ricercano una certa "parsimonia" o "economia" della sintassi, che possa ridurre i tempi di scrittura al minimo e consentire al live coder di intervenire rapidamente nel momento in cui ne percepisce il bisogno.
Ciononostante, le dinamiche temporali del live coding possono risultare piuttosto pressanti, al punto che Julian Rohrhuber, Tom Hall, Renate Wieser e Alberto de Campo (Ludions, 2007) hanno immaginato la possibilità di redigere un manifesto dello *slow coding,* una forma di improvvisazione algoritmica in cui il programmatore non è interessato a ottenere il miglior risultato nel minor tempo possibile, ma anzi si approccia alla scrittura in maniera riflessiva. Questo manifesto non è stato mai scritto e non si riscontrano sviluppi successivi a questa proposta.
Incuriosito dall'idea dello slow coding, ho deciso di applicarne i princìpi in un'improvvisazione svoltasi al Klang, Roma, il 29 luglio 2020. Sebbene mi sia sentito meno teso per l'esibizione e in generale abbia apprezzato la dimensione mentale dello slow coding, ho osservato alcune criticità nella ricezione del pubblico. L'assenza di variazioni rapide nei materiali e nelle forme ha generato una sensazione di ridondanza in molti momenti.
Ritengo perciò che lo slow coding possa essere usato fruttuosamente per la ricerca di materiali compositivi nel contesto di un'improvvisazione in studio, ma che non sia una pratica efficace per una performance. A mio avviso il principio del tempo cairologico è la dimensione esecutiva più appropriata per il live coding.

2.2 Il codice come partitura dinamica

L'aspetto improvvisativo del live coding mette in luce caratteristiche appartenenti alle culture orali, quali l'assenza di strutture formali costituite a priori e una relazione imprescindibile col tempo presente dell'esecuzione. Tuttavia il live coding è una forma di codifica simbolica della musica e in quanto tale presenta relazioni altrettanto forti con la cultura scritta.
La scrittura musicale algoritmica può essere concepita come una partitura con regole e forme sintattiche specifiche del linguaggio adoperato. Benché nei linguaggi di live coding la codifica delle istruzioni riguardanti i parametri del suono non sia rappresentata sotto forma di pentagramma, essa conserva il carattere prescrittivo-descrittivo specifico della scrittura musicale. Difatti il pentagramma come sistema simbolico è soltanto una delle molteplici forme di notazione musicale storicamente utilizzate.
L'aspetto primario della partitura non è la codifica in sé, ma il fatto che quella codifica rappresenti un sistema di riferenti tra compositore e interprete. Non a caso nei linguaggi di live coding il sistema di elaborazione del codice è chiamato *interprete.*

L'idea di una musica meccanica controllata da forme di partitura alternative precede di gran lunga lo sviluppo dei primi computer. La costruzione di strumenti musicali automatizzati, gli *automata,* costruiti a partire dai meccanismi degli orologi, risale a circa un millennio fa (Collins, 2018).
Nel primo decennio del Novecento si diffuse il player piano, un pianoforte automatizzato in grado di eseguire partiture scritte sotto forma di fogli perforati. Questo tipo di scrittura, antenata dei *piano rolls* usati nella scrittura MIDI su digital audio workstation, permetteva di far eseguire al player piano ritmi e frasi estremamente difficoltose per un essere umano.
Il compositore statunitense Conlon Nancarrow, affascinato dalla possibilità di ascoltare musica ineseguibile, compose quarantanove studi per player piano, esplorando combinazioni ritmiche e contrappuntistiche estremamente complesse (Drott, 2004). La musica di Nancarrow anticipa il live coding per

modalità esecutiva e pensiero musicale, prefigurando una forma compositiva in cui l'interprete è una macchina.
Naturalmente tanto la musica per player piano quanto il live coding richiedono una forma di scrittura estremamente rispettosa della sintassi accettata dalla macchina; se un esecutore umano è in grado di interpretare forme simboliche ambigue, sistemi meccanici o informatici funzionano soltanto seguendo regole sintattiche precise.

L'aspetto meccanico e rigoroso nell'interazione tra scrittura e macchina interprete non va inteso in direzione di una prescrizione rigida dei parametri. Molti linguaggi di live coding consentono di applicare forme aleatorie sfruttando variabili stocastiche.
La razionalizzazione in forme sintattiche semplici delle tecniche compositive aleatorie rappresenta una formalizzazione delle sperimentazioni delle avanguardie musicali del XX secolo. Dalle composizioni combinatorie dei minimalisti alla musica stocastica di Xenakis, la ricerca di una forma di codifica più efficiente del pentagramma nella designazione di elementi algoritmici o aleatori trova compimento nei linguaggi di programmazione. Da questo punto di vista, i linguaggi di live coding non divergono da linguaggi in tempo differito come Csound, con la differenza che l'interazione tra compositore e testo può variare dinamicamente durante l'esecuzione. Come afferma Magnusson (2011), questa specifica relazione temporale con la scrittura compositiva presenta caratteri d'innovazione:

> "Il live coder è primariamente un compositore che scrive uno spartito da far eseguire al computer. […] La novità del live coding non è semplicemente che la composizione è diventata un'attività in tempo reale, ma anche che lo strumento compositivo è avanzato al punto di essere visto come una composizione musicale in sé e per sé." (p. 22).

Piuttosto che cercare un linguaggio universale per la scrittura musicale, i programmatori preferiscono creare mini-linguaggi ottimizzati per applicazioni

specifiche. In questo modo la logica della codifica è indirizzata ad adempiere le esigenze particolari di un determinato approccio alla composizione musicale.
Alcuni linguaggi di live coding sono così incentrati su un singolo campo di utilizzo, da essere usati talvolta per un'unica performance, diventando così parte della composizione stessa (Magnusson, 2015).
Ciò che rende il live coding di interesse per lo studio delle modalità di formalizzazione del suono è l'estrema varietà nel design dei linguaggi. Difatti è impossibile parlare in maniera generica delle implicazioni derivanti dai sistemi simbolici usati dal live coding, dal momento che tali sistemi variano enormemente in funzione del linguaggio impiegato.
Vi sono linguaggi puramente testuali e linguaggi ibridi con componenti grafiche; linguaggi con elementi spaziali o del tutto privi di essi; ve ne sono di imperativi, orientati agli oggetti o funzionali. Ciascuno ha sviluppato strategie proprie per la notazione musicale (per una più ampia trattazione sul design dei vari linguaggi di live coding, si veda la sez. 3.1). Ciò che li mette in comune è la capacità di gestire il suono ricorrendo ad astrazioni simboliche sotto forma di partitura dinamica.

2.3 Logiche e potenzialità della scrittura algoritmica

Conciliando improvvisazione e scrittura musicale algoritmica, i linguaggi di live coding consentono di impiegare tutte le tecniche compositive tradizionali nel dominio del tempo reale. Naturalmente il campo di possibilità è strettamente dipendente dal linguaggio utilizzato e dalla capacità del live coder di tradurre il pensiero musicale sotto forma di astrazioni simboliche differenti dalla codifica convenzionale del pentagramma.
La maggioranza dei linguaggi permettono di implementare numerose tecniche di sviluppo metrico, armonico, melodico e contrappuntistico; a seconda del design del linguaggio, alcune operazioni sui materiali compositivi possono risultare più o meno semplici di altre.

Generalmente i linguaggi di live coding eccellono nella capacità di sviluppare molteplici variazioni complesse a partire da una cellula semplice. La proliferazione di materiali minimali attraverso processi matematici e combinatori costituisce una delle modalità compositive più proficue.
Immagazzinando cellule musicali e dati parametrici all'interno di array, liste, collezioni e dizionari, i linguaggi algoritmici permettono di realizzare permutazioni, combinazioni, trasposizioni, rotazioni e tecniche seriali.
Una cellula può essere rappresentata come un insieme ordinato di dati trasformabili con algoritmi di *sorting,* ossia processi che cambiano l'ordine degli elementi in maniera analoga alle tecniche contrappuntistiche tradizionali.
È possibile immagazzinare grandi quantità di dati, come ad esempio intere librerie di cellule ritmiche o melodiche, oppure progressioni armoniche, al fine di richiamarle in tempo reale e sottoporle a trasformazioni mediante *metodi* e *funzioni*.
Nella programmazione *orientata agli oggetti,* un metodo è un'operazione applicata a una specifica tipologia di oggetto; possiamo ad esempio definire il metodo "inverse" e applicarlo a una collezione di dati ordinati (*array*) per invertirne l'ordine:

```
~serie = Array.fill([60, 63, 67, 70, 74]);
~invSerie = ~serie.inverse;
~invSerie.print;

post window: ~invSerie([74, 70, 67, 63, 60]);
```

Queste semplici righe in pseudo-codice "orientato agli oggetti" operano un'inversione di una collezione di dati. Nella prima riga viene creata la *variabile* "~serie", un'istanza dell'oggetto Array; l'array, o vettore, è un contenitore ordinato di dati, in questo caso cinque numeri che in linguaggio MIDI rappresentano un arpeggio di do minore nona ascendente sulla terza ottava del pianoforte. Successivamente alla definizione della variabile ~serie è applicato il metodo "inverse" e il risultato viene immagazzinato nella nuova variabile "~invSerie". Dunque viene richiesto all'interprete di "stampare" il contenuto di ~invSerie sulla *console window* – l'interfaccia di output dell'editor

di scrittura. La console window mostra il contenuto della variabile, in questo caso l'arpeggio precedente in forma discendente.
Con questa logica, ampiamente utilizzata dai programmatori per l'organizzazione di dati di qualunque tipo, è possibile immaginare l'esistenza di metodi e funzioni in grado di compiere qualunque tipo di tecnica compositiva tradizionale. Si osservi ad esempio l'ipotetica funzione "~overlap" in linguaggio SuperCollider:

```
~overlap = { args: array1, array2;
var listnew, maxitem;
listnew = List();

if (array1.size > array2.size)
{maxitem = array1.size}
{maxitem = array2.size};
maxitem.do({args: item, index;

if (array1.includes(array2[index])
{listnew.add(array2[item])};

});

listnew.print; };

~overlap.value([60,64,65,69,100],
[65,64,59,92,85,78,19,100]);

post window: List[65, 64, 100]
```

Questo codice contiene la definizione della funzione "~overlap", un'operazione composita che può essere utilizzata liberamente una volta definita.
Una funzione è solitamente un insieme di istruzioni richiamabili avente dei parametri specifici, detti *argomenti*. In questo caso abbiamo definito che la

funzione ~overlap ha due argomenti, chiamati internamente "array1" e "array2". Successivamente sono generate due variabili, "listnew" e "maxitem". La variabile "listnew" viene istanziata come una lista vuota. Nel secondo paragrafo c'è un *if statement,* un'operazione condizionale; in questo caso l'if statement definisce che se l'array del primo argomento contiene più elementi di quello del secondo, la variabile "maxitem" avrà le dimensioni del primo array; in caso contrario, avrà invece quelle del secondo. Una volta definite le dimensioni della variabile "maxitem" mediante l'if statement, usiamo il valore come contatore per un *ciclo do.*

Un ciclo do compie una funzione un determinato numero di volte. All'interno del ciclo do è presente infatti un procedimento condizionale che compara ciascuno degli elementi del primo array e verifica se è presente anche nel secondo array; nel caso in cui un elemento sia presente in entrambi, esso viene aggiunto al nuovo array "listnew". Dunque viene chiesto alla funzione "~overlap" di stampare il risultato del nuovo array ogni volta che viene richiamata. Successivamente la funzione viene testata assegnando ai due argomenti "array1" e "array2" due liste di dati, ciascuna contenente un numero variabile di numeri. La post window mostra una nuova lista contenente i numeri in comune tra i due array in ingresso.

La funzione creata in questo esempio svolge un'intersezione, una semplice operazione insiemistica che può essere usata in numerosi contesti musicali. Possiamo ad esempio comparare due serie di altezze e ottenere le note comuni, per poi adoperare la nuova serie come un accordo, oppure svolgere ulteriori operazioni sul risultato.

Le funzioni definite possono essere richiamate in tempo reale per elaborare il materiale durante l'esecuzione. Il campo di possibilità derivante da questa logica è estremamente vasto.

Strutture aleatorie o probabilistiche possono essere impiegate nello stesso modo, ricorrendo a librerie di sequenze pseudo-casuali. È possibile ad esempio generare un suono composto da cento oscillatori sinusoidali a frequenza casuale con estrema semplicità, oppure usare una catena di Markov per controllare l'andamento di un parametro del suono. Talvolta è possibile

implementare strutture matematiche complesse come automi cellulari, reti neurali artificiali, attrattori, sistemi frattali, funzioni caotiche, sistemi di Lindenmayer, feedback di dati e algoritmi generativi (a riguardo si veda Nierhaus, 2009). L'uso di tali sistemi può risultare utile tanto per il controllo micro-formale, quanto per definire andamenti più ampi. Alcune applicazioni di strutture stocastiche sono approfondite nella sezione 4.2.

Grazie alla precisione nel calcolo e nell'esecuzione, alcuni linguaggi di live coding sono particolarmente efficaci nell'elaborazione metrica e ritmica dei materiali. Gestendo gli intervalli temporali tra gli eventi mediante astrazioni matematiche come cicli, metronomi e contatori, risulta piuttosto agevole la scrittura di polimetrie, poliritmi, metriche binarie e algoritmi di Björklund. In molti casi la sintassi algoritmica è estremamente più concisa della notazione tradizionale. Si veda ad esempio questo pattern scritto in linguaggio TidalCycles:

```
let a p = struct "t(7,13,<0 1 3>)" $ p
    b p = struct "t(<3 4 5>,13,<3 5 7 8>)"
            $ p
in
d1 $ ev 5 (rev)
    $ stack [
     a $ s "bd",
     b $ s "cp"]
```

Questo breve codice riproduce due pattern applicati ai campioni "bd" e "cp". Il primo pattern, "a", divide una misura in tredici parti e applica sette eventi distribuiti nella maniera più equidistante possibile, rispettando la suddivisione metrica; nella prima ripetizione della misura, gli eventi di "a" ruotano verso destra di una suddivisione, dunque nella seconda ripetizione ruotano di altre due suddivisioni per poi tornare alla posizione originaria. Parallelamente, il pattern "b" usa la stessa suddivisione metrica, distribuendo un numero variabile tra tre e cinque eventi equidistanti, la cui rotazione varia da tre a otto suddivisioni verso destra seguendo un pattern della durata di quattro misure.

Ogni cinque misure, entrambi i pattern vengono invertiti per una singola misura. Dopo esser stato valutato, il blocco di codice continua a ripetersi finché non viene modificato o interrotto, creando sfasamenti tramite la rotazione e l'inversione dei due pattern.
La sequenza ritmica prodotta sarebbe estremamente complessa e lunga se fosse rappresentata su spartito. Con la logica algoritmica, interazioni di questo genere divengono immediatamente disponibili per l'improvvisazione.

Tutte le tecniche sopramenzionate possono essere impiegate per il controllo dei parametri timbrici di un sintetizzatore o di un sampler.
Generalmente i linguaggi di live coding constano di una sezione di controllo dei parametri e una di sintesi del suono. Talvolta la generazione di segnale è separata dalla scrittura di pattern e sequenze di dati, facendo uso di due linguaggi di programmazione differenti. Molti linguaggi, tra cui TidalCycles, ixilang, SonicPi e FoxDot, si occupano unicamente del controllo parametrico, sfruttando il motore audio e il linguaggio di SuperCollider per la gestione di sintetizzatori, sampler e processori di segnale.
Nella maggioranza dei casi i sintetizzatori vengono programmati prima della performance e successivamente controllati con variabili e funzioni assegnati ai parametri timbrici. Le mie implementazioni di sintetizzatori e processori di segnale su SuperCollider sono illustrate nella sezione 4.1.

Qualunque sia il linguaggio di live coding prediletto, risulta evidente la necessità di preparare le proprie astrazioni per l'organizzazione dei materiali e la manipolazione dei parametri. Sebbene ogni linguaggio abbia le proprie modalità di formalizzazione, è compito del live coder elaborare delle strategie proprie, trovando modalità di rappresentazione algoritmica del pensiero compositivo.

2.4 Aspetti semiotici e audiovisivi del live coding

Proiettare il codice durante le performance è una consuetudine consolidata nell'assoluta maggioranza dei live coders. Sebbene alcuni di loro preferiscano non mostrare le proprie astrazioni linguistiche, tanto in streaming quanto dal vivo la condivisione dello schermo è una pratica estremamente diffusa.
Il presupposto concettuale della proiezione del codice è fondato sull'idea di trasparenza del processo (McLean e Wiggins, 2009). In assenza di un gesto performativo forte, come quello della relazione motoria diretta tra strumento fisico ed emissione del suono, il live coder decide di mostrare quello che avviene all'interno del suo computer, rendendo il pubblico partecipe di ciò che scrive. Questo gesto di apertura manifesta la volontà di rendere visibili i processi cognitivi messi in atto dall'esecutore sotto forma di linguaggi informatici.

Circa venti anni fa il compositore elettroacustico Kim Cascone (2002, 2004) mise in evidenza la necessità di trovare nuovi contesti e modalità performative per la laptop music. Egli infatti criticò l'opacità del processo performativo e definì l'assenza di un codice condiviso tra pubblico e musicisti come elemento estremamente problematico. Da allora si è diffuso il topos, certamente ironico ma indubbiamente rilevante da considerare, per il quale un musicista di laptop music, vista la mancanza di relazione diretta tra gesto e suono, potrebbe tranquillamente star controllando le mail durante un'esibizione.
Proprio per evitare una tale distanza comunicativa, la comunità del live coding adotta la condivisione dello schermo come input visivo.

Naturalmente la proiezione del codice rende la performance audiovisiva, aprendo un campo complesso di conseguenze. Inoltre il materiale visivo, essendo costituito perlopiù da parole e simboli, porta con sé una dimensione semantico-linguistica ulteriore alla semplice osservazione visiva.
Benché il significato dei codici proiettati sia certamente incomprensibile per la maggioranza degli osservatori, i live coders sperano che la condivisione

possa dare luogo a una maggiore interazione con il pubblico. Non è esclusa l'idea che mostrare gli algoritmi durante un'esibizione divenga un codice performativo collettivamente condiviso, al pari del vedere un chitarrista fare un assolo.
Sinora la proiezione del codice ha scaturito reazioni di meraviglia e un senso di novità per il pubblico, talvolta associato a un immaginario futuristico. L'impossibilità di comprendere il significato degli algoritmi – oltre all'incredulità nel relazionare causalmente il testo osservato col fenomeno sonoro – ha prodotto in molte persone una sensazione di virtuosismo e talvolta anche di superflua complicazione del processo creativo. Questo genere di reazioni sono conseguenze dell'aspetto di sostanziale novità del live coding per una larga parte del pubblico; difatti in coloro che si sono abituati all'esternazione del codice mediante proiettori si manifesta un più misurato interesse nell'osservare il codice.

È incerto se la proiezione del codice rappresenti una modalità proficua per l'attenzione del pubblico nei confronti della musica. Come in altre metodologie di relazione audiovisiva dal vivo con la musica elettronica, persiste il dubbio che il contenuto visivo possa costituire un elemento di distrazione.
L'aspetto visuale rappresentato dal puro codice rappresenta infatti una relazione audiovisiva meritevole di studi più approfonditi. Certamente si osserva che esiste una correlazione tra movimento fisico del performer e codice proiettato; tuttavia non sempre nei concerti è possibile osservare le dita del live coder sulla tastiera del computer.
L'interazione audiovisiva tra musica e editor di testo è sostanzialmente basata su una relazione semiotica tra astrazione linguistica e suono prodotto. Dal punto di vista puramente visivo, il prodotto percettivo della proiezione è estremamente statico.
Avendo partecipato a diversi algorave, le mie osservazioni empiriche mostrano che l'interesse per il codice da parte del pubblico è altissimo all'inizio della performance e tende a svanire del tutto dopo un intervallo temporale compreso tra i cinque e i quindici minuti. A quel punto solitamente

sopraggiunge negli ascoltatori la volontà di relazionarsi col suono attraverso il movimento corporeo.

L'editor di testo proiettato può assolvere alla funzione di interfaccia comunicativa tra performer e pubblico. Talvolta i live coders scrivono messaggi rivolti agli ascoltatori, tentando d'instaurare una relazione discorsiva. Durante le sue improvvisazioni Renick Bell (2020) utilizza un editor di testo personalizzato dove gran parte dello spazio è occupato dalla console window. Attraverso una funzione di *flooding* egli può stamparvi messaggi che possono essere reiterati centinaia di volte fino a occuparne l'intera area. Bell sfrutta questa modalità per spiegare al pubblico il suo processo creativo e le sue prospettive estetiche, comunicando a un livello meta-testuale in cui codice, linguaggio scritto, suono e disposizione spaziale interagiscono nella produzione di significato.

Esiste un'estetica visiva intrinseca del codice. Alcuni improvvisatori algoritmici pongono estrema attenzione alle forme e i colori prodotti dai linguaggi di programmazione. Gli stessi sviluppatori usano colori e simboli come sintassi secondaria anche in un'ottica puramente estetica, al punto che è stata supposta la creazione di un'analisi morfologica del codice analoga alla spettromorfologia dei suoni di Denis Smalley (McLean et al., 2010).
Indubbiamente gli aspetti percettivi e interpretativi del codice da parte del pubblico potrebbero essere analizzati più approfonditamente; non mancano esperimenti di interfacce di testo dal design visivo ornamentale (si veda ad esempio ORCA, n.d.) e in altri casi l'uso di caratteri in codici ASCII per creare immagini figurative (*ASCII art*).

Forme di relazione audiovisiva più efficaci si riscontrano quando al live coding audio è accompagnato un live coding video. Analogamente ai linguaggi di programmazione audio in tempo reale, vi sono numerosi linguaggi di improvvisazione video mediante scrittura di codice. Durante i festival di live coding si possono osservare performance audiovisive, eseguite

spesso da più persone; talvolta vi sono due o più proiettori, uno dedicato al linguaggio audio e l'altro al video coding.
I linguaggi di programmazione video più diffusi negli algorave sono Hydra, GLSL, Veda.js, VVVV, Open Frameworks e Processing. In questi linguaggi il codice viene mostrato sovrimpresso al video generato, generalmente costituito da sintesi o da manipolazione di filmati pre-registrati. Il campo di possibilità dei linguaggi di video coding è limitato dalle capacità computazionali di rendering in tempo reale. Si osserva l'uso di poligoni regolari con poche facce o *instancing* di modelli tridimensionali semplici, generalmente modulati dal segnale audio.
Sebbene un'analisi approfondita del video coding esuli dagli scopi di questa tesi, vale la pena soffermarsi brevemente su Hydra, un mini-linguaggio in Javascript creato da Olivia Jack. Hydra è ispirato alla sintesi video modulare analogica e funziona mediante operatori interconnessi, presentando numerose analogie con la sintesi audio, quali ad esempio l'uso di oscillatori (Jack, 2019). Disponibile direttamente su web browser senza bisogno di installazione, Hydra permette di operare compositing in tempo reale tra diverse sorgenti, usare segnali audio sotto forma di dati FFT per modulare i parametri video e creare sistemi di feedback tra videocamera, schermata del computer, browser tab e video pre-registrati.
L'uso del feedback dell'editor di testo permette di costruire sistemi visivi fortemente dinamici a partire da traslazioni, movimenti e sovrimpressioni del codice stesso, dando luogo a un'estetica della programmazione coerente con la proiezione del live coding audio.

Capitolo III: Definizione di un sistema di live coding

3.1 Software e librerie per il live coding

Il primo passo per articolare il proprio ambiente di live coding consiste nella scelta del linguaggio. Al momento esistono più di quarantacinque linguaggi di improvvisazione musicale algoritmica (Toplap, n.d.) e il numero è in costante aumento.
Prima di procedere nell'analisi dei parametri rilevanti per la creazione o l'adozione di un sistema di live coding, può essere proficuo trattare alcuni linguaggi per compararne il design e la filosofia.
Benché vi sia una notevole proliferazione di librerie, sistemi e linguaggi, soltanto pochi di essi offrono un sistema di programmazione musicale sufficientemente elaborato, flessibile e ben documentato. Sono stati dunque selezionati alcuni tra i linguaggi che presentano tali qualità, per mostrare convergenze e divergenze nella concezione del live coding. Si noti tuttavia che TidalCycles, il linguaggio oggetto della parte pratica di questa tesi, sarà esaminato nella sezione 3.2.

SonicPi è un linguaggio di programmazione open source sviluppato da Sam Aaron e incentrato sulla didattica del live coding. Differentemente dalla maggioranza degli altri sistemi, SonicPi è pensato per essere appreso da bambini e adulti, fornendo un ambiente di improvvisazione giocoso ed estremamente semplice.
Benché faccia uso del server audio di SuperCollider, l'utente non ha bisogno di conoscere alcun elemento della programmazione di SuperCollider per suonare con SonicPi; infatti quest'ultimo contiene una libreria di sintetizzatori pre-programmati.
In nome dell'intelligibilità, SonicPi sacrifica l'efficienza nella scrittura. Se comparato ad altri linguaggi, esso risulta estremamente *verboso,* specie per quanto riguarda la scrittura di sequenze di altezze. Ad esempio la sintassi per scrivere un arpeggio di do minore settima formato da semiminime è la seguente:

```
play 60
sleep 1
play 63
sleep 1
play 67
sleep 1
play 70
```

Lo stesso arpeggio in TidalCycles è annotato in questo modo:

```
"60 63 67 70"
```

Nel primo caso la sequenza richiede la digitazione di più di cinquanta caratteri, mentre nel secondo soltanto tredici. Tuttavia il primo presenta una maggiore chiarezza per un principiante, perciò SonicPi è un linguaggio frequentemente impiegato nella didattica scolastica di tutte le età.

FoxDot (n.d.) è un ambiente di live coding nato nel 2015 che permette di improvvisare musica algoritmica in Python, il linguaggio di programmazione attualmente più usato al mondo. Vista l'estrema diffusione di Python e la semplicità della sua sintassi, FoxDot vanta una vasta comunità di utenti.
Analogamente a SonicPi, il server audio prescelto è SuperCollider e vi sono dei sintetizzatori specificamente programmati per FoxDot, con la possibilità per l'utente di programmarne altri in linguaggio SuperCollider. Oltre a sintetizzatori, FoxDot contiene una libreria di campioni che possono essere articolati in pattern con una sintassi simbolica piuttosto sintetica:

```
d1 >> play(P["x - - (- [- o]) - - o (- =) -"])
```

L'esempio mostra un pattern formato da lettere, simboli e parentesi, dove le lettere corrispondono a un campione collocato in una specifica cartella

dell'hard disk, mentre i simboli e le parentesi permettono di articolare gli eventi nel tempo, creando raggruppamenti, pause e sincopi.
Inoltre FoxDot sfrutta la programmazione orientata agli oggetti di Python per implementare metodi e funzioni in grado di mutare i pattern e modulare i parametri del suono:

```
d1 >> play(P["x - - (- [- o]) x o -).layer("mirror")])
```

In questo caso il metodo ".layer("mirror")" crea un altro pattern sovrapposto, a cui viene applicata un'inversione rispetto al pattern originario.

Gibber è un ambiente di coding creativo che permette di programmare contenuti audiovisivi in linguaggio JavaScript direttamente su web browser (Roberts e Kuchera-Morin, 2012). Senza bisogno di alcuna installazione, accedendo al sito gibber.cc è possibile controllare sintetizzatori e creare video 3D usando Web Audio e WebGL, i protocolli per la produzione di suono e immagine su browser.
Nonostante l'estrema portabilità e la possibilità di condividere codice facilmente, Gibber presenta alcuni limiti derivanti dall'insufficiente complessità del protocollo audio su web. Tuttavia, grazie all'uso di librerie esterne in JavaScript è possibile estendere le potenzialità dell'ambiente, usando ad esempio file audio contenuti in archivi online.

Un ambiente di coding certamente degno di nota è Extempore (n.d.), sviluppato da Andrew Sorensen con lo scopo di creare un sistema di live coding non necessariamente legato alla musica, ma che permetta di programmare in tempo reale anche per altre necessità. Extempore dunque è al contempo un software di live coding e un ambiente per la programmazione in tempo reale intesa in senso più generico.
Esso è costituito dall'interazione di due linguaggi: Scheme, un linguaggio funzionale che fa ampio uso di funzioni ricorsive e *lazy programming,* e xtlang, un linguaggio simile a C. Funzioni scritte in Extempore comprendono i due

linguaggi usati talvolta simultaneamente, laddove Scheme è generalmente usato per controllare liste e *chiusure,* mentre xtlang è impiegato per gestire la memoria e assolvere a funzioni di livello più basso.
Extempore comprende un sintetizzatore programmabile chiamato *sharedsystem* e ispirato alla sintesi modulare, col quale è possibile creare delle patch personali. Inoltre i linguaggi utilizzati permettono di creare propri strumenti a livello DSP e controllarli mediante funzioni.
Vista la complessità dell'architettura di Extempore, questo linguaggio è solitamente utilizzato da musicisti con numerose esperienze pregresse nella programmazione. In questo senso, le potenzialità di Extempore sono piuttosto ampie, ma il linguaggio è pressoché inaccessibile ai principianti.

Di tutt'altra concezione è invece ixilang (n.d.), un linguaggio di programmazione sviluppato da Thor Magnusson al fine di improvvisare con SuperCollider mediante astrazioni simboliche più rapide e intuitive. A tal scopo ixilang sfrutta l'organizzazione spaziale dei simboli come modalità di scrittura dei pattern, combinando testi e spazi con un approccio ibrido tra coding e partitura. Scrivendo in riferimento a una griglia rappresentata dal carattere pipe (|), ixilang consente di sovrapporre linee musicali applicate a sintetizzatori scritti su SuperCollider:

```
wiio -> |t  t  t  t  |
woos -> |t   t t |!2^2889^
xoss -> |qqit ti i     iii       iiii  |^2899^
group drum -> wiio woos xoss
drum -> reverb
```

Nell'esempio qui riportato vi sono tre linee percussive parallele (chiamate arbitrariamente wiio, woos, xoss) i cui pattern sono articolati spazialmente nell'area compresa tra i pipe. Gli spazi tra i caratteri definiscono le pause tra un evento e l'altro.

Thor Magnusson ha creato anche Threnoscope, un ambiente di live coding ibrido che contiene elementi visivi e testuali (Magnusson, 2015). Concepito per l'improvvisazione di suoni tessiturali estesi nel tempo, l'interfaccia di Threnoscope consta di tre parti: un editor di testo per la scrittura del codice, una zona grafica in cui gli eventi sono rappresentati sotto forma di cerchi concentrici e un sistema di arrangiamento verticale degli eventi simile a un piano roll. Threnoscope permette dunque di interagire tramite la scrittura, ma anche con l'uso del mouse.

Piuttosto singolare è il design di Orca, un "linguaggio di programmazione esoterica" incentrato sulla creazione di segnale MIDI e OSC da inviare a sintetizzatori o software esterni (hundredrabbits, n.d.). In Orca ogni lettera della tastiera equivale a una funzione, perciò il linguaggio intesse una relazione stretta tra il gesto del live coder e il risultato sonoro.
La schermata di Orca è rappresentata da una matrice di punti nei quali l'utente può scrivere lettere; la disposizione spaziale della scrittura rispetto alla matrice influenza l'interazione tra gli elementi.
Sebbene estremamente immediato e capace di creare pattern sofisticati, Orca non è in grado di generare segnale audio e il numero di funzioni di controllo è pari alle lettere dell'alfabeto. Per questo motivo, benché visualmente affascinante, il linguaggio presenta delle evidenti limitazioni e non consente sufficiente personalizzazione.

3.2 Parametri per la scelta di un linguaggio di programmazione

Vista la pluralità di design disponibili per il live coding, scegliere un linguaggio su cui articolare il proprio pensiero musicale può risultare difficile. Occorre riflettere sulle caratteristiche ideali in funzione delle proprie conoscenze informatiche e musicali pregresse, orientandosi verso linguaggi flessibili che permettano di realizzare un vasto numero di tecniche compositive e di processamento elettroacustico.

Imparare a padroneggiare un linguaggio di programmazione musicale può richiedere mesi o anni di pratica, perciò è importante decidere con cognizione di causa quale linguaggio sia il prediletto prima di spendere tempo nello studio di superflue astrazioni informatiche.
Naturalmente la curva di apprendimento è un parametro fondamentale da considerare: con alcuni linguaggi è possibile imparare a realizzare sequenze sonore significative anche in meno di un'ora, con altri c'è bisogno di studiare per settimane l'infrastruttura logica e informatica prima di intraprendere la pratica vera e propria. Tuttavia quando un linguaggio si presenta estremamente semplice nell'apprendimento, bisogna considerare sin da subito se esso è in grado di raggiungere la complessità necessaria a realizzare le proprie idee musicali; alcuni mini-linguaggi di live coding sono estremamente facili da imparare e possono dare molta soddisfazione nelle prime ore di utilizzo, per poi rivelare gradualmente i propri limiti.

Molti tra coloro che si avvicinano al live coding sono programmatori o appassionati di informatica. Premesso che alcuni di loro, proprio per via del fascino nei confronti della programmazione, prediligono imparare più di un linguaggio per fare esperienza di differenti concezioni di design, solitamente essi direzionano le proprie scelte verso software di live coding costruiti su linguaggi a loro già noti. Nella sezione precedente sono stati menzionati ambienti di programmazione che impiegano linguaggi *generic purpose* come Python e JavaScript; esistono naturalmente anche librerie di live coding che usano altri linguaggi noti come C, Pearl, Ruby, Rust, Haskell, Java e Clojure.
Conoscere preventivamente la sintassi del linguaggio può essere un vantaggio notevole per accelerare l'apprendimento, tuttavia una selezione basata soltanto sulle conoscenze pregresse può rivelarsi insufficiente da alcuni punti di vista. Occorre infatti ragionare su molti altri parametri.

Benché si conosca la logica sintattica di un linguaggio per via di precedenti studi, è bene valutare se essa sia efficace nel contesto del live coding. La programmazione in tempo reale ha necessità sintattiche del tutto differenti dalla programmazione per compilazione. Quando si scrive un software, uno

script o una pagina web, si cerca generalmente di fare uso di strutture sintattiche chiare, facendo attenzione al rendere la totalità dei processi intelligibili a una lettura postuma.
Nel caso del live coding, la caratteristica sintattica più rilevante è la brevità: dover scrivere anche soltanto una decina di caratteri in più per svolgere una determinata operazione può influenzare negativamente le possibilità performative. Pertanto in questo contesto l'economia nella scrittura è più importante dell'intelligibilità.
La facilità nella scrittura della sintassi è altrettanto rilevante, in quanto nel contesto del live coding non ci si può permettere di incorrere in errori di battitura dovuti a un sistema linguistico o simbolico inutilmente complicato. Indubbiamente la pratica permette di superare alcuni ostacoli nella scrittura e si può far uso di editor di testo in grado di suggerire una parola prima che essa sia stata scritta per intero, oppure si possono creare alias testuali più brevi e che inducano meno confusione. Ciononostante è più efficiente fare uso di un linguaggio che in partenza sia costituito da astrazioni simboliche brevi – ad esempio usando un simbolo come "$" invece delle parentesi, o la lettera "r" al posto della funzione "reverse".

Un elemento fondante per la scelta del linguaggio di live coding consiste nelle strategie da esso adottate per il *pattern design*. Col termine *pattern design* mi riferisco al complesso di logiche, funzioni ed elementi sintattici coinvolti nella scrittura e nell'articolazione temporale dei pattern musicali, vale a dire allo sviluppo di incisi, semi-frasi, frasi e periodi nel corso dell'improvvisazione.
A tal proposito la brevità è imprescindibile, ma lo è in primo luogo la flessibilità ritmica e l'insieme di possibilità contrappuntistiche. In una concezione di pattern design efficiente i pattern vengono scritti tra virgolette (" ") e fanno uso di simboli per denotare pause ed eventi – talvolta simboli come la tilde "~" equivalgono a una pausa, oppure si usano i valori booleani 0 e 1 per definire il ritmo degli eventi.
Una formalizzazione sintattica delle funzioni temporali è necessaria al rendere agevole la scrittura. Bisogna inoltre prendere in analisi le potenzialità del

linguaggio nel gestire suddivisioni ritmiche complesse e valori di durata non lineari.
La modifica dei pattern tramite funzioni o metodi è una strategia estremamente proficua per l'elaborazione dei materiali; un buon linguaggio di live coding consta di una libreria di astrazioni facilmente richiamabili in grado di trasformare in maniera complessa e rapida un pattern semplice.
Qualunque linguaggio non preveda funzioni e metodi di questo tipo tende a rendere difficoltosa la programmazione ritmica e la gestione di linee musicali parallele.

Più in generale si può affermare che la quantità e la varietà di funzioni disponibili tende ad ampliare il campo di possibilità per il live coder. Ci sono linguaggi che cercano di assolvere all'improvvisazione tramite un numero ristretto di funzioni; sebbene questo genere di limitazione richieda meno sforzo mnemonico e possa talvolta risultare stimolante per l'improvvisatore, la stessa scelta può essere operata in un linguaggio che comprende un vasto insieme di operazioni possibili.
Naturalmente non si sostiene di dover giudicare i linguaggi in termini puramente quantitativi, ma la presenza di un "vocabolario" ampio corrisponde a più estese potenzialità espressive.
In questo senso può essere utile, specie per l'utente intermedio o esperto, la possibilità di scrivere le proprie astrazioni sotto forma di funzioni. La creazione di una propria libreria, oltre a consentire lo sviluppo di nuove forme organizzative del suono, permette di semplificare alcuni processi ricorrenti nella pratica personale con sistemi più rapidi.
Se ad esempio in TidalCycles un utente ricorre piuttosto spesso a un algoritmo di questo tipo:

```
d1 $ sometimesBy x ((fast y) . (degradeBy z))
$ ecc.
```

Egli può definire una funzione che abbia il medesimo comportamento:

```
let funz x y z = $ sometimesBy x (fast y).
(degradeBy z)) $ p
```

In questo modo può richiamare quel comportamento composito mantenendo la flessibilità della definizione dei parametri attraverso l'uso di argomenti (in questo esempio rappresentati dai simboli x, y e z):

```
d1 $ funz 0.5 4 0.3 $ ecc.
```

Nell'esempio un processo sonoro che richiedeva abitualmente circa 45 caratteri, per altro di difficoltosa scrittura, adesso ne richiede circa 15, migliorando sensibilmente la parsimonia nella sintassi.

Tra i parametri da prendere in considerazione non può essere sottovalutata l'implementazione del server audio. Generalmente i linguaggi di live coding includono una libreria di sintetizzatori e campionatori controllabili, ma soltanto alcuni permettono modifiche e aggiunte. Per un musicista elettroacustico il fatto di poter gestire i propri campioni e i propri algoritmi di sintesi è una *conditio sine qua non* a prescindere dalle metodologie compositive e improvvisative scelte.
D'altronde le librerie di live coding constano in genere di qualche sintetizzatore sottrattivo e di campioni percussivi, certamente un insieme di elementi insufficiente per la maggioranza degli improvvisatori interessati all'esplorazione timbrica.
A tal proposito si evidenzia che un gran numero di utenti sembrerebbe essere maggiormente interessato allo *scheduling,* ovvero alla logica gestionale degli eventi, più che alla trasformazione dei suoni nel tempo e alla ricerca di timbri originali. Applicare una concezione elettroacustica al live coding richiede di scegliere un linguaggio che permetta di articolare tecniche di sintesi e processamento sofisticate, pur mantenendo la scrittura dei parametri agevole.
I vari linguaggi costruiti sul server audio di SuperCollider presentano tali possibilità, dal momento che SuperCollider permette di programmare delle *SynthDef,* cioè delle definizioni di algoritmi di sintesi richiamabili e modulabili

durante l'esecuzione. La creazione di SynthDef richiede la conoscenza del linguaggio SuperCollider oltre al linguaggio di live coding nel quale esse vengono utilizzate.

Dunque la curva di apprendimento, l'economia della sintassi, la personalizzazione, la quantità di funzioni disponibili e la flessibilità del server audio sono gli elementi principali da prendere in considerazione. È tuttavia necessario ragionare anche sulla stabilità e sulle informazioni disponibili in merito al linguaggio.
La presenza di bug o la possibilità di produrre errori che causino crash di sistema, distorsioni e glitch non desiderati può compromettere sensibilmente una performance. Un buon linguaggio di live coding deve avere meccanismi di controllo accurati che prevengano dall'insorgenza di questo genere di problematiche. Di solito quando una linea di codice non rispetta la sintassi richiesta, il software non ne consente l'esecuzione e mantiene in riproduzione il codice precedentemente inviato; un messaggio di errore appare sulla console window, segnalando che l'interprete non ha potuto applicare l'algoritmo richiesto.
A seconda del linguaggio di programmazione usato e del tipo di errore, talvolta può essere difficile capire dalla risposta della console window dove si trovi la svista. È importante che un linguaggio di live coding dia risposte chiare in grado di indirizzare l'esecutore nel punto dove la sintassi è errata, onde evitare perdite di tempo durante l'improvvisazione.

Qualunque problema di sintassi si verifichi, può essere utile consultare la documentazione per indagare il motivo degli errori compiuti. La presenza di tutorial, esempi e forum online rappresenta uno spazio di apprendimento edassistenza tecnica insostituibile per il live coder. Perciò è opportuno considerare la presenza di una community di utenti quando si sceglie di studiare un linguaggio di programmazione.
Si può generalizzare sostenendo che maggiore è il numero di programmatori attivi in un dato linguaggio, maggiore è la capacità di superare problemi che dalla prospettiva di un musicista possono risultare insormontabili. Spesso la

programmazione produce effetti inaspettati, difficili da astrarre per il singolo utente, ma lineari e banali per chi conosce a fondo l'architettura del linguaggio. È anche per questo che attorno alla cultura del live coding si è creata una comunità partecipata e trasparente, perché la musica algoritmica richiede la collaborazione di una pluralità di menti per produrre risultati fecondi.

3.3 TidalCycles: architettura, sintassi e pattern design

TidalCycles è un software di live coding sviluppato da Alex McLean (2011). Concepito come un linguaggio *domain specific* contenuto all'interno del più ampio e generico linguaggio funzionale Haskell, TidalCycles consta di una libreria di funzioni, un programmatore degli eventi sonori e un'interfaccia di live coding. Si potrebbe dire che TidalCycles sia una sorta di "dialetto" di Haskell, creato al fine di generare e combinare pattern musicali da inviare al server audio di SuperCollider.
Quando le informazioni contenute nel codice scritto in TidalCycles vengono inviate al programmatore degli eventi (*event scheduler*), questo le organizza nel tempo e le trasmette sotto forma di messaggi in protocollo OSC a SuperDirt, un versatile generatore sonoro scritto in SuperCollider da McLean in collaborazione con Julian Rohrhuber (musikinformatik, n.d.).
SuperDirt è un sistema di routing che permette di gestire i messaggi OSC in arrivo da TidalCycles per controllare una libreria di processori sonori contenente un versatile campionatore e numerosi sintetizzatori. È possibile programmare i propri sintetizzatori su SuperCollider e configurarli nel routing. SuperDirt supporta audio multi-canale e permette di usare la polifonia senza alcuna limitazione.
TidalCycles può essere anche usato per inviare segnali ad altri sistemi, purché essi supportino i protocolli OSC e MIDI. Affinché i sistemi riceventi siano in grado di interpretare correttamente i messaggi OSC, è necessario che essi siano programmati specificamente per questo scopo. Ciò significa che è virtualmente possibile usare TidalCycles per controllare strumenti e media

differenti, come ad esempio laser, luci e sintetizzatori video. Ciononostante il software consta di funzioni create appositamente per il contrappunto sonoro.

La logica organizzativa di TidalCycles è incentrata sul concetto di *ciclo*. Il ciclo è un'unità temporale di riferimento dalla quale dipende la durata di tutti gli eventi sonori. La scrittura dei pattern consiste nel suddividere frazioni e multipli del ciclo:

```
d1 $ s "bd bd [~ bd]"
```

In questo esempio si può osservare un pattern riferito alla funzione sound ("s"). Quando si usano suoni campionati, all'interno del pattern ci si riferisce al nome della cartella contenente il campione desiderato, in questo caso il primo suono della cartella "bd". Il pattern suddivide il ciclo in tre gruppi di egual durata, i primi due contenenti ciascuno un singolo evento sonoro "bd", l'ultimo invece a sua volta suddiviso in due parti di egual durata, laddove la prima parte è occupata da una pausa, mentre la seconda riproduce il suono "bd" – si noti che in questo caso l'evento "bd" dura la metà degli altri due, in quanto è racchiuso in una suddivisione tramite le parentesi quadre.
A un pattern si possono applicare una serie di funzioni trasformatrici, scritte alla sinistra del pattern e collegate dal simbolo "$":

```
d1 $ slow 2 $ rev $ s "bd bd [~ bd]"
```

Al pattern precedente sono state applicate due funzioni. La prima ("slow 2") ne raddoppia la durata facendo corrispondere il pattern alla lunghezza di due cicli, mentre la seconda ("rev") fa sì che il pattern sia riprodotto al contrario.
Le funzioni applicate possono essere periodiche o condizionali:

```
d1 $ every 3 (slow 2) $ sometimesBy 0.3 (rev)
$ s "bd bd [~ bd]"
```

Qui la funzione “slow 2” precedentemente mostrata funge da argomento per la funzione “every”, che la applica soltanto ogni tre cicli per la durata di un singolo ciclo. Ciò significa che il pattern continuerà a ripetersi e a ogni tre ripetizioni ne sarà riprodotta soltanto la prima metà col doppio delle durate, per poi ricominciare alla velocità originaria nel ciclo successivo. La funzione “rev” è invece divenuta argomento della funzione “sometimesBy”, con la quale è specificato che “rev” avrà effetto soltanto nel 30% degli eventi sonori (“0.3”).
Risulta evidente come mediante la combinazione di suddivisioni del ciclo e di funzioni che mutano il comportamento del pattern sia possibile ottenere interazioni complesse in poco tempo. Inoltre ogni argomento di ogni funzione può a sua volta essere un pattern:

```
d1 $ slow "2 1 0.5 3" $ s "bd bd [~ bd]"
```

Invece di un valore fisso, alla funzione “slow” è applicato un pattern formato da quattro valori. Si noti che è sempre il pattern alla destra a definire le suddivisioni, perciò dei quattro valori saranno applicati soltanto i primi tre, ognuno rispettivamente nel momento in cui il pattern di “s” sarà passato da una suddivisione all’altra. Perciò il primo terzo di ciclo sarà riprodotto alla metà della velocità, il secondo a velocità regolare, il terzo al doppio. È interessante denotare che in questo caso la struttura dei pattern è data dalla funzione “s”, ma la durata reale del ciclo dipende dall’interazione tra “s” e “slow”.
Oltre a trasformare i pattern, naturalmente è possibile anche modificare i parametri del suono:

```
d1 $ s "bd bd [~ bd]" # speed "0.5 0.75 1" #
pan (range 0 1 $ slow 1.5 tri)
```

Le funzioni applicate alla destra del pattern “s” sono collegate tramite il simbolo “#” ed esprimono delle variazioni timbriche o di ampiezza. Anche in questo caso i valori possono essere espressi come pattern, ad esempio alla

funzione “speed” sono dati come argomenti tre numeri che corrispondono a valori di pitch derivanti dalla velocità di lettura del campione “bd”. Nel caso della funzione “pan” si può osservare l’uso di un *low frequency oscillator* di forma triangolare, la cui frequenza è definita in rapporto alla durata del ciclo (“slow 1.5” è uguale a una frequenza di un ciclo e mezzo); il valore di posizionamento stereofonico dell’intero pattern dipende dalla funzione “pan” modulata dall’onda triangolare.
Non si deve tuttavia interpretare tale onda triangolare come un vero e proprio LFO in banda audio: il prodotto di questo algoritmo è un flusso di segnali OSC che vengono inviati al server audio di SuperCollider a ogni suddivisione del ciclo; perciò l’oscillatore triangolare viene campionato – analogamente a un *sample & hold* – ogniqualvolta è necessario inviare dati al server audio. Una vera e propria modulazione in banda audio è esprimibile soltanto tramite la programmazione di uno specifico processore di segnale in linguaggio SuperCollider, il cui controllo avrebbe in ogni caso la velocità della suddivisione minima del pattern principale (si veda il capitolo 3.5 per approfondire il funzionamento del server audio).

Sinora è stata mostrata un’unica linea musicale alla volta. Si possono mettere in parallelo un qualunque numero di pattern controllabili in maniera indipendente, oppure si può raggruppare una serie di pattern in un unico *stack:*

```
d1
$ every 4 (rev)
$ stack [
    slow 2 $ s "bd bd bd" # pan (rand),
    fast 1.5 $ s "hh [hh ~ hh]",
    every 2 (slow 2) $ s "~ sn"
        ] # speed 0.5
```

Nell’esempio vi sono tre linee musicali contenute tra parentesi quadre all’interno della funzione “stack”. Esse sono dunque riprodotte in parallelo

simultaneamente, in maniera indipendente l'una dall'altra. Tuttavia le funzioni applicate prima e dopo lo stack ("every 4 (rev) e "# speed 0.5") influenzano tutte le linee musicali. Dunque in TidalCycles si può gestire più voci in maniera indipendente, mantenendo modalità di controllo globale se lo si desidera.

Al momento TidalCycles comprende una libreria di circa trecentocinquanta funzioni che assolvono alle più disparate necessità, dalla stratificazione di linee musicali alle tecniche contrappuntistiche, passando per strutture condizionali e aleatorie, funzioni di controllo timbrico, scrittura MIDI e funzioni di alto livello in grado di regolare il comportamento di altre funzioni nel tempo.
Un insieme così vasto di astrazioni richiede un graduale apprendimento; tuttavia non è necessario impararle tutte, quanto piuttosto selezionare quelle che si avvicinano alle proprie volontà espressive. Un metodo efficace di studio delle funzioni consiste nel ricreare partiture algoritmiche note in TidalCycles. Si osservi ad esempio questo codice:

```
d1 $ stack [
struct ("t t t f t t f t f t t f") $ s "cp" #
gain "0.75",
(("[<1 2 3 4 5 6 7 8 9 10 11 12>]/12"/12) <~)
$ struct ("t t t f t t f t f t t f") $ s
"cp" ]
```

Queste quattro righe riproducono esattamente l'intera partitura di *Clapping Music* di Steve Reich, composizione minimalista in cui una linea ritmica ripete costantemente un pattern in 12/8, mentre una seconda linea ruota di un ottavo ogni dodici ripetizioni fino a compiere una rotazione completa. In questo caso la rotazione avviene mediante la funzione "<~".
Benché il codice presenti una sintassi non immediatamente comprensibile per chi non conosce il linguaggio, si osservi la brevità in comparazione alla

partitura originale. Con TidalCycles un live coder esperto può scrivere strutture musicali del genere inventandole durante l'improvvisazione.

Nell'eventualità in cui la libreria di funzioni non contenesse un'operazione desiderata, è possibile scrivere le proprie funzioni in linguaggio Haskell. Valutandole prima di cominciare a suonare, esse vengono riconosciute al pari di quelle già presenti in TidalCycles. Ciò implica la conoscenza approfondita di Haskell, un linguaggio che può essere piuttosto ostico nell'apprendimento; difatti, non serve conoscere Haskell per poter suonare TidalCycles, ma è necessario per creare funzioni proprie.
Vista la presenza di una vasta comunità di utenti e programmatori, si può richiedere nei forum consigli sulla scrittura di nuove funzioni e proporre aggiunte al linguaggio per le successive versioni. Finché la comunità sarà attiva, TidalCycles potrà espandere il campo di possibilità dell'improvvisazione algoritmica, fornendo al musicista elettroacustico uno strumento di ricerca, sperimentazione e prassi dalle considerevoli potenzialità.

3.4 Haskell: vantaggi di un linguaggio funzionale puro

Sebbene chi scrive non abbia una conoscenza informatica e matematica sufficiente ad analizzarne tutte le implicazioni, vale la pena di soffermarsi brevemente sul funzionamento di Haskell. Questo linguaggio di programmazione, così chiamato in onore del matematico Haskell Curry, è stato sviluppato nel 1990 da un comitato internazionale di accademici al fine di creare un linguaggio puramente funzionale ispirato al lambda calcolo, un sistema matematico per il calcolo delle funzioni ricorsive.

Con la locuzione "puramente funzionale" si intende che ogni funzione del linguaggio corrisponde alla definizione matematica per la quale una funzione è un'espressione fissa tra due insiemi, la cui associazione è singolare e immutabile.

Una volta definita, una funzione in Haskell non può essere mutata. Inoltre ogni elemento del linguaggio è una funzione, persino i numeri interi o booleani. La programmazione consiste dunque nella definizione e concatenazione di funzioni che presentano degli argomenti.

Gli argomenti delle funzioni hanno un *data type* fisso, cioè è necessario definire la tipologia di dati che saranno coinvolti nell'operazione svolta dalla funzione. Se dunque la funzione "somma" richiede in ingresso numeri interi e produce numeri interi, se richiamata essa verrà attivata unicamente se gli argomenti sono tutti numeri interi:

```
somma : : Int -> Int
somma x = x + x
```

In questa definizione la prima riga descrive i data types compatibili con la funzione, mentre la riga successiva designa l'operazione da svolgere. Quando i data type non vengono specificati nella definizione di una funzione, essi vengono inferiti da Haskell (*type inference*).

Haskell applica il *lazy programming,* ossia la "programmazione pigra". Nessuna funzione viene computata fino a quando la computazione non viene esplicitamente richiesta. In questo modo è possibile sfruttare insiemi numerici infiniti – come ad esempio l'insieme R dei numeri reali, oppure tutti i multipli di 17 – senza che il programma sia effettivamente costretto a creare tali insiemi. Possiamo perciò applicare una funzione ricorsiva su un insieme infinito senza incorrere in un crash del programma. Questa logica facilita la concatenazione di funzioni, riducendo il carico computazionale necessario.

L'apprendimento di Haskell può risultare estremamente difficoltoso anche per programmatori esperti. Infatti la logica funzionale del linguaggio differisce quasi del tutto dai linguaggi più usati, generalmente basati sulla programmazione orientata agli oggetti. Haskell richiede uno sforzo di astrazione logico-matematica indubbiamente maggiore di Python, JavaScript

o Java; soprattutto, richiede di pensare alla programmazione in termini dissimili ai linguaggi più convenzionali.
Haskell non fa uso di variabili, cicli do, cicli for e numerosi altri sistemi presenti in quasi tutti gli altri linguaggi. In particolare l'assenza di variabili rappresenta sostanziali difficoltà per chi è abituato a usare contenitori di dati temporanei, la cui definizione varia nel corso del codice. In un certo senso, il fatto che non vi siano variabili riduce la possibilità di bug del linguaggio, fornendo un sistema più stabile; infatti, dal momento che tutti gli elementi definiti sono fissi, non è possibile che si verifichino ambiguità.

Perciò, seppur ostico, Haskell rappresenta un ottimo linguaggio di programmazione per il live coding. È estremamente conciso – a volte due o tre pagine di codice in Python possono essere scritte in Haskell in meno di dieci righe – ed è estremamente stabile, rigoroso nella specifica di funzioni e argomenti, pigro nella computazione, perciò parsimonioso delle risorse della CPU. La logica di concatenamento delle funzioni si adatta efficacemente alla costruzione di strutture complesse dei parametri del suono.
Certamente alcuni dei concetti più avanzati del linguaggio richiedono uno studio approfondito non facile per un musicista, come ad esempio l'uso di *funtori* e *monadi*.
Poiché la scrittura di funzioni proprie non implica solitamente l'uso di strutture così sofisticate, il musicista elettroacustico può accontentarsi di comprendere la logica del linguaggio, scrivere le proprie astrazioni per mezzo degli elementi di programmazione più basilari ed eventualmente mettersi in contatto con programmatori esperti se ve ne è l'effettiva necessità.

3.5 SuperCollider come server audio per il live coding

SuperDirt è un sistema di routing e controllo di sintetizzatori e campioni sviluppato da Julian Rohrhuber e Alex McLean per interfacciare TidalCycles al server audio di SuperCollider. Distribuito come estensione (*quark*) gratuitamente scaricabile da SuperCollider, il suo compito è ricevere messaggi

OSC e decodificarli entro una specifica logica organizzativa del segnale audio; esso consiste infatti in una raccolta di *classi* e definizioni che permettono a SuperCollider di interpretare gli eventi provenienti da TidalCycles.
SuperDirt è scritto in linguaggio SuperCollider, lo stesso usato dall'utente per programmare i sintetizzatori – nonché lo stesso con cui gran parte del software SuperCollider è scritto, permettendo una vasta *introspezione* del codice da parte del linguaggio stesso.

Una trattazione specifica delle caratteristiche del linguaggio SuperCollider è al di là degli scopi di questa tesi; ciononostante può essere rilevante denotare che si tratta di un linguaggio orientato agli oggetti. Nella programmazione orientata agli oggetti esiste una gerarchia, o per meglio dire una tassonomia degli elementi del linguaggio che definisce la struttura globale. Ad esempio l'oggetto *Array* discende dall'oggetto *SequenceableCollection*, che a sua volta proviene dall'oggetto *Collection*; *Array* eredità tutte le proprietà dei suoi antenati. Più si risale la genealogia degli oggetti, più ci si avvicina a enti astratti, fino a giungere all'oggetto da cui tutti gli altri discendono, l'oggetto *Object*.
Perciò quando si crea un oggetto *Array* si sta generando un'istanza particolare – generalmente assegnata a una variabile – di una tipologia di oggetti, cioè della classe di tutte le istanze possibili di *Array*; una classe possiede delle proprietà specifiche ed eredita tutte le proprietà delle classi di oggetti da cui deriva, dando in eredità allo stesso modo tutte le sue proprietà alle classi discendenti da essa.
In questo senso SuperDirt si occupa di definire che tipo di classi siano *DirtError, DirtEvent, DirtModule, DirtOrbit* o *DirtSoundLibrary,* ossia l'insieme di astrazioni necessarie a interpretare i messaggi OSC in arrivo da TidalCycles. La classe degli oggetti *DirtOrbit* equivale alla definizione di un bus audio che permette il routing delle varie linee musicali in parallelo, favorendo il processamento indipendente dei suoni e l'uso di output multicanale, mentre la classe *DirtSoundLibrary* si occupa di organizzare i vari percorsi di cartelle contenenti file audio, caricandoli su appositi buffer predisposti alla lettura.

Allo stesso modo le altre classi definite da SuperDirt permettono una codifica completa e rigorosa di tutti gli elementi necessari all'improvvisazione.

Oltre alle definizioni di nuove classi, SuperDirt include una libreria di campioni e di sintetizzatori. In SuperCollider i sintetizzatori sono programmati generando un'istanza dell'oggetto *SynthDef*. Una SynthDef è letteralmente la definizione di un sintetizzatore; per analogia con la sintesi modulare, potremmo dire che una SynthDef è lo schema della patch che illustra in che modo alcuni moduli sonori elementari vadano relazionati per ottenere un determinato algoritmo di sintesi. Le SynthDef vengono dichiarate allo stesso modo delle funzioni, descrivendo cioè le operazioni e le connessioni da svolgere per ottenere un risultato – in questo caso l'output sonoro finale.
Dal momento che si tratta di mere definizioni, quando una SynthDef viene dichiarata, al server audio viene inviata un'informazione di pochi byte che si occupa di spiegare in che modo una serie di oggetti sono relazionati. Tuttavia la definizione non corrisponde all'esistenza concreta di un sintetizzatore attivo; infatti il server audio genera un'istanza della SynthDef definita soltanto quando essa viene esplicitamente richiesta, per poi estinguerla quando l'inviluppo d'ampiezza del suono è terminato. In questo modo è possibile definire migliaia di algoritmi di sintesi e processamento differenti senza creare peso consistente sul server; se dovessimo cercare di avere a disposizione un tale numero di algoritmi di sintesi in un software come Max/MSP, saremmo costretti a istanziarli effettivamente, impiegando un carico computazionale costante, la cui presenza può risultare superflua per la maggior parte del tempo. In SuperCollider invece si possono avere a disposizione un'infinità di algoritmi differenti, richiamabili semplicemente con il loro nome e con eventuali argomenti in grado di descriverne i parametri.

La descrizione di una SynthDef consiste nella relazione di *UGens* (*Unit Generators*). Una UGen è un modulo sonoro minimale che descrive una determinata operazione a livello DSP o di controllo. In SuperCollider sono

UGens oscillatori, filtri, buffer, trigger, inviluppi, modulatori, matrici FFT, granulatori, algoritmi di modelli fisici, operatori algebrici in banda audio, algoritmi di convoluzione, processi dinamici primitivi, interfacce di ingresso e uscita, ecc.
Al momento la libreria estesa di SuperCollider consta di 1116 UGens, un numero estremamente alto che consente di costruire sostanzialmente qualunque algoritmo di sintesi desiderato. Si osservi ad esempio questo semplice sintetizzatore FM:

```
(
    SynthDef(\nome_arbitrario, {
        | carrierFreq = 400, cmRatio = 1.5,
index = 3, decayTime = 0.01, amp = 1|
        var signal, modFreq;
        modFreq = (carrierFreq *
cmRatio.reciprocal);
        signal = EnvGen.kr(Env.perc(0.001,
decayTime,0.2),1, amp, doneAction: 2) *
         PMOsc.ar(carrierFreq, modFreq, index,
0);
        Out.ar(0, signal)
    }).add
)
```

Questa SynthDef racchiude quanto già descritto sinora: viene generata un'istanza della classe SynthDef con un nome specifico (*nome_arbitrario*), successivamente si definiscono degli argomenti (*carrierFreq, cmRatio, index*) a cui vengono dati dei valori di default. Dunque si procede creando delle variabili, contenitori vuoti necessari all'organizzazione dell'algoritmo (*signal, modFreq*), che vengono occupati da una certa operazione matematica – nel caso di *modFreq*, la moltiplicazione tra la frequenza portante e il reciproco del rapporto, mentre per *signal*, una forma d'inviluppo moltiplicata per *PMOsc*, una UGen che funge da oscillatore pre-configurato per la *phase modulation*.

Dunque la variabile *signal* viene assegnata come contenuto della UGen di uscita "Out"; in quanto ultima operazione della funzione, l'output del segnale ne è il risultato.

3.6 Max/MSP come server audio per il live coding

Dalla versione 1.0 di TidalCycles è possibile trasmettere messaggi OSC a programmi diversi da SuperCollider. Questa funzionalità permette di inviare i dati riguardanti gli eventi sonori a un server audio differente, quale ad esempio Max/MSP.
Per reindirizzare i messaggi OSC a Max è necessario creare una versione personalizzata del file *BootTidal.hs*, un documento in linguaggio Haskell interpretato durante l'avvio del sistema; *BootTidal.hs* descrive in che modo TidalCycles dovrà organizzare i messaggi, a quale indirizzo UDP dovrà inviarli e con quanta latenza. Quando viene aperto un file con estensione *.tidal* salvato all'interno di una cartella contenente un file denominato *BootTidal.hs,* l'interprete applica le istruzioni a esso relative; ciò avviene soltanto se la cartella è posizionata nella sezione *Documenti* del computer. Affinché *BootTidal.hs* prescriva a TidalCycles di inviare i messaggi OSC a Max, occorre modificare la parte del file che definisce la funzione *customTarget:*

```
:{
customTarget = Target {oName = "Max",
oAddress = "127.0.0.1",
oPort = 2020,
oLatency = 0.02,
oSchedule = Live,
oWindow = Nothing
   }
:}
```

oAddress imposta l'indirizzo UDP al nostro computer (il cosiddetto *localhost* ha sempre indirizzo 127.0.0.1), mentre *oPort* definisce la porta OSC che riceverà i messaggi; in questo caso è stata scelta arbitrariamente la porta 2020. È fondamentale che il parametro *oSchedule* sia impostato in modalità *Live:* in questo modo ogni evento sarà inviato nel momento esatto in cui esso deve avvenire e tutte le informazioni saranno opportunamente organizzate.
Per ricevere i messaggi OSC in arrivo da TidalCycles su Max/MSP si utilizza l'oggetto *udpreceive* il cui argomento deve coincidere con la porta OSC definita da *BootTidal.hs* (nel nostro caso la porta 2020). I messaggi ricevuti sono preceduti dalla parola */play2* a cui seguono coppie di termini, il primo relativo al nome del parametro (ad esempio "*note*") e il secondo al valore di tale parametro (solitamente un numero). Per prima cosa è necessario rimuovere la parola */play2* collegando all'outlet dell'*udpreceive* un oggetto *route* con argomento */play2*.
A questo punto si ha un flusso di messaggi ognuno contenente i valori di tutti i parametri inviati da TidalCycles. Affinché i messaggi siano inviati soltanto allo strumento a cui sono riferiti, si può specificare su TidalCycles il nome dello strumento – assegnandolo ad esempio come parametro relativo alla funzione *sound* o *s;* occorre dunque creare su Max/MSP un sistema che sia in grado di smistare i messaggi ai vari strumenti a seconda di quanto è specificato dalla funzione *sound*. Esistono molteplici possibilità costruttive di tale algoritmo. Una delle più semplici consiste nell'usare l'oggetto *route* con argomento *sound* e collegare all'outlet una serie di oggetti *sel*, ciascuno avente come argomento il nome di uno strumento. I messaggi in uscita dagli oggetti *sel* possono essere usati come trigger per indirizzare il messaggio in arrivo verso un determinato outlet mediante l'oggetto *gate*.
A questo punto ciascuno strumento può essere programmato a proprio piacimento, usando i parametri in arrivo da TidalCycles per assolvere a funzioni sonore differenti, oppure creando nuovi parametri in TidalCycles appositamente concepiti per funzionare con quello strumento specifico. Ciascuno strumento dovrà avere un meccanismo di segmentazione e indirizzamento dei vari parametri in ingresso: un buon metodo è quello di separare le coppie parametro-valore con l'oggetto *zl.iter 2* e inviare tali coppie

a un oggetto *route* avente come argomenti ciascuno dei parametri relativi allo strumento. Applicando questa tecnica si può creare un sistema di smistamento che evita che messaggi inappropriati o errati arrivino in parti sensibili dell'algoritmo.

La creazione degli strumenti su Max/MSP può seguire le più disparate concezioni di design e permette di costruire sintetizzatori, campionatori o sistemi di controllo di qualunque tipo. Tuttavia è necessario fare sì che tutti gli strumenti siano polifonici se si vuole evitare che gli eventi sonori vengano interrotti dall'arrivo di un nuovo messaggio indirizzato allo stesso strumento; si consiglia l'utilizzo dell'oggetto *poly~*.
L'output audio degli strumenti può essere successivamente organizzato come se si avesse un mixer di segnali, usando l'oggetto *matrix~* per controllare mandate di eventuali effetti. L'approccio improvvisativo in live coding non impedisce inoltre di fare uso di un controller MIDI o OSC fisico per gestire i livelli dei vari canali in maniera accurata. Ogni parametro può essere controllato simultaneamente dai messaggi OSC in arrivo da TidalCycles e da interfacce di controllo esterne quali ad esempio controller MIDI, sensori o dati in arrivo da altri sistemi. In ogni caso è rilevante considerare il costo computazionale di ciascuna delle operazioni svolte all'interno del sistema costruito in Max/MSP: tale sistema è solitamente più dispendioso di SuperDirt, poiché in quest'ultimo gli strumenti possono essere definiti senza che vengano create delle specifiche istanze di essi; al contrario su Max/MSP gli oggetti devono essere creati in anticipo e l'unico modo per abbattere il costo computazionale è gestire le voci polifoniche tramite l'oggetto *poly~*. Tutto ciò che si trova all'esterno dell'oggetto *poly~* imprime costantemente un carico sulla CPU più o meno elevato a seconda dell'attività svolta. Tale costo, per quanto irrisorio, è sommato a tutti gli altri oggetti presenti nel sistema e pertanto non va sottovalutato.

Benché la mia ricerca sul controllo in live coding di algoritmi sonori creati su Max/MSP sia ancora in uno stadio germinale, se ne osservano già le estreme potenzialità. Sebbene più dispendioso per la CPU, Max/MSP permette un

raffinato controllo del segnale audio. Se comparato a SuperCollider, Max permette di creare algoritmi a un livello più profondo delle UGen usando il linguaggio gen~. Sebbene sia possibile arrivare agli stessi risultati anche programmando in SuperCollider, la difficoltà del linguaggio rende il processo più lento e complesso.
Dal mio punto di vista la programmazione grafica a nodi, grazie anche alla sua somiglianza con la sintesi modulare analogica, si presta maggiormente alla creazione di algoritmi di sintesi e processamento del suono. D'altro canto, la programmazione testuale risulta molto più efficace per il controllo e il sequenziamento dei parametri. Perciò un sistema ibrido come quello descritto, dove la programmazione testuale è sfruttata in tempo reale per controllare sintetizzatori pre-programmati in forma nodale, si manifesta come una soluzione intermedia in grado di trarre beneficio dalle potenzialità di entrambi gli approcci. In ogni caso saranno necessari ulteriori approfondimenti futuri.

Capitolo IV: Implementazione di tecniche compositive e processamenti elettroacustici

4.1 Tecniche di sintesi e processamento

La programmazione di sintetizzatori personalizzati presenta la stessa sintassi della SynthDef mostrata nel capitolo precedente. Tuttavia affinché TidalCycles sia in grado di controllare i parametri dei nuovi sintetizzatori, è necessario definire in Haskell tutti i nuovi argomenti creati:

```
let atk = pF "atk"
    rel = pF "rel"
    note = pF "note"
    tuning = pF "tuning"
```

Gli argomenti delle funzioni, inclusi i nomi delle SynthDef, devono essere descritti mediante questa sintassi ed essere valutati prima dell'improvvisazione. In questo modo è possibile comandare un determinato sintetizzatore (ad esempio *supermandolin*) scrivendo pattern di questo tipo:

```
d1 $ note "60 63 65 67" # s "supermandolin" #
rel 125
```

Con questa linea di codice abbiamo chiesto a SuperDirt di fare eseguire al sintetizzatore *supermandolin* un arpeggio di do minore settima sulla terza ottava con un valore di release pari a centoventicinque millisecondi. Naturalmente la capacità del sintetizzatore di interpretare i messaggi in arrivo da TidalCycles dipende da come è stato programmato: in questo caso si è sottinteso che *supermandolin* sia in grado di interpretare valori di altezza analoghi al protocollo MIDI quando riferiti al parametro *note* – si osservi che anche se la numerazione delle note è uguale al MIDI, la trasmissione del segnale avviene in ogni caso in protocollo OSC; con una corretta configurazione sarebbe stato possibile controllare le altezze con i valori frequenziali esatti.

La creazione di una SynthDef contenente un processore di segnale controllabile mediante TidalCycles richiede alcuni elementi sintattici specifici. A differenza dei sintetizzatori, per i processori è infatti necessario creare un *modulo,* ossia una funzione in grado di relazionare gli eventi ricevuti da SuperDirt con i parametri dell'effetto e di inserirli correttamente nel sistema di routing.
Se creassimo la SynthDef *kstrong* con i parametri *kdepth*, *krate* e *kdecay*, dovremmo anteporre alla funzione di definizione un modulo di questo genere:

```
( ~dirt.addModule('kstrong', { |dirtEvent|
    dirtEvent.sendSynth('kstrong' ++
~dirt.numChannels,
    [
        kdepth: ~kdepth,
        krate: ~krate,
        kdecay: ~kdecay,
        out: ~out
     ]
 )
}, {~kdepth.notNil or: {~krate.notNil or:
{~kdecay.notNil}}}));
```

In questo modo SuperDirt è in grado di interpretare quando l'effetto è attivo, riconoscendo la presenza dei parametri a esso relativi. Dal momento che un processore di segnale viene messo in azione ogniqualvolta SuperDirt riceve un messaggio OSC contenente il nome di uno dei suoi parametri, ogni parametro di ogni effetto deve essere nominato differentemente. Se avessimo un altro processore di segnale col parametro *krate*, entrambe le SynthDef verrebbero attivate contemporaneamente nella catena di processamento. Si osservi che il parametro *out* deve invece essere presente in tutti i moduli affinché essi si configurino correttamente nel routing di SuperDirt.

Su TidalCycles i parametri dei processori di segnale vengono definiti con la stessa sintassi dei parametri dei sintetizzatori:

```
let kdepth = pF "kdepth"
    krate = pF "krate"
    kdecay = pF "kdecay"
```

Dal momento che i processori di segnale sono collegati in serie, può essere rilevante definire il loro ordine prima di cominciare a suonare. A tal fine SuperDirt implementa un metodo:

```
~dirt.orderModules([lpf, bpf, hpf, kstrong,
vowel, distort]);
```

Tramite il metodo *orderModules* viene comunicato a SuperDirt che l'effetto *kstrong* appena creato deve essere inserito nella catena di processamento dopo i filtri *lpf*, *bpf* e *hpf*, ma prima del filtro formante *vowel* e del distorsore *distort*.

Benché normalmente i processori di segnale siano creati in serie, SuperDirt ne presenta alcuni in parallelo, come il riverbero e il delay. Le SynthDef di processamento di questo tipo sono inserite all'interno di ogni *orbit,* cioè ciascuna linea parallela di output possiede una catena di processamento indipendente che contiene un'istanza di tutti i processori creati. I sintetizzatori e i campioni associati a uno stesso orbit condividono la stessa catena.
Al momento della redazione di questa tesi, non è stato ancora possibile programmare effetti paralleli personalizzati, in quanto l'architettura del sistema di routing manifesta delle anomalie nel comportamento degli *orbit* quando tali effetti vengono creati. Sebbene dunque nella documentazione di SuperDirt esista una guida per la configurazione di effetti paralleli, al momento non se ne osservano implementazioni da parte della comunità.

Affinché i sintetizzatori e i processori di segnale creati possano essere usati durante l'improvvisazione, è necessario che essi siano aggiunti al server audio durante l'avvio del sistema. Quando il server viene disattivato non vengono tenute in memoria le SynthDef. Perciò occorre creare un file di *start-up,* ossia un file .scd (l'estensione dei file di SuperCollider) che permetta di definire tutte le SynthDef e collegarle correttamente nella catena di processamento. Dopo aver valutato il metodo *SuperDirt.start,* è sufficiente valutare l'intera lista delle SynthDef racchiusa da parentesi tonde.
La stessa operazione di configurazione va svolta per i parametri personalizzati su TidalCycles; si può creare un file .tidal contenente tutte le funzioni create e valutarlo subito dopo il *boot* dell'interprete di TidalCycles sul proprio editor di testo. Un'altra soluzione consiste nel creare una propria libreria e salvarla nelle cartelle di sistema di *ghci*, il compilatore dinamico di Haskell. Con questa modalità è sufficiente scrivere poche lettere per configurare ampi file contenenti funzioni e parametri; tuttavia, dal momento che il processo di modifica di una libreria è piuttosto laborioso – occorrono infatti diversi comandi su terminale affinché sia correttamente configurata – è consigliato l'uso di un file .tidal da valutare prima dell'esecuzione, specie se si stanno testando nuovi sintetizzatori.

Quanto segue è una panoramica di alcune delle SynthDef da me programmate per l'improvvisazione algoritmica. L'intento di quest'esposizione è mostrare la varietà degli strumenti costruibili e chiarire alcuni aspetti della loro programmazione. Alcune delle soluzioni trovate per la creazione di queste SynthDef sono ispirate a idee degli utenti Guiot (n.d.), Nesso (n.d.) e Mike Hodnick (2016).

Tubes
Tubes è un sintetizzatore basato sul modello fisico di due tubi risonanti. È costruito a partire dalla UGen *TwoTube* sviluppata da Nick Collins e presente nei plugin di estensione di SuperCollider.
Un impulso ad ampio spettro viene usato come eccitatore per il modello risonante, la cui altezza percepita proviene dalla relazione dei valori

frequenziali *f1* e *f2* relativi ai due tubi. Il parametro *depth* permette di controllare il coefficiente *k*, cioè l'intersezione tra i due corpi risonanti. Una variabile chiamata *dummyenv* consente di estendere la risonanza del suono fino a cinque secondi, evitando che la voce venga interrotta dai messaggi OSC successivi.

```
(
SynthDef(\tubes, { |out, pan, depth = 0.3, f1
= 300, f2 = 300|
    var sig, dummyenv, source;
    depth = depth.clip(0, 1);
    f1 = f1.clip(3, 1000);
    f2 = f2.clip(3, 1000);
    source=
WhiteNoise.ar(0.5)*EnvGen.ar(Env([1,1,0],
[(f1+f2)/SampleRate.ir,0.0]));
    dummyenv = EnvGen.ar(Env([0, 0], [5],
[0]), doneAction:2);
    sig = TwoTube.ar(source, depth. linlin(0,
1, -1, 1),0.999,f1,f2);
    sig = Mix(sig);
    OffsetOut.ar(out, DirtPan.ar(sig,
~dirt.numChannels, pan));
}).add;
);
```

Waveblender

Waveblender è un sintetizzatore wavetable formato da quattro tabelle interpolabili e un waveshaper. I buffer delle tabelle non contengono file audio, ma vengono riempiti di forme d'onda ottenute costruendo segmenti d'inviluppo:

```
(
```

```
~tabelle = Buffer.allocConsecutive(4, s,
4096);

~env1 = Env(
    [0.000001, 0.5, -0.5, 0.8, -0.8, 1, -1,
0.000001],
    [1, 0.75, 1.25, 0.75, 1.25, 0.5, 2],
    [-5, -3, 8, -11, 20, -25, 4 ]);
~sig1 = ~env1.asSignal(2048);
~wt1 = ~sig1.asWavetable;
~tabelle[0].loadCollection(~wt1);

~env2 = Env(
    [0.000001, 0.75, -0.35, 0.9, -0.25, 0.6,
-0.8, 0.000001],
    [1, 1.5, 0.75, 0.25, 1.5, 0.75, 1.75],
    [5, 3, -8, 11, -20, 25, -4 ]);
~sig2 = ~env2.asSignal(2048);
~wt2 = ~sig2.asWavetable;
~tabelle[1].loadCollection(~wt2);

~env3 = Env(
    [0.000001, -0.25, 0.5, -0.75, 0.65, -1, 1,
0.000001],
    [0.75, 1, 1.5, 0.5, 1, 0.75, 1.75],
    [5, -2, 10, -13, 15, -4, 9 ]);
~sig3 = ~env3.asSignal(2048);
~wt3 = ~sig3.asWavetable;
~tabelle[2].loadCollection(~wt3);

~env4 = Env(
    [0.000001, 0.15, -0.75, 0.5, -0.35, 1, -1,
0.000001],
```

```
    [0.25, 1.5, 1, 0.25, 1.75, 0.25, 2.5],
    [20, -0.5, 9, -15, 20, 8, -2.1 ]);

~sig4 = ~env4.asSignal(2048);
~wt4 = ~sig4.asWavetable;
~tabelle[3].loadCollection(~wt4);
);
```

In questo modo vengono allocati quattro buffer e sono specificati quattro inviluppi differenti. Gli inviluppi vengono convertiti in formato wavetable dal metodo *.asWavetable*.
Anche il waveshaper non fa uso di file audio, ma contiene una forma d'onda generata dalla somma di armoniche ad ampiezza casuale. Perciò il comportamento del waveshaper cambia ogni volta che la SynthDef viene valutata.

```
(
~forma = Env([-1, 1], [1],
[0]).asSignal(2049);
~segnale = (Signal.sineFill(2049, (0!3) ++ [0,
0, 0, 1, 1, 1].scramble, {rrand(0, 2pi)} !9) /
4; );
~forma = ~forma + ~segnale;
~forma = ~forma.normalize;
~forma = ~forma.asWavetableNoWrap;
~formashaping = Buffer.loadCollection(s,
~forma);
);
```

La SynthDef presenta i parametri canonici di un sintetizzatore (nota, pan, attacco, rilascio) e presenta inoltre il parametro *pos* che permette di controllare la posizione di lettura tra le quattro tabelle, scorrendo tra l'una e l'altra

liberamente. La variabile *displace* viene usata per creare sette voci parallele del sintetizzatore, la cui variazione in termini frequenziali è espressa dal parametro *depth*.
Inoltre Waveblender presenta la possibilità di produrre altezze con qualunque temperamento equabile. Il parametro *tuning* definisce mediante un numero intero il numero di divisioni eque dell'ottava. Ad esempio se il parametro *tuning* è uguale a dodici, il valore 60 applicato al parametro *note* sarà uguale al do centrale del pianoforte; se invece *tuning* è uguale a diciannove, la nota 60 corrisponderà al terzo grado della quarta ottava in temperamento 19-EDO.

```
SynthDef (\waveblender, {
    |out, pan, note = 60, tuning = 19, depth =
1, atk = 0.001, rel = 0.5, pos = 0|
    var sig, env, octave, freq, displace,
bufnum;
    bufnum = ~tabelle[0].bufnum;
    octave = ((note/tuning)-5).trunc(1);
    freq = [440 * (pow(2, octave)) * (pow(2,
((mod(note, tuning))/tuning)))];
    displace = depth.wrap(1, 20);
    sig = VOsc.ar( EnvGen.ar(Env([bufnum,
bufnum + pos.linlin(0, 1, 0, 3), bufnum],
[atk, rel])),
[freq, freq + (0.1 * displace), freq -(0.1 *
displace), freq + (0.3 * displace), freq -(0.3
* displace), freq + (0.5 * displace), freq -
(0.5 * displace)]);
    sig = Splay.ar(sig);
    env = EnvGen.ar(Env([0.0001, 1, 0.0001],
[atk, rel], [5, -5]), doneAction: 2);
    sig = sig * env;
    sig = Shaper.ar(~formashaping, sig);
    sig = LeakDC.ar(sig);
```

```
    sig = Mix.ar(sig);
    OffsetOut.ar(out, DirtPan.ar(sig,
~dirt.numChannels, pan, env));
 }).add);
```

Vortex

Vortex è un sintetizzatore che combina tecniche granulari e FFT per creare tessiture lunghe e complesse. Dal momento che è pensato per produrre eventi di lunga durata, su TidalCycles viene attivato mediante la funzione *once*, che permette di riprodurre un evento una volta sola, senza che esso si trovi all'interno di un pattern ripetuto circolarmente.

In Vortex un treno di impulsi pseudo-casuali viene usato come trigger di un sintetizzatore granulare FM, il cui output viene convertito nel dominio spettrale e trasformato mediante algoritmi di diffusione, freezing e variazione delle magnitudini; successivamente lo spettro viene riconvertito in suono e viene processato in parallelo da un riverbero algoritmico i cui valori vengono modulati da oscillatori pseudo-casuali.

Lo strumento è pensato per produrre suoni complessi e caratteristici chiedendo all'esecutore di inserire soltanto quattro parametri: posizione stereofonica, valore d'attacco, valore di rilascio e densità dei grani. Il risultato sonoro è ogni volta diverso, in quanto la SynthDef contiene numerosi generatori casuali i cui valori vengono aggiornati a ogni evento richiesto.

```
(
SynthDef(\vortex, { arg out, pan, atk = 1, rel
= 5, grainmax = 100;
    var  trig, dur, freq, sound, modfreq,
chain, freeze, drywet, env, rev;
    trig = Dust.kr(EnvGen.ar(Env([1, grainmax,
1],[atk, rel])));
    dur = EnvGen.ar(Env([0.1,0.01,0.2],[rel,
atk]));
```

```
    env = EnvGen.ar(Env([0.0001, 1, 0.0001],
[atk, rel], [-1, -2]), doneAction: 2);
    //freq = [80, 121, 235, 353, 489, 528,
615, 722, 834, 1012];
    freq = ExpRand(150.0, [3000.0, 2500.0,
2800.0, 2200.0]);
    freq.postcs ;
    modfreq = freq * (1.rrand(3.1) + 1);
    sound = GrainFM.ar(2, trig, dur, freq,
modfreq, LFNoise0.ar(2, mul:0.2, add:0.7)) *
0.02;
    sound = Splay.ar(sound);
    sound = (sound * 3.5).tanh;
     chain = FFT(LocalBuf([2048, 2048]),
sound);
    chain = PV_Diffuser(chain, trig);
    freeze = PV_Freeze(chain, trig);
    freeze = PV_MagDiv(chain, freeze);
    freeze = PV_Freeze(freeze, trig);
    chain = IFFT(chain);
    freeze = IFFT(freeze);
    drywet = XFade2.ar(freeze, chain,
LFNoise1.ar(3).range(-0.8, 1));
    rev = JPverb.ar(drywet, 10, 0.15, 0.5,
0.05, 0.01, 0.2, 0.7 , 0.9, 0.8, 250, 1500);
    drywet = XFade2.ar(drywet, rev,
LFNoise1.ar(3.5).range(-0.8, 0.8));
    drywet = (drywet * env).tanh;
    drywet = LeakDC.ar(drywet.tanh);
    drywet = mix.ar(drywet);
    OffsetOut.ar(out, DirtPan.ar(drywet,
~dirt.numChannels, pan, env));
    }).add;
```

```
);
```

Flutter

Analogamente a Vortex, Flutter è un sintetizzatore programmato per la creazione di lunghe tessiture complesse mediante la combinazione di tecniche granulari e FFT. In questo caso il generatore sonoro è un rumore rosa che viene granulato in tempo reale con parametri gestiti da LFO pseudo-casuali. Dunque il segnale viene trasformato in matrice FFT e viene applicato un algoritmo di freezing. L'output finale è un bilanciamento tra il segnale originale e il segnale processato tramite FFT; si osservi che il segnale processato è in ritardo di 2048 campioni rispetto all'originale, producendo in questo modo cancellazioni di fase e movimenti timbrici dinamici.

```
(
SynthDef(\flutter, { arg out, pan, atk = 1.5,
rel = 7, fspeed = 3;
    var sound, rate, chain, drywet, env;
    env = EnvGen.ar(Env([0.0001, 1, 0.0001],
[atk, rel], [1, -3] ), doneAction: 2);
    rate = LFNoise1.ar(fspeed).exprange(0.25,
11);
    sound = PinkNoise.ar([0.89998, 0.89999]);
    sound = MonoGrain.ar(sound, 0.1, [fspeed +
rate, fspeed + rate + 0.01], 0.5);
    chain = FFT(LocalBuf([2048, 2048]),
sound);
    chain = PV_Freeze(chain, 0.1);
    chain = IFFT(chain);
    drywet = chain * SinOsc.ar(11);
    drywet = XFade2.ar(chain, drywet,
rate.linlin(0.25, 11, -1, 1));
    drywet = LeakDC.ar(drywet * env * 4).tanh;
    drywet = Mix.ar(drywet);
```

```
    OffsetOut.ar(out, DirtPan.ar(drywet,
~dirt.numChannels, pan, env));
    }).add;
);
```

Bee

Bee è un sintetizzatore ispirato al comportamento degli sciami delle api. Sebbene si tratti di un'ispirazione del tutto metaforica, questa SynthDef sperimenta le similarità tra il comportamento di LFO basati sul moto browniano e il movimento degli apidi.

Il valore di frequenza ottenuto dal moto browniano è applicato a una forma d'onda dinamica ed estremamente complessa, ottenuta attraverso la distorsione, il folding, l'inversione e la modulazione della forma d'onda attraverso modulatori casuali e caotici. Dunque la forma d'onda ottenuta viene processata dall'emulazione di un filtro ladder che ne riduce l'energia sulle alte frequenze; la frequenza di taglio del filtro è anch'essa controllata da LFO pseudo-casuali.

Successivamente il segnale viene convertito in spettro e trasformato dal processore FFT *PV_ConformalMap* che applica un algoritmo di phase vocoding trasformando il piano complesso; il *conformal mapping* produce degli artefatti spettrali che rendono imprevedibile il comportamento di questo sintetizzatore.

```
(
SynthDef(\bee, { arg out, pan, atk=5, rel=10,
beedust = 30;
    var freq, trig, signal, signal2, signal3,
env, chain;
    env = EnvGen.ar(Env([0.0001, 1, 0.0001],
[atk, rel], [-1, 2]), doneAction: 2);
    trig = Dust.ar([beedust, beedust -1 ]);
```

```
    freq = TBrownRand.ar(Rand(280, 320) +
LFNoise1.ar(2).range(-3, 3), 850, 35, 3,
trig);
    signal= SinOsc.ar(Lag.ar(freq,
0.05)).ring3(SinOsc.ar(LFTri.ar(0.0625).range(
2.5,9)))  *  0.85;
    signal2 =
signal.sqrdif( SinOsc.ar(LFNoise1.ar(11)));
    signal2 = signal2.moddif(PinkNoise.ar(0.9)
* SinOsc.ar(100) * 0.2);
    signal2 = signal2.distort;
    signal = signal2.scaleneg(signal);
    signal =
signal.scaleneg(HenonL.ar(SampleRate.ir/64,
0.7, 0.7, 0.1, 0.3));
    signal = signal.fold2(Saw.ar(freq *
Rand(0.9, 1.1))) * 0.5;
    signal = signal.clip2(signal2);
    signal3 = signal;
    signal = MoogLadder.ar(signal, freq *
LFNoise1.ar(11).range(3.8, 7), 0.5);
    signal = signal.softclip * 7;
    chain = FFT(LocalBuf([4096, 4096]),
signal);
    chain = PV_ConformalMap(chain, Rand(0.01,
1.0) * LFNoise1.ar(11).range(1.0, 2.0),
Rand(0.01, 5.0) * LFNoise1.ar(10).range(1.0,
2.0));
    signal = IFFT(chain);
    signal = (signal * 1.8).tanh;
    signal = LeakDC.ar(signal * env);
    signal = Mix.ar(signal);
```

```
    OffsetOut.ar(out, DirtPan.ar(signal,
~dirt.numChannels, pan, env));
}).add
);
```

Diffract

Diffract è un processore FFT che scala e trasla i bin dello spettro del segnale audio in ingresso. È costruito a partire dalla UGen *PV_Bin_Shift*. I parametri di controllo sono *diffract* (fattore di scalamento), *diffshift* (fattore di traslazione) e *diffmix* (dry/wet).

```
(
~dirt.addModule('diffract', { |dirtEvent|
    dirtEvent.sendSynth('diffract' ++
~dirt.numChannels,
        [
            diffract: ~diffract,
            diffshift: ~diffshift,
            diffmix: ~diffmix,
            out: ~out
        ]
    )
}, { ~diffract.notNil or: {~diffshift.notNil
or: {~diffmix.notNil}}})
);

(
SynthDef("diffract" ++ ~dirt.numChannels, { |
out, diffract = 1,
    diffshift = 1, diffmix = 0.5|
    var signal, chain, drywet, clean;
    signal = In.ar(out, ~dirt.numChannels);
    clean = signal;
```

```
    chain = signal.asArray.collect { |x|
FFT(LocalBuf([2048, 2048]), x) };
    chain = PV_BinShift(chain, diffract,
diffshift, 1);
    chain = IFFT(chain);
    drywet = XFade2.ar(signal, chain,
diffmix.linlin(0, 1, -1, 1));
    ReplaceOut.ar(out, drywet)
    }
  ).add
);
```

Invert

Invert è un semplice processore FFT che inverte i valori di magnitudine dei bin dello spettro. I valori di ampiezza delle frequenze basse sono applicati alle frequenze alte e viceversa. L'effetto produce suoni estremamente intensi nell'intervallo tra i 10 e i 20kHz, poiché la metà dei bin FFT occupano questo spazio frequenziale. Perciò può essere necessario usare un filtro passa-basso per evitare eventi sonori eccessivamente acuti.

```
(
~dirt.addModule('invert', { |dirtEvent|
    dirtEvent.sendSynth('invert' ++
~dirt.numChannels,
        [
            invert: ~invert,
            out: ~out
        ]
    )
}, { ~invert.notNil})
);

(
```

```
SynthDef("invert" ++ ~dirt.numChannels, { |
out, invert = 0.1|
    var signal, chain, drywet, clean;
    signal = In.ar(out, ~dirt.numChannels);
        clean = signal;
    chain = signal.asArray.collect { |x|
FFT(LocalBuf([512, 512]), x) };
    chain = PV_Invert(chain);
    chain = IFFT(chain);
    drywet = XFade2.ar(signal, chain,
invert.linlin(0, 1, -1, 1));
    ReplaceOut.ar(out, drywet)
    }
  ).add;
);
```

Ringshape

Ringshape è un processore di segnale che combina waveshaping e modulazione ad anello. Ha un unico parametro che controlla la frequenza del modulatore ad anello, mentre il waveshaper è applicato su un valore fisso; infatti la variazione timbrica della forma d'onda è data dalla dinamica interna dei suoni processati.

```
(
~dirt.addModule('ringshape', { |dirtEvent|
    dirtEvent.sendSynth('ringshape' ++
~dirt.numChannels,
        [
            out: ~out,
            ringshape: ~ringshape
        ]
    )
}, {~ringshape.notNil})
```

```
);

(
SynthDef("ringshape" ++ ~dirt.numChannels, { |
out, ringshape|
    var signal;
    signal = In.ar(out, ~dirt.numChannels);
    signal = SineShaper.ar(signal, 0.7) *
0.85;
    signal = DiodeRingMod.ar(signal,
SinOsc.ar(ringshape));
    signal = LPF.ar(signal, 10000);
    signal = LeakDC.ar(signal);
    ReplaceOut.ar(out, signal)
    }
).add
);
```

Tanh

Tanh applica il calcolo della tangente iperbolica sul segnale in ingresso. Dal momento che il dominio di una tangente iperbolica è compreso tra -1 e 1, tanh si comporta come un limiter. Quando il segnale in input eccede di molto l'intervallo tra -1 e 1, oltre a limitarne il range, la tangente iperbolica modifica la forma d'onda aggiungendo armoniche. In questo senso Tanh è in grado di comportarsi come un waveshaper.

L'unico parametro di questo semplice processore è il fattore di moltiplicazione dell'ampiezza del segnale in ingresso. Quando il valore di moltiplicazione è molto alto, ne risulta un aumento del livello medio corrispondente a una maggior sensazione di loudness. Perciò Tanh assolve anche alla funzione di compressore per usi creativi.

```
(
~dirt.addModule('tanh', { |dirtEvent|
```

```
    dirtEvent.sendSynth('tanh' ++
~dirt.numChannels,
        [
            tanh: ~tanh,
            out: ~out
        ]
    )
}, {~tanh.notNil})
);

(
SynthDef("tanh" ++ ~dirt.numChannels, { |out,
tanh|
    var signal;
    signal = In.ar(out, ~dirt.numChannels);
    signal = signal * tanh;
    signal = signal.tanh;
    ReplaceOut.ar(out, signal)
    }).add
);
```

Tantanh

Questo processore si comporta in maniera analoga al precedente, tuttavia prima di calcolare la tangente iperbolica, esso calcola anche la tangente. Il calcolo della tangente produce frequentemente numeri ben più alti dell'intervallo d'ampiezza consentito; l'uso della tangente iperbolica applicata in serie permette di far tornare il segnale in valori compresi tra -1 e 1. Questo processamento, benché formato da semplici operatori trigonometrici, è in grado di produrre distorsioni non-lineari che non richiedono una compensazione del livello in uscita. Quando il segnale in ingresso viene moltiplicato oltre trecento volte il suo livello originale, l'output tende a diventare estremamente rumoroso, generando segmenti d'onda rettangolari dal periodo variabile.

```
(
~dirt.addModule('tantanh', { |dirtEvent|
    dirtEvent.sendSynth('tantanh' ++
~dirt.numChannels,
        [
            tantanh: ~tantanh,
            out: ~out
        ]
    )
}, {~tantanh.notNil})
);

(
SynthDef("tantanh" ++ ~dirt.numChannels, { |
out, tantanh|
    var signal;
    signal = In.ar(out, ~dirt.numChannels);
    signal = signal * tantanh;
    signal = signal.tan.tanh;
    signal = LeakDC.ar(signal);
    ReplaceOut.ar(out, signal)
    }).add
);
```

4.2 Tecniche stocastiche

Come osservato nella sezione 1.2, l'uso di tecniche probabilistiche è alla radice dello sviluppo della musica algoritmica. Un buon linguaggio di live coding deve permettere l'impiego di un vasto numero di strategie di controllo del caso.

TidalCycles presenta numerose funzioni per gestire segnali pseudo-casuali, flussi numerici probabilistici e strutture stocastiche. Inoltre, vista la natura puramente funzionale di Haskell, è possibile combinare le funzioni per ottenere funzioni composite che mettano in relazione diversi gradi di aleatorietà e determinismo.

TidalCycles fa uso di due generatori di segnale pseudo-casuale, *rand* e *irand*. In entrambi i casi si tratta di segnali continui il cui valore viene campionato nel momento in cui l'interprete richiede la compilazione di un evento sonoro; *rand* genera numeri decimali compresi tra 0 e 1, mentre *irand* produce numeri interi compresi tra 0 e x-1, dove x è uguale all'unico argomento della funzione.

```
d1 $ s "bd*8" # n (irand 8) # pan rand
```

In questa linea di codice *irand* viene usato per riprodurre diversi campioni presenti nella cartella *bd*, compresi rispettivamente tra il campione di indice 0 e quello di indice 7; l'applicazione di *irand* per la selezione delle sorgenti sonore può essere utile per indurre variazioni timbriche in una linea percussiva, o per creare linee di *micromontage* dove centinaia di file audio molto brevi e simili tra loro sono riprodotti velocemente e in ordine casuale. L'oscillatore *rand* è invece usato in questo caso per generare una posizione stereofonica casuale per ogni evento sonoro prodotto; naturalmente può essere fruttuosamente impiegato per la modulazione di qualunque parametro.

Per la permutazione casuale degli elementi interni a un pattern si usano le funzioni *scramble* e *shuffle*. Algoritmi di *scrambling* sono ampiamente diffusi in tutti i linguaggi di programmazione. La loro funzione è quella di scambiare gli indici di array e liste per ottenere combinazioni casuali degli elementi:

```
d1 $ sound $ scramble "uno due tre"
```

La funzione *scramble* crea a ogni ciclo una permutazione casuale dei campioni del pattern, perciò può produrre le seguenti combinazioni: “uno due tre”, “due uno tre”, “tre uno due”, “uno tre due”, “due tre uno”, “tre due uno”, ma anche ripetere lo stesso elemento più di una volta, creando ad esempio “due due uno”, “tre due tre” o “uno uno uno”.
Shuffle si comporta nello stesso modo, ma ammette soltanto permutazioni esatte del pattern, cioè solamente risultati che contengono tutti e tre gli elementi in un qualunque ordine.

Per applicare casualmente una funzione scelta tra un insieme di funzioni possibili si usa la funzione *spreadr*:

```
d1 $ spreadr ($) [slow 2, fast 2, (# speed
0.5), id] $ s "bd*4"
```

A ogni ciclo *spreadr* seleziona una delle funzioni contenute tra gli argomenti. In questo caso potrebbe dimezzare la velocità del pattern, raddoppiarla, dimezzare la velocità di riproduzione e il pitch dei campioni, oppure lasciare il pattern inalterato.

TidalCycles fa uso inoltre dei g*ate di Bernoulli,* detti anche trigger probabilistici. In un gate di Bernoulli c’è una certa percentuale di possibilità che un dato evento avvenga. Solitamente implementato applicando un operatore logico (come <, >, <= o >=) al risultato di un generatore pseudo-casuale, in TidalCycles può essere applicato con le funzioni *degradeBy* e *sometimesBy*. La prima è utilizzata per gestire la densità degli eventi interni a un pattern, la seconda influenza la possibilità che una data funzione agisca durante un ciclo:

```
d1 $ degradeBy 0.65 $ sometimesBy 0.25 (slow
2) $ s "bd*8"
```

Nell’esempio è presente il campione *bd* ripetuto otto volte ogni ciclo, ma nel 65% dei casi l’evento viene saltato; l’effetto percettivo è interessante, in

quanto i pattern risultanti cambiano continuamente in maniera imprevedibile, ma mantengono implicitamente l'unità di suddivisione originaria. Contemporaneamente all'effetto di *degradeBy, sometimesBy* determina che c'è il 25% di probabilità che la velocità dimezzi, producendo una variazione metrica continua tra due valori di suddivisione. Dunque il pattern risultante è una linea costantemente variabile di quarti e ottavi che rispetta in ogni caso la griglia metrica degli ottavi.

La funzione *choose* consente invece di definire una lista di possibili argomenti per una funzione, lasciando a un algoritmo pseudo-casuale la selezione di un elemento per ogni ciclo:

```
d1 $ sound "bd*4" # speed (choose [1, 2,
0.5, 0.75, 1.25])
```

Nel pattern qui descritto, la velocità di riproduzione del campione *bd* varia casualmente tra i cinque valori scelti. Si osservi che questa tecnica corrisponde a uno dei princìpi di selezione enunciato da Göttfried Michael Koenig (cfr. sez. 1.2).
Se si volesse definire una percentuale di probabilità di scelta di ognuno degli elementi della lista, si potrebbe usare la funzione *wchoose*:

```
d1 $ sound "bd*4"
# speed (wchoose [(1, 0.25), (0.5, 0.35), (2,
0.15), (0.75, 0.1), (1.25, 0.15)])
```

In questo caso a ognuno dei valori della lista precedente è stato accoppiato un numero che ne descrive la percentuale di possibilità. La somma delle possibilità è uguale a 1.0. Tuttavia è rilevante specificare che è possibile inserire valori la cui somma è minore o maggiore di 1.0 e l'interprete di TidalCycles provvederà a normalizzare i rapporti; così facendo il performer non deve preoccuparsi di calcolare una somma decimale composta da numerosi addendi, alleggerendo il carico cognitivo del processo.

Tutte le funzioni precedentemente illustrate possono essere combinate per creare funzioni dal comportamento più complesso. Si osservi ad esempio questa definizione dalla mia libreria:

```
let rangedBy x y p = somecyclesBy (rand - x)
                     (degradeBy y) $ p
```

La funzione *rangedBy* così descritta combina due funzioni probabilistiche e un segnale pseudo-casuale. Essa definisce che la possibilità che la densità di un pattern sia diminuita di y per la durata di un ciclo dipende dal valore casuale della funzione *rand* a cui è sottratto il valore x di offset negativo.

```
d1 $ rangedBy 0.25 0.85 $ s "bd*4"
```

In questo esempio la funzione enunciata determina che c'è una possibilità tra lo 0 e il 25% che a ogni ciclo venga applicata una diminuzione di densità dell'85 %.
Si osservi inoltre quest'altra funzione personalizzata:

```
let chooseDegrade n x y p = sometimesBy n
   (choose ([degradeBy x, degradeBy y])) $ p
```

Il fattore n definisce la possibilità che a ogni evento venga applicata una selezione casuale tra due valori di densità definiti dagli argomenti x e y:

```
d1 $ chooseDegrade 0.15 0.25 0.75 $ s "bd*4"
```

Nel 15% dei casi verrà applicato un gate di Bernoulli il cui valore è scelto casualmente tra due possibilità, il 25 e il 75%.

Strutture stocastiche più complesse possono essere ottenute tramite l'uso di catene di Markov di prima specie. Una catena di Markov di prima specie

definisce la probabilità di passaggio da un valore all'altro in base a un rapporto di peso basato su quanto è avvenuto nel passaggio di stato precedente. In questo caso è pertanto necessario scrivere una *matrice di transizione* a partire da un determinato stato iniziale; la funzione *markovPat* ne permette l'applicazione durante l'improvvisazione:

```
d1 $ s "bd*8"
# speed (fmap ([1,2,3]!!)
$ markovPat 8 1 [[2,4,1],[4 0 2],[2,3,3]])
```

La matrice di transizione di markovPat produce un pattern di 8 valori scelti tra tre stati possibili, a partire dal primo stato. Il pattern creato viene poi assegnato tramite la funzione *fmap* a tre valori di argomento possibili per la funzione *speed* applicata al pattern principale.
La scrittura di matrici di transizione più grandi è possibile, ma richiede quantitativi di tempo e carico cognitivo spesso incompatibili con l'improvvisazione musicale. Una strategia possibile è quella di preparare delle matrici di transizione e assegnarle a un nome, ad esempio *matrice_uno*, da richiamare successivamente. Ciononostante questo livello di complessità stocastica potrebbe non essere nemmeno necessario in un contesto di continua variazione dell'algoritmo da parte del performer.

Strutture stocastiche possono essere sfruttate anche nel dominio della sintesi. Iannis Xenakis ha teorizzato – e successivamente implementato – la sintesi stocastica, una tecnica in cui ogni periodo di una forma d'onda è formato da un determinato numero di segmenti di durata e ampiezza variabile in funzione di leggi di distribuzione (Luque, 2009). Oscillatori stocastici chiamati *Gendy* sono stati implementati sotto forma di UGen da Nick Collins sulla base della programmazione originale di Marie-Helena Serra.
Un oscillatore stocastico presenta i seguenti parametri:

- Modello di distribuzione delle probabilità (scelto tra lineare, Cauchy, arco-seno, esponenziale, sinusoidale, iperbole-coseno, ecc.)
- Numero di segmenti per periodo possibili

- Numero di segmenti per periodo utilizzati (il cui valore è modulabile nel tempo)
- Valore minimo di frequenza del periodo prodotto dai segmenti
- Valore massimo di frequenza del periodo prodotto dai segmenti
- Fattore d'influenza della distribuzione di probabilità sui valori d'ampiezza
- Fattore d'influenza della distribuzione di probabilità sui valori di durata dei segmenti
- Fattore di scalamento d'ampiezza
- Fattore di scalamento di durata

Il tipo di timbri producibili da un oscillatore stocastico varia enormemente in funzione della variazione dei parametri; in ogni caso, se i valori minimo e massimo di frequenza del periodo non coincidono, si avranno delle variazioni continue di altezza del suono in grado di produrre micro-glissandi il cui comportamento varia a seconda della funzione di distribuzione delle probabilità.
Il numero di segmenti per periodo definisce la complessità delle forme d'onda possibili: maggiore è il numero di segmenti, maggiore è la probabilità di creare timbri inarmonici, in quanto la presenza di picchi variabili interni alla forma d'onda produce dei rapporti armonici irregolari e costantemente variabili, in maniera analoga a quanto viene percepito ascoltando generatori di rumori pseudo-casuali.
Se opportunamente programmati sotto forma di SynthDef, gli oscillatori stocastici possono essere usati in live coding. Seguono tre sintetizzatori che ho programmato a partire dalla UGen "Gendy1".

Xenbass

Xenbass è un sintetizzatore che fa uso di due oscillatori stocastici all'interno di un convenzionale algoritmo di sintesi sottrattiva; un filtro passa basso del secondo ordine controllato da un inviluppo consente di generare i caratteristici suoni di basso della club music. Tuttavia, dal momento che gli oscillatori sono stocastici, il segmento di rilascio del suono ha un comportamento diverso ogni volta, producendo delle "code" caotiche. I

parametri dei due oscillatori sono uguali eccetto per quanto riguarda l'escursione massima nella variazione di frequenza, producendo effetti di cancellazione di fase, ma anche micro-cluster armonici. Un argomento chiamato *depth* regola la variazione nell'escursione del valore di frequenza dei due oscillatori. Il sintetizzatore è inoltre configurato per produrre qualunque tipo di temperamento a divisione equa dell'ottava.

```
(
SynthDef(\xenbass, { arg out, pan, note,
tuning = 19, atk = 0.001, rel = 1, depth = 1;
    var signal, env, freqenv, octave, freq;
    octave = ((note/tuning)-5).trunc(1);
    freq = [440 * (pow(2, octave)) * (pow(2,
((mod(note, tuning))/tuning)))];
    freqenv = EnvGen.ar(Env([freq, freq * 6,
freq], [atk, rel], 'exp'));
    env = EnvGen.ar(Env([0.0001, 1, 0.0001],
[atk, rel], [-5, -4]), doneAction: 2);
    signal = Gendy1.ar(1, 5, 1, 1, freq, [freq
+ depth, freq + depth + 1 ], 0.5, 0.6, 6);
    signal = LPF.ar(signal, freqenv) * env;
    signal = LeakDC.ar(signal);
    signal = Mix.ar(signal);
    OffsetOut.ar(out, DirtPan.ar(signal,
~dirt.numChannels, pan, env));
    }).add;
);
```

Xenharp

Progettato con una struttura analoga a Xenbass, Xenharp è pensato per creare suoni ad alta frequenza e lungo decadimento. A questo scopo gli inviluppi hanno fattori di curvatura esponenziali e logaritmici differenti da Xenbass e il filtro è un filtro risonante con un elevato coefficiente Q.

```
(
SynthDef(\xenharp, { arg out, pan, note,
tuning = 19, atk = 0.001, rel = 1, depth = 1;
    var signal, env, octave, freq, filterenv;
    octave = ((note/tuning)-5).trunc(1);
    freq = [440 * (pow(2, octave)) * (pow(2,
((mod(note, tuning))/tuning)))];
    filterenv = EnvGen.ar(Env([freq, 10000,
freq ], [atk, rel], [-4, -1]));
    env = EnvGen.ar(Env([0.0001, 1, 0.0001],
[atk, rel], [-10, -1]), doneAction: 2);
    signal = Gendy1.ar(3, 5, 1, 1, [freq, freq
+ 1], [freq + depth + 0.5, freq + depth],
0.0005, 0.0005, 25);
    signal = RLPF.ar(signal, filterenv, 0.95);
    signal = LeakDC.ar(signal * env);
    signal = Mix.ar(signal);
    OffsetOut.ar(out, DirtPan.ar(signal,
~dirt.numChannels, pan, env));
}).add;
);
```

Xenoise

Xenoise usa un oscillatore stocastico all'interno di una complessa architettura di sintesi che impiega modulazioni d'ampiezza, riverberi algoritmici controllati da oscillatori pseudo-casuali, morphing spettrale e trasformazioni FFT. È pensato per produrre suoni inarmonici e aspri di lunga durata, dal comportamento ogni volta differente in funzione di soli tre argomenti: attacco, rilascio e numero di segmenti impiegati per periodo.

```
(
SynthDef( \xenoise,
```

```
{ arg out, pan, atk=3, rel=10, nseg = 0.5;
    var signal, env, chain, trig, rvalue, rev,
revchain;
    rvalue = Rand(5.5, 14);
    trig = Dust.ar(rvalue);
    nseg = nseg.linlin(0.0, 1.0, 3, 15);
    env = EnvGen.ar(Env([0.0001, 1, 0.0001],
[atk, rel], [-5, 0.5]), doneAction: 2);
    signal = Gendy1.ar(3, 0.15, 0.001, 0.001,
Rand(300, 450), Rand(800, 1400), 0.5, 0.75,
[nseg, nseg - 1 ]);
    signal = signal * SinOsc.ar(rvalue);
    rev = Greyhole.ar(signal, ExpRand(2.1, 4),
0.65, 0.65, 0.75, 0, 0);
    chain = FFT(LocalBuf([2048, 2048]),
signal);
    revchain = FFT(LocalBuf([2048, 2048]),
rev);
    chain = PV_PhaseShift(chain,
LFTri.ar(rvalue).range(1, 180));
    chain = PV_Morph(chain, revchain,
LFNoise1.ar(rvalue).range(0, 0.85));
    signal =  IFFT(chain);
    signal = signal.softclip.distort;
    signal = LeakDC.ar(signal * env);
    signal = Mix.ar(signal);
    OffsetOut.ar(out, DirtPan.ar(signal,
~dirt.numChannels, pan, env));
}
).add;
);
```

4.3 Tecniche ritmiche e contrappuntistiche

La logica circolare di TidalCycles permette di elaborare contrappunti ritmici estremamente sofisticati con facilità. È il design del software stesso a suggerire all'utente una modalità improvvisativa incentrata su relazioni e divergenze ritmiche tra le linee musicali.
Nella libreria di funzioni ve ne sono numerose dozzine preposte ad assolvere a tecniche ritmiche e contrappuntistiche come rotazioni, trasposizioni, canoni, inversioni, poliritmi, polimetrie, ritmi di Björklund, accelerandi e decelerandi compositi, riduzioni ed espansioni delle durate.
Sebbene le capacità costruttive del linguaggio siano state già dimostrate nelle sezioni precedenti, si osservi ancora una volta quanto avviene in questo esempio:

```
d1
$ superimpose ((slow 1.5) . (off (+7)))
$ sometimesBy 0.25 (slow 2)
$ degradeBy 0.15
$ every 9 (fast 2)
$ whenmod 16 14 ((ply 3) . (degradeBy 0.85))
$ note "{60 68 63 70 63, 55 50 56 48, 74 80
79, 36 ~ ~ 43 ~ ~ ~ }"
# sound "waveblender"
# bpm 75
```

Il pattern consta di quattro linee polimetriche sulla scala giapponese *hirajoshi* (do, re, mi bemolle, sol, la bemolle) applicate su quattro diverse ottave; le linee sono formate rispettivamente da sette, cinque, quattro e tre elementi, perciò necessitano di 420 (7 x 5 x 4 x 3) ripetizioni del pattern prima di ritornare nella posizione iniziale. Al pattern sono applicate variazioni periodiche e condizionali. Ogni sedici cicli, per la durata di due cicli, ogni evento sonoro viene ripetuto tre volte nello stesso spazio metrico, ma c'è una probabilità dell'85% che un evento non venga suonato (funzioni *whenmod, ply* e *degradeBy*).

Ogni nove cicli, per il tempo di un ciclo le durate di tutto il pattern dimezzano (funzioni *every* e *fast*). In tutte e quattro le linee, c'è il 15% di possibilità che un evento venga saltato (funzione *degradeBy*). Nel 25% degli eventi la velocità complessiva del pattern viene dimezzata (funzioni *sometimesBy* e *slow*). Tutto il pattern, compreso delle variazioni indotte dalle funzioni descritte, è contrappuntato dallo stesso identico pattern trasposto di una quinta giusta, le cui durate sono più lunghe del 150% (funzioni *superimpose, slow* e *off*).
Il risultato di questo algoritmo produce otto linee melodiche coerenti tra loro in quanto appartenenti alla stessa scala, relazionate ritmicamente in maniera stabile, ma continuamente mutevoli. La possibilità che nel giro di una decina di minuti un ciclo si ripeta in forma uguale a uno precedentemente ascoltato sono estremamente basse.

Sebbene il campo di possibilità ritmico-contrappuntistiche di TidalCycles sembri già piuttosto vasto, è auspicabile un ulteriore ampliamento tramite la creazione di proprie funzioni in Haskell.
Il sistema di sequenziamento fondato sulla variazione di un pattern ciclico può dare luogo a molti sviluppi, eppure in certi in casi il performer può avere la volontà di comporre e improvvisare mediante astrazioni caratterizzate dalla più vasta periodicità, ovvero sfruttando pattern algoritmici che sin da principio si sviluppano su un intervallo temporale molto più ampio di un singolo ciclo. Partendo da quest'idea ho deciso di implementare in Haskell la tecnica dei *sieves* inventata da Iannis Xenakis (1992).
Il *sieve* è un algoritmo di organizzazione ritmica – o più generalmente parametrica – fondato sull'uso della logica booleana applicata all'intersezione di rapporti temporali periodici. Esso consta generalmente di un contatore, un'operazione di modulo (il resto di una divisione) e uno o più operatori logici (AND, OR, NOR, XOR, NAND, ecc.). Il contatore incrementa il suo valore a ogni ciclo temporale; questo valore viene applicato come dividendo di un'operazione modulo il cui divisore è stato precedentemente definito; se il resto della divisione corrisponde a 0 o a un altro numero prefissato, allora il risultato del calcolo è positivo e viene rappresentato dal valore 1; in caso

contrario il risultato è 0. L'operazione di modulo viene svolta in parallelo da diversi divisori con eventuali fattori di offset, tutti sincronizzati dal contatore. I risultati rappresentati sotto forma di 0 e 1 sono comparati tra loro dagli operatori logici il cui output determina il pattern ritmico.
Ad esempio se avessimo due operazioni di modulo parallele, la prima con un divisore uguale a 7 e la seconda con un divisore uguale a 2, e collegassimo i risultati delle operazioni a un operatore OR, avremmo un pattern che produce un evento sonoro ogniqualvolta il contatore è divisibile per 7 o per 2:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
_ _ X _ X _ X X X _ X  _  X  _  X  _  X
```

Se invece i due risultati fossero connessi da un operatore AND, l'evento sonoro sarebbe prodotto soltanto nel momento in cui entrambe le operazioni di modulo dessero un risultato uguale a 1:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
_ _ _ _ _ _ _ _ _ _ _  _  _  _  X  _  _
```

Questo tipo di logica permette di esprimere ritmi elaborati ottenuti da periodicità composite, la cui unità di suddivisione è costante. Si possono concatenare più divisori e più operatori booleani, ottenendo sequenze molto sofisticate. Il vantaggio di questa modalità organizzativa è che in grado di produrre pattern lunghi ma coerenti tra loro, per i quali è richiesto un input di dati piuttosto esiguo. La logica dei sieves si adatta perfettamente al funzionamento di TidalCycles, poiché entrambi si basano su un'unità temporale ricorsiva; inoltre Haskell è un linguaggio piuttosto efficiente nella programmazione di operazioni logico-matematiche di questo genere.

A partire dall'implementazione dei sieves in linguaggio Python (Ariza, 2005) ho scritto una serie di funzioni per TidalCycles, ognuna delle quali permette di creare un sieve con operatori differenti.
Tutte le funzioni si basano sulla concatenazione di un'operazione di resto:

```
let restogen value offset amount  = [if (x
`mod` value) == 0 then 1 else 0 | x <- [(0 +
offset)..(amount + offset)]]
    gen x y z = restogen x y z
```

La funzione *gen* qui descritta viene poi applicata agli operatori. Ad esempio per l'operatore OR:

```
let sieveor val1 off1 val2 off2 d = listToPat
$ [if (gen val1 off1 d !! x) == 1 || (gen val2
off2 d !! x) == 1 then True else False | x <-
[0..d-1]]
```

L'operazione si svolge fino a un numero massimo di cicli, oltre il quale il pattern si ripete. La lista ottenuta viene convertita in un formato compatibile con i pattern di TidalCycles e viene organizzata temporalmente affinché la lunghezza del pattern sia una potenza di due:

```
let sor val1 off1 val2 off2 d p = slow
((fromIntegral d)/8) $ struct (sieveor val1
off1 val2 off2 d) $ p
```

Una volta enunciate queste tre funzioni, il sieve formato da un singolo operatore OR può essere utilizzato su TidalCycles:

```
d1 $ sor 11 0 8 3 256 # sound " bd"
```

Nell'esempio il sieve combina i numeri divisibili per 11 e i numeri che divisi per 8 danno come resto il numero 3, per una durata complessiva del pattern di 256 cicli oltre la quale la sequenza ricomincerà da capo. Allo stesso modo si possono produrre tutti i tipi di sieves. Un sieve che combina i valori inversi

del risultato tra la concatenazione di un OR, un AND e un altro OR è enunciato in questo modo:

```
let sieveinvorandor val1 off1 val2 off2 val3
off3 val4 off4 d = listToPat $ [if (gen val1
off1 d !! x) == 1 || (gen val2 off2 d !! x) ==
1 && (gen val3 off3 d !! x) == 1 || (gen val4
off4 d !! x) == 1 then False else True | x <-
[0..d-1]]

let sinvorandor val1 off1 val2 off2 val3 off3
val4 off4 d p = slow ((fromIntegral d)/8) $
struct (sieveinvorandor val1 off1 val2 off2
val3 off3 val4 off4 d) $ p
```

La lista di completa di tutti i sieves prodotti è troppo lunga per essere riportata, ma è disponibile sul mio GitHub (Olbos, 2020) e periodicamente aggiornata.

Conclusioni

Questa tesi ha tentato di delineare alcune direzioni possibili nella ricerca storica, musicologica e tecnologica del live coding, ponendo particolare attenzione sui recenti sviluppi della comunità Algorave. Nonostante negli ultimi anni l'argomento abbia cominciato a essere oggetto di discussione in ambito accademico e artistico, risulta evidente che quanto descritto sinora costituisce soltanto un punto di partenza per indagini più specifiche.

Dal punto di vista dell'analisi storico-sociologica sono state compiute numerose e approfondite ricerche. Tuttavia un'osservazione del contesto attuale mette in luce quanto le dinamiche storiche e sociali siano mutevoli e si prevede un'ampia evoluzione del live coding nei prossimi anni. È opportuno che tali processi continuino a essere osservati in un'ottica critica e storiografica, riflettendo sulle potenzialità dell'espansione della pratica dell'improvvisazione algoritmica a gruppi sociali sempre più ampi. Conseguentemente sarà necessario sviluppare modalità divulgative e didattiche commisurate ai vari contesti socio-culturali.

Per ciò che concerne lo studio delle implicazioni performative derivanti dal live coding, si osserva l'urgenza di comprendere al meglio il fenomeno indagandolo in chiave scientifica, psicologica e musicologica. Le ricerche menzionate nella tesi illustrano alcune possibili ipotesi di studio, ma di per sé non sono sufficienti a fornire un quadro fenomenologico chiaro sulla questione.
L'incremento della partecipazione nelle pratiche del live coding darà luogo a scenari musicologici estremamente differenziati che meriteranno indagini specifiche, facendo uso di aree della conoscenza tangenti o sinora del tutto parallele alla musica algoritmica.

Un campo di interesse che merita indubbiamente maggiori approfondimenti è lo studio delle relazioni multimediali messe in moto dalla proiezione del codice e dalle pratiche di video live coding. In questo senso la comunità dei

praticanti si è sinora approcciata al contesto audiovisivo in un'ottica essenzialmente naive. Il fatto che il live coding stia vivendo un tale progresso in concomitanza con lo sviluppo esponenziale delle tecnologie video può mettere in atto innumerevoli interrelazioni le cui implicazioni teoretiche, estetiche e percettive dovranno essere prese in esame.

Per quanto riguarda il piano tecnologico, il campo di possibilità è estremamente vasto. Come osservato più volte durante la trattazione, le tecnologie relative al live coding sono numerose e in continuo mutamento. Il compito del compositore o performer sarà quello di districarsi in uno scenario plurale e complesso, fatto di una molteplicità di linguaggi e librerie in costante aggiornamento, ma anche di limiti imposti dall'hardware o dal design degli strumenti.
Quanto trattato nei capitoli 3 e 4 si propone come una possibile direzione di sviluppo, aperta alle necessità di un compositore consapevole delle tecniche compositive tradizionali e dei mezzi elettroacustici; tuttavia rappresenta soltanto un esempio metodologico di quella che necessariamente si manifesta come una ricerca estremamente personale e veicolata dalle proprie volontà espressive; un ricerca, dunque, che sia consapevole dei mezzi tecnici e che abbia come forza motrice l'individuazione di nuove modalità creative in grado di formare nuovi significati.
Lo scopo è quello di fornire una mappa, un orientamento teorico e pratico aperto, che possa costituire un punto di partenza per quanti vorranno intraprendere la ricerca nel campo dell'improvvisazione musicale algoritmica.

Bibliografia

Anderson, D. P. and Kuivila, R., 1991. Formula: A Programming language for expressive computer music [online]. *Computer,* 24(7), 12-21.

Andreatta, M., 2013. *Musique algorithmique* [online]. Paris: Symétrie.

Ariza, C., 2005. The Xenakis Sieve as Object: A New Model and a Complete Implementation. *Computer Music Journal* [online]*,* 29 (2), 40-60.

Baalman, M., 2015. Embodiment of Code. In: *Proceedings of the International Conference on Live Coding (ICLC), Leeds, 13-15 July 2015.*

Baker, C., 2013. Open-source, Custom Interfaces and Devices with Live Coding in Participatory Performance. In: *Electronic Visualisation and the Arts (EVA 2013),* London.

Battisti, E., n.d. The Experimental Music Studio at The University of Illinois, 1958-1968. *University of Illinois at Urbana-Champaign* [online]. Available from: https://music.illinois.edu/ems-history [Accessed 3 August 2020].

Belet, B., 2008. Theoretical and Formal Continuity in James Tenney's Music. *Contemporary Music Review.* [online], 27(1): 23-45.

Bell, R., 2011. An Interface for Realtime Music Using Interpreted Haskell. In: *Proceedings of LAC 2011, Maynooth Ireland.*

Bell, R., 2013a. Considering Interaction in Live Coding through a Pragmatic Aesthetic Theory. In: *Proceedings of SI13, NTU/ADM Symposium on Sound and Interactivity, Nanyang Technological University, Singapore.*

Bell, R., 2013b. Pragmatic Aesthetic Evaluation of Abstractions for Live Coding. In: *Proceedings of Meeting N. 17 of the Japanese Society for Sonic Arts, Nagoya.*

Bell, R., 2014. Experimenting with a Generalized Rhythmic Density Function for Live Coding. In: *Proceedings of LAC 2014, Karlsruhe, Germany.*

Bell, R., 2020. *Renick Bell, live coding + text, May 14, 2020*. Available from: https://www.youtube.com/watch?v=fXuLsLV20bw&t=2505s [Accessed 20 August 2020].

Bischoff. J. and Brown, C., n.d. Experimental Music in the Bay Area. Available from: http://crossfade.walkerart.org/brownbischoff/ [Accessed 5 August 2020].

Blackwell, A.F. and Aaron, S., 2015. Craft Practices of Live Coding Language Design [online]. In: *Proceedings of the International Conference on Live Coding (ICLC), Leeds, 13-15 July.*

Blackwell, A. F. and Collins, N., 2005. The Programming Language as a Musical Instrument. In: *Proceedings of 17th Psychology of Programming Interest Group 2005, University of Sussex,* 120-130.

Boutwell, B., 2009. *The League of Automatic Music Composers,* 1978-1983. With John Bischoff, Jim Horton, Tim Perkis, David Behrman, Paul DeMarinis, and Rich Gold. New World Records 80671-2, 2007. *Journal of the Society for American music,* 3, 263-264.

Boyle, K., 2017. Phonological and Musical Loops in Live Coding Performance Practice. *Leonardo Music Journal* [online], 27 (1), 40-44.

Brown, A. R., 2006. Code Jamming. *M/C Journal* [online], 9(6).

Bullock, J., 2018. Designing Interfaces for Musical Algorithms. In: Dean, R. T. and McLean, A., eds. *The Oxford Handbook of Algorithmic Music* [online]. New York: Oxford University Press, 461-492.

Cárdenas, A., 2018. Mexico and India: Diversifying and Expanding the Live Coding Community. In: Dean, R. T. and McLean, A., eds. *The Oxford Handbook of Algorithmic Music* [online]. New York: Oxford University Press, 129-135.

Cárdenas, A., 2019. Street Code - live coding in public space [online]. In: *Proceedings of the Fourth International Conference on Live Coding, Medialab Prado, Madrid 19-21 January 2019,* 114 - 143.

Cascone, K., 2002. Laptop music - counterfaiting aura in the age of infinite reproduction. *Parachute* [online]. 107.

Cascone, K., 2004. Grain, Sequence, System [three levels of reception in the performance of laptop music]. *Intelligent Agent* [online], 4 (1, Winter 2004).

Cheshire, T., 2013. Hacking meets clubbing with the 'algorave'. *Wired* [online], 29 August 2013. Available from: https://www.wired.co.uk/article/algorave [Accessed 29 July 2020].

Cheung, M., 2019. Reflections on Learning Live Codings as as Musician. In: *Proceedings of the International Conference on Live Coding (ICLC), Madrid, 16-18 January 2019.*

Cipriani, A., 1996. Verso una tradizione elettroacustica? Appunti per una ricerca. In: *Musica/Realtà* [online], 49.

CLiC (Colectivo de Live Coders), 2020. *Eulerroom Equinox 2020* [online]. Available from: www.equinox.euleroom.com [Accessed 3 August 2020].

Clic, n.d. *CLiC - Colectivo de Live Coders.* Available from: https://colectivo-de-livecoders.gitlab.io/ [Accesse 7 August 2020].

Cocker, E., 2016. Kairotic Coding – Performing Thinking in Action. In: *Proceedings of the International Conference on Live Coding (ICLC), Hamilton, Canada, 12-15 October 2016*.

Coleman, G., 2016. Hacker. *In:* Peters, B., ed. *Digital Keywords: A Vocabulary of Information Society and Culture* [online]. Princeton: Princeton University Press.

Collins, N., 2011. Live Coding of Consequence. *Leonardo* [online], 44(3), 207-211.

Collins, N., 2018. Origins of Algorithmic Thinking in Music. In: Dean, R. T. and McLean, A., eds. *The Oxford Handbook of Algorithmic Music* [online]. New York: Oxford University Press, 81-93.

Collins, N., McLean, A., Rohrhuber, J., and Ward, A., 2003. Live coding in laptop performance. *Organised Sound* [online], 8(3), 321-330.

Computer Club, n.d. *ComputerClub.* Available from: https://computerclub.bandcamp.com/ [Accessed 7 August 2020]

Conditional, n.d. *Conditional.* Available from: http://www.conditional.club/ [Accessed 7 August 2020].

Cox, G., 2015. What Does Live Coding Know? In: *Proceedings of the International Conference on Live Coding (ICLC), Leeds, 13-15 July 2015.*

Cox, G. and McLean, A., 2013. *Speaking code: Coding as Aesthetic and Political Expression.* Cambridge, Massachusetts: MIT Press

Delalande, F., 2001. Il paradigma elettroacustico. *In: Enciclopedia della musica I: il novecento,* Torino: Einaudi.

Dennehy, D., 2008. Interview with James Tenney. *Contemporary Music Review* [online], 27(1): 79-89.

Di Bona, C. and Ockman, S., eds., 1999. *Open Sources: Voices from the Open Source Revolution*. Sebastopol, California: O'Reilly Media.

Drott, E., 2004. Conlon Nancarrow and the Technological Sublime. *American Music* [online], 22(4), 533-563.

Emmerson, S., 2007. *Living Electronic Music*. Burlington, VT: Ashgate Publishing.

Erickson, K., 2018. Performing Algorithms. In: Dean, R. T. and McLean, A., eds. *The Oxford Handbook of Algorithmic Music* [online]. New York: Oxford University Press, 372-378.

Extempore, n.d. *Extempore docs. Philosophy*. Available from: https://extemporelang.github.io/docs/overview/philosophy/ [Accessed 22 August 2020].

Flašar, M., 2016. Listening with the Eyes: Remarks on Live Coding Performance. *MAP - Media / Archive / Performance* [online], 2016 (7).

Flyer, 2019. *Algorave Roma 28 march 2019*. Available from: https://flyer.it/cultural-productions/algorave-roma/ [Accessed 7 August 2020].

FoxDot, n.d. *>>FoxDot. Live coding with Python and SuperCollider*. Available from: https://foxdot.org/ [Accessed 22 August 2020].

Giomi, F., 2017. *Pietro Grossi. L'istante zero - Intervista a Francesco Giomi* [video, online]. Youtube: Tempo Reale. Available from: https://www.youtube.com/watch?v=p952Jkgqwyo [Accessed 3 August 2020]

Goldman, A., 2019. Live coding helps to distinguish between embodied and propositional improvisation. *Journal of New Music Research, 48 (April 2019)*

Grossi, P., n.d. *Home Art.* Available from: https://www.pietrogrossi.org/home-art [Accessed 5 August 2020].

Grossmann, R., 2008. The tip of the iceberg: Laptop music and the information-technological transformation of music. *Organised Sound* [online], 13 (01), 5-11.

Guiot, n.d. *mxmxyz/tidal-guiot.* Available from: https://github.com/mxmxyz/tidal-guiot [Accessed 7 August 2020].

Gunnarson, B., 2012. *Processes and Potentials. Composing through objects, networks and interactions* [online]. Master thesis. Institute of Sonology, The Hague.

Haworth, C., 2018. Technology, Creativity, and the Social in Algorithmic Music. In: Dean, R. T. and McLean, A., eds. *The Oxford Handbook of Algorithmic Music* [online]. New York: Oxford University Press, 615-641.

Herstatt, C. and Ehls, D. eds., 2015. *Open Source Innovation - The Phenomenon, Participant's Behavior, Business Implications.* New York: Routledge.

HFBK-Hamburg, 2004. Changing grammars. Available from: https://swiki.hfbk-hamburg.de/MusicTechnology/609 [Accessed 05 August 2020].

Hiller, L., 1959. Computer Music. *Scientific American* [online], December 1959.

Hiller, L., 1981. Composing with Computers: A Progress Report. *Computer Music Journal* [online], 5(4): 7-21.

Hodnick, M., 2016. *Kindohm / live-2016*. Available from: https://github.com/kindohm/live-2016 [Accessed 7 August 2020].

hundredrabbits, n.d. *hundredrabbits / Orca*. Available from: https://github.com/hundredrabbits/Orca [Accessed 22 August 2020].

Hutton, G., 2016. Programming in Haskell [online]. 2nd edition. Cambridge: Cambridge University Press.

ICLC (International Conference on Live Coding), 2020. *ICLC 2020 - Limerick, Ireland* [online]. Available from: https://iclc.toplap.org/2020/schedule.html [Accessed 2 August 2020].

ixilang, n.d. - - *ixi lang* - - Available from: http://www.ixi-audio.net/ixilang/index.html [Accessed 22 August 2020].

Jack, O., 2019. *Hydra, Live Coding Visuals in the Browser*. Algorithmic Art Assembly. Available from: https://www.youtube.com/watch?v=cw7tPDrFIQg [Accessed 19 August 2020].

Kaer'Uiks, n.d. *Kaer'Uiks*. Available from: http://kaer-uiks.com/ [Accessed 7 August 2020].

Kirkbride, R., 2016. Programming in Time: New Implications for Temporality in Live Coding. In: *Proccedings of the International Conference on Live Coding (ICLC), Hamilton, Canada, 12-15 October 2016*.

Koenig, G. M., 2018. *Process and Form: Selected Writings on Music*. Hofheim am Taunus (Germany): Wolke Verlag.

Kowalkowski, J., 2008. Techniques for Thwarting Hegemony: Anticommunication and Gesture-Inhibiting Musical Material as Compositional Resource in the Music of Herber Brün. *Perspectives of New Music* [online], 46(2): 237-242.

Lawson, S. and Smith, R. R., 2019. What Am I Looking At? An Approach to Describing the Projected Image in Live Coding Performance. In: *Proceedings of the Fourth International Conference on Live Coding, Medialab Prado, Madrid 19-21 January 2019,* 144-153.

Lialina, O. and Espenschied, D. eds., 2009. *Digital Folklore* [online]. Stuttgart: Merz & Solitude.

Ludions, 2007. *Towards a Slow Code Manifesto.* Available from: http://ludions.com/texts/2007a/ [Accessed 14 August 2020].

Luque, S., 2009. The Stochastic Synthesis of Iannis Xenakis. *Leonardo Music Journal* [online], 19(4).

Lurk, n.d. Chat toplap Home. Available from: https://chat.toplap.org/home [Accessed 7 August 2020].

Magnusson, T., 2011. Algorithms as Scores: Coding Live Music. *Leonardo Music Journal* [online], 21 (December 2011): 19-23.

Magnusson, T., 2014. Herding Cats: Observing Live Coding in the Wild. *Computer Music Journal* [online], 38(1): 8-16.

Magnusson, T., 2015. Code scores in live coding practice. In: *TENOR 2015: First International Conference on Technologies for Music Notation and Representation, Paris, 28-30 May, 2015,* 134-139.

Magnusson, T. and McLean, A., 2018. Performing with Patterns of Time. In: Dean, R. T. and McLean, A., eds. *The Oxford Handbook of Algorithmic Music* [online]. New York: Oxford University Press, 280-301.

McLean, A., n.d. *Alex McLean Making Music With Text[ure]* [online]. Available from: www.slab.org [Accessed 4 August 2020].

McLean, A., 2011. *Artist-Programmers and Programming Languages for the Arts* [online]. Thesis (PhD). Department of Computing, Goldsmiths, University of London.

McLean, A., 2019. Pattern, Code and Algorithmic Drumming Circles [online]. In: *Proceedings of the Fourth International Conference on Live Coding, Medialab Prado, Madrid 19-21 January 2019,* 37-43.

McLean, A. and Dean, R.T., 2018a. Algorithmic Trajectories. In: Dean, R. T. and McLean, A., eds. *The Oxford Handbook of Algorithmic Music* [online]. New York: Oxford University Press, 710-722.

McLean, A. and Dean, R.T., 2018b. Musical Algorithms as Tools, Languages, and Partners: A Perspective. *In:* Dean, R. T. and McLean, A., eds. *The Oxford Handbook of Algorithmic Music* [online]. New York: Oxford University Press, 8-22.

McLean, A., Griffiths, D., Collins, N., and Wiggins, G., 2010. Visualisation of Live Code [online]. In: *EVA London 2010 Electronic Visualization and the Arts.*

McLean, A. and Wiggins, G., 2009. Patterns of movement in live languages [online]. In: *Proceedings of the Computers and the History of Art (CHArt) conference 2009, 11 November.*

McLean, A. and Wiggins, G., 2010a. Live Coding Towards Computational Creativity [online]. In: *Proceedings of the 1st International Conference on Computational Creativity 2010,* Lisbon, Portugal, 7-9 January 2010.

McLean, A., and Wiggins, G., 2010b. Tidal - Pattern Language for the Live Coding of Music. In: *Proceedings of the 7th Sound and Music Computing conference 2010, Barcelona.*

McLean, A., and Wiggins, G., 2010c. Bricolage programming in the creative arts [online]. In: *22nd Psychology of Programming Interest Group.*

Mori, G., 2010. *Pietro Grossi e la musica automatica: un percorso di ricerca rivolto verso il futuro* [online]. Thesis (Master). Università degli Studi di Pavia, Facoltà di Musicologia.

Mori, G., 2015a. Analysing Live Coding with Ethnographic Approach: A New Perspective [online]. In: *Proceedings of the International Conference on Live Coding (ICLC), Leeds, 13-15 July 2015,* 117-124.

Mori, G., 2015b. Pietro Grossi's Live Coding. An early case of computer music performance [online]. In: *Proceedings of the International Conference on Live Coding (ICLC), Leeds, 13-15 July 2015,* 125-131.

Mori, G., 2020. *Live Coding? What does it mean? An ethnographical survey on an innovative improvisational approach* [online]. Roma: Aracne Editrice.

musikinformatik, n.d. *musikinformatik / SuperDirt.* Available from: https://github.com/musikinformatik/SuperDirt [Accessed 24 August 2020].

Nesso, n.d. f*racnesco / Tidal-Olbos forked.* Available from: https://github.com/fracnesco/Tidal-Olbos [Accessed 1 September, 2020].

Nierhaus, G., 2009. *Algorithmic Composition. Paradigms of Automated Music Generation.* Wien: SpringerWien.

Nikitina, S., 2012. Hackers as Tricksters of the Digital Age: Creativity in Hacker Culture. *The Journal of Popular Culture,* 45(1), 133-152.

Nilson, C., 2007. Live Coding Practice [online]. In: *Proceedings of New Interfaces for Musical Expression Conference 2007, New York.*

Nilson, C., 2016. *Collected Rewritings: Live Coding Thoughts, 1968-2015* [online]. Burntwood: Verbose.

Ogborn, D., 2014. Live Coding in a Scalable, Participatory Laptop Orchestra. *Computer Music Journal* [online], 38(1), 17-30.

Ogborn, D., 2018. Network Music and the Algorithmic Ensemble. *In:* Dean, R. T. and McLean, A., eds. *The Oxford Handbook of Algorithmic Music* [online]. New York: Oxford University Press, 378-398.

Olbos, 2020 *Olbos / Tidal-Olbos.* Available from: *https://github.com/Olbos/Tidal-Olbos*

Orca, n.d. *hundredrabbits / Orca.* Available from: https://github.com/hundredrabbits/Orca [Accessed 20 August 2020].

Outlines, 2020. *Groove 11.* Available from: *https://outlineslabel.bandcamp.com/album/groove-11* [Accessed 29 July 2020].

Parisi, L., 2013. *Contagious Architecture. Computation, Aesthetics, and Space.* Cambridge, Massachusetts: The MIT Press.

Paz, I., 2015. Live Coding Through Rule-Based Modelling of High-Level Structures: exploring output spaces of algorithmic composition systems. *In:*

Proceedings of the International Conference on Live Coding (ICLC), Leeds, 13-15 July 2015.

Paz, I. and Roig Torrubiano, S., 2019. Tweaking Parameters, Charting Perceptual Spaces. In: *Proceedings of the Fourth International Conference on Live Coding, Medialab Prado, Madrid 19-21 January 2019,* 104-113.

PC Music, 2019. *Folder Dot Zip.* Available from: https://lildatapcmusic.bandcamp.com/ [Accessed 7 August, 2020].

Phase, 2018. *night.* Available from: https://ppphhhaaassseee.com/night/ [Accessed 7 August 2020].

PLOrk, n.d. The Princeton Laptop Orchestra. Available from: https://plork.princeton.edu/ [Accessed 6 August 2020].

Polansky, L., Barnett, A. and Winter, M., 2011. A few more words about James Tenney: dissonant counterpoint and statistical feedback. *Journal of Mathematics and Music: Mathematical and Computational Approaches to Music Theory, Analysis, Composition and Performance* [online], 5(2): 63-82.

Primo, 2019. Due giorni di workshop per scoprire il mondo del live coding, 18 October 2019. Primo comunicazione [online]. Available from: https://primocomunicazione.it/articoli/cultura/due-giorni-di-workshop-scoprire-il-mondo-del-live-coding [Accessed 7 August 2020].

Reed, T. V., 2014. *Digitized Lives: Culture, Power, and Social Change in the Internet Era.* London: Routledge.

Reynolds. S., 2013. *Energy Flash. A Journey through Rave Music and Dance Culture.* New York: Soft Skull Press.

Resident Advisor, 2012. *SuperCollider 2012 Warm Up - Live Algorave* [online]. Available from: https://www.residentadvisor.net/events/345648 [Accessed 26 July 2020].

Riddell, A., 2009. Gesture and Musical Expression Entailment in a Live Coding Context. In: *Proceedings of Australasian Computer Music Association (2009): Improvise, Brisbane, Australia, 2-4 July 2009.*

Roads, C., 2015. *Composing electronic music: a new aesthetic* [online]. New York: Oxford.

Robert, C., 2018. *"Musiques Algorithmiques"* [video, online]. Youtube: Espace Turing. Available from: https://www.youtube.com/watch?v=5ccLWx4NO7U [Accessed 1 August 2020].

Roberts, C. and Kuchera-Morin, J., 2012. Gibber: Live Coding Audio in the Browser [online]. In: *Non-Cochlear Sound: Proceedings of the 38th International Computer Music Confererence, ICMC 2012, Ljubljana, Slovenia, September 9-14, 2012.*

Roberts, C. and Wakefield, G., 2018. Tensions and Techniques in Live Coding Performance. *In:* Dean, R. T. and McLean, A., eds. *The Oxford Handbook of Algorithmic Music* [online]. New York: Oxford University Press, 331-359.

Rohrhuber, J., de Campo, A., 2009. Improvising Formalisation: Conversational Programming and Live Coding. *In:* Assayag. G. and Gerzso, A., eds., *New Computational Paradigms for Computer Music* [online]. Paris: Delatour / IRCAM-Centre Pompidou, 113-124.

Rohrhuber, J., de Campo, A., Wiesers, R., 2005. Algorithms Today. Notes on Language Design for Just in Time Programming [online]. In: *International Computer Music Conference Proceedings 2005, Barcelona, 4-10 September.*

Ruiz-del Olmo, F., Vertedor-Romero, J. A. and Alonso-Calero, J. M., 2019. Creación sonora y nuevas tendencias artísticas en el siglo XXI: Algoritmos, música electrónica y Algorave. *Arte, Individuo y Sociedad* [online], 31(2) 2019, 425-440.

Salazar, S., 2017. Searching for Gesture and Embodiment in Live Coding. In: *Proceedings of the International Conference on Live Coding (ICLC), Morelia, Mexico, 4-8 December 2017.*

Sayer, T., 2015. Cognition and Improvisation: Some Implications for Live Coding. *In: Proceedings of the International Conference on Live Coding (ICLC), Leeds, 13-15 July 2015.*

Solomos, M., ed., 2001. *Présences de Iannis Xenakis.* Paris: Cdmc.

Sorensen, A., 2005. Impromptu: An interactive programming environment for composition and performance [online]. In: *Proceedings of the Australasian Computer Music Conference 2005,* 149-153.

Sorensen, A. and Brown, A. R., 2007. Aa-cell in practice: An approach to musical live coding [online]. In: *Proceedings of the International Computer Music Conference 2007, Copenhagen.*

Sorensen, A. and Gardner, H., 2010. Programming with time: cyber-physical programming with impromptu [online]. In: *Proceedings of ACM OOPLSA, New York,* 822-834.

Sorensen, A., Swift, B. and Riddell, A., 2014. The Many Meanings of Live Coding. *Computer Music Journal* [online], 38(1), 65-76.

Smith, S. and Smith, S., 1979. A Portrait of Herbert Brün. *Perspectives of New Music* [online], 17(2): 56-75.

Spiegel, L., 1981. Manipulations of Musical Patterns. *In: Proceedings of the Symposium on Small Computers and the Arts, Oct. 1981,* 19-22.

Spiegel, L., 2018. Thoughts on Composing with Algorithms. *In:* Dean, R. T. and McLean, A., eds. *The Oxford Handbook of Algorithmic Music* [online]. New York: Oxford University Press, 121-128.

Street, Z., Albornoz, A., Bell, R., John, G., Jack, O., Knotts, S., McLean, A., Smith, N. C., Tadokoro, A., van der Walt, J. S., Rea Velasco, G., 2019. Facilitating technical connections and shared language between visualist and musician livecode performers, and developing practices that enable smoother collaboration. In*: Proceedings of the Fourth International Conference on Live Coding, Medialab Prado, Madrid 19-21 January 2019,* 84-89.

Tenney, J., 1963. Sound Generation by means of a Digital Computer. *Journal of Music Theory* [online], 7(1): 24-70.

Terranova, T., 2014. Algorithms, Capital and the Automation of the Common. *In:* Mackay, R., and Avanessian, A., eds. # *ACCELERATE - The Accelerationist Reader.* Falmouth, UK: Urbanomic.

Thompson, S., 1999. *Haskell: The craft of Functional Programming.* 2nd edition. Boston: Addison-Wesley.

TidalCycles, n.d. a. *Tidalcycles userbase: Related Links* [online]. Available from: *https://tidalcycles.org/index.php/Welcome#Related_links* [Accessed 5 August 2020].

TidalCycles, n.d. b. *Tidal club membership.* Available from: *https://blog.tidalcycles.org/shop/* [Accessed 7 August 2020].

Toplap, 2015. *TOPLAP On Slack.* Available from: *https://toplap.org/toplap-on-slack/* [Accessed 7 August 2020].

Toplap, n.d. *toplap / awesome-livecoding*. Available from: *https://github.com/toplap/awesome-livecoding* [Accessed 22 August 2020].

Toplap Italia, n.d. *toplap italia*. Available from: *https://toplapitalia.gitlab.io/* [Accessed 7 August 2020].

Umanesimo Artificiale, 2020. *Live code your track live*. Available from: *https://umanesimoartificiale.bandcamp.com/album/live-code-your-track-live* [Accessed 1 August 2020].

Vaggione, H., 2001. Some Ontological Remarks about Music Composition Processes. *Computer Music Journal* [online], 25 (1), 54-61.

Valle, A., 2015. *Introduzione a SuperCollider* [online]. Sant'Arcangelo di Romagna, Italy: Apogeo Education.

Wang, G. and Cook, P. R., 2004. On-the-fly programming: using code as an expressive musical instrument [online]. In: *Proceedings of the 2004 conference on New Interfaces for Musical Expression, Singapore,* 138-143.

Wang, G., Cook, P. R., and Salazar, S., 2015. ChucK: A Concurrent, On-the-fly Audio Programming Language. *Computer Music Journal* [online], 39(4), 10-29.

Ward, A., Rohrhuber, J., Olofsson, F., McLean, A., Griffiths, D., Collins, N., and Alexander, A., 2004. Live algorithm programming and a temporary organisation for its promotion. *In:* Goriunova, O. and Shulgin, A., eds. *read_me - Software Arts and Cultures* [online]. Aarhus: Digital Aesthetic Research Centre, 161-174.

Wiggins, G. A. and Forth, J., 2018. Computational Creativity and Live Algorithms. *In:* Dean, R. T. and McLean, A., eds. *The Oxford Handbook of Algorithmic Music* [online]. New York: Oxford University Press, 302-330.

Wilson, S., Cottle, D. and Collins, N., 2011. *The SuperCollider Book*. Cambridge, Massachusetts: The MIT Press.

Wilson, S., Lorway, N., Coull, R., Vasilakos, K. and Moyers, T., 2014. Free as in BEER: Some Explorations into Structured Improvisation Using Networked Live-Coding Systems. *Computer Music Journal* [online], 38(1), 54-64.

Xambó, A., Lerch, A. and Freeman, J., 2018. Music Information Retrieval in Live Coding: A Theoretical Framework [online]. *Computer Music Journal* [online], 42 (4), 9-25.

Xenakis, I., 1992. *Formalized Music. Thought and Mathematics in Composition.* Revised edition. Stuyvesant, NY: Pendragon Press.

yaxu, 2017. *Algorave / guidelines* [online]. Available from: *https://github.com/Algorave/guidelines* [Accessed 25 July 2020].

Zeilinger, M., 2014. Live Coding the Law: Improvisation, Code, and Copyright. *Computer Music Journal* [online], 38(1), 77-89.