Université de Montréal

Le cycle des voix

par
Olivier Bélanger

Faculté de musique

Thèse présentée à la Faculté des études supérieures
en vue de l'obtention du grade de Docteur en Musique (D.Mus.)
option Composition électroacoustique

Novembre 2008

Université de Montréal
Faculté des études supérieures

Cette thèse intitulée:

Le cycle des voix

présentée par:

Olivier Bélanger

a été évaluée par un jury composé des personnes suivantes:

M. Michel Longtin,	président-rapporteur
M. Jean Piché,	directeur de recherche
M^{me} Caroline Traube,	codirectrice de recherche
M. Robert Normandeau,	membre du jury
M. Philippe Depalle,	examinateur externe

Thèse acceptée le: .....

À tous ceux et celles qui croient en la gratuité...

REMERCIEMENTS

Un merci spécial à Jean et Caroline pour la générosité, la patience, la tolérance et la liberté d'esprit...

RÉSUMÉ

Cette thèse se divise en quatre parties. Les trois premiers chapitres concernent les outils, développés au cours des cinq dernières années, qui ont été utilisés pour la composition du *Cycle des voix*. Il s'agit de deux modèles de synthèse en mode source-filtre et d'un environnement de programmation musicale. Le premier chapitre explique la démarche de création d'un modèle de la voix chantée, ainsi que les avancements apportés à un domaine déjà largement développé. Le second chapitre élabore sur les analyses acoustiques effectuées sur les sons provenant d'un didjeridu et sur la construction d'un modèle virtuel. Le troisième chapitre concerne le logiciel de programmation musicale Ounk, alliant la puissance du langage de programmation Python et la qualité du moteur audio Csound. Dans ce chapitre, sont présentés les objectifs qui ont poussé au développement d'un nouvel environnement de programmation musicale, la démarche utilisée ainsi que les principales caractéristiques du logiciel. La quatrième partie élabore sur les intentions musicales, les contraintes imposées ainsi que sur les stratégies adoptées lors de la composition de l'oeuvre musicale le *Cycle des voix*.

Mots-clés : synthèse vocale, didjeridu, composition électroacoustique, programmation musicale, développement logiciel, Csound, Python

ABSTRACT

This thesis is divided in four parts. The first three chapters present the tools, developed over the past five years, that were used for the composition of *the Cycle of Voices*. These tools are two source-filter synthesis models and an audio scripting environment for musical signal processing and composition. The first chapter explains the chosen approach for the development of a singing voice synthesis model and how it contributes to a well developed domain. The second chapter develops on acoustic analysis of sounds produced by the didjeridu and on the development of a virtual model. The third chapter is concerned with Ounk, a software program for music programming, combining the power of the programming language Python and the qualities of the Csound audio engine. In this chapter, the goals that lead to the development of a new musical scripting environment, some design decisions and the main features of the software are explained. The fourth chapter discusses musical intentions, imposed constraints and some strategies developed for the composition of the pieces *the Cycle of Voices*.

Keywords : voice synthesis, didjeridu, electroacoustic composition, musical programming, software development, Csound, Python

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ANNEXES

INTRODUCTION

La synthèse vocale demeure aujourd'hui un des rares types de synthèse qui n'a pas encore trouvé de solution idéale. Une synthèse de la voix chantée de bonne qualité, légère en puissance de calcul, en espace disque et en mémoire vive, capable de synthétiser des phrases complètes n'existe pas encore. La synthèse de la voix chantée développée au cours de ce projet est orientée spécialement pour les compositeurs en ce qu'elle offre une très bonne qualité sonore tout en étant peu coûteuse et extrêmement simple à manipuler. Une avancée au niveau de la production de phonèmes incluant des consonnes a été apportée, spécialement en ce qui concerne les consonnes plosives.

Un modèle de synthèse du didjeridu, tout en étant relativement simple de conception, constitue un instrument de traitement possédant une sonorité exceptionnelle. De par sa forme conique, le didjeridu crée une suite de modes de résonance ayant un rapport inharmoniques entre eux, permettant de générer des filtres très complexes tout en restant simple à manipuler. Bien que la théorie sur l'acoustique du didjeridu soit facilement accessible, aucun modèle n'est actuellement disponible dans les environnements de composition et de synthèse sonore. Le modèle qui a été développé au cours de ce projet offre une très bonne qualité sonore et des paramètres de contrôle simples, reliés aux principes physiques et aux modes de jeu de l'instrument.

Les modèles de synthèse décrits ci-dessus ont d'abord été implémentés dans l'environnement Max/MSP, qui s'est avéré inefficace à plusieurs niveaux. Durant la composition des pièces de ce projet est né le besoin d'un environnement de programmation plus versatile et puissant au niveau de l'algorithmie, de meilleure qualité sonore et permettant un contrôle simplifié des modèles de synthèse. Le logiciel de programmation musicale Ounk a été développé pour répondre à ces besoins. Alliant la puissance du langage de programmation Python à la qualité

du moteur audio Csound, cet environnement permet de construire des algorithmes de contrôle extrêmement puissants, sans avoir le souci des capacités de calcul du processeur puisque la musique peut être rendue en temps différé. Le langage Python permet, en outre, d'attribuer des valeurs par défaut aux paramètres de contrôle des modèles de synthèse, ce qui en simplifie grandement la manipulation.

Le cycle des voix est une suite de pièces entièrement composées à l'aide des outils mentionnés ci-dessus. Le dernier chapitre élaborera sur les intentions musicales qui ont motivé la composition de ces pièces. Il sera aussi question des contraintes imposées lors de ce projet ainsi que des stratégies adoptées lors de la conception de l'oeuvre musicale.

CHAPITRE 1

MODÈLE DE VOIX CHANTÉE EN MODE SOURCE-FILTRE

Dans ce chapitre seront exposées différentes méthodes de synthèse de la voix chantée élaborées depuis l'émergence de l'informatique jusqu'à aujourd'hui. Ces modèles ont été choisis, parmi bien d'autres, pour souligner l'importance de leurs développements sur la recherche en synthèse vocale et aussi en fonction des similitudes qui existent avec le modèle présenté dans le cadre de cette recherche. Seront aussi expliqués les objectifs et les besoins qui ont conduit au développement d'un nouveau modèle de synthèse en mode source-filtre. La dernière section élaborera sur les stratégies adoptées afin de mettre en place un système de contrôle des trajectoires de formants permettant la synthèse de phonèmes de type consonne (Bélanger et al., 2007).

1.1 Acoustique de la voix

La voix est produite par l'interaction de deux systèmes indépendants : une source d'excitation et un résonateur complexe. L'excitation est générée par trois types de sources :

- Une source **voisée**, émise par la vibration des cordes vocales au passage de l'air en provenance des poumons. Cette source génère un signal quasi-périodique avec une fréquence fondamentale identifiable, présente notamment lors de la production des voyelles.
- Une source **fricative**, qui correspond aux turbulences produites par le passage de l'air en certains points de resserrement du conduit vocal, créés par le déplacement de la langue, des lèvres ou des dents. Cette source est présente lors de la production des consonnes fricatives telles que [f], [s] et [ch].
- Une source **plosive**, causée par la fermeture complète du passage de l'air,

créant une montée de pression, et le relâchement soudain de l'air. Cette source est utilisée pour produire les consonnes plosives telles que [b], [d], [g] et [p], [t], [k].

Le résonateur correspond aux transformations continues de la forme du conduit vocal. Il peut être modélisé par un banc de filtres passe-bande en parallèle, qui constitue l'enveloppe spectrale modifiant la source d'excitation. L'enveloppe spectrale d'un signal vocal, particulièrement lors de la production des voyelles, est caractérisée par la présence de formants, c'est-à-dire des pics de résonance dans le spectre.

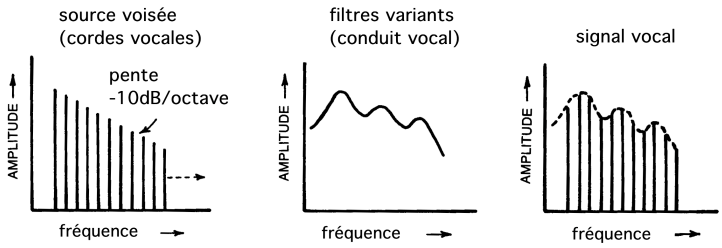


FIG. 1.1 – Schéma de production d'un signal vocal voisée. (Everest, 1989)

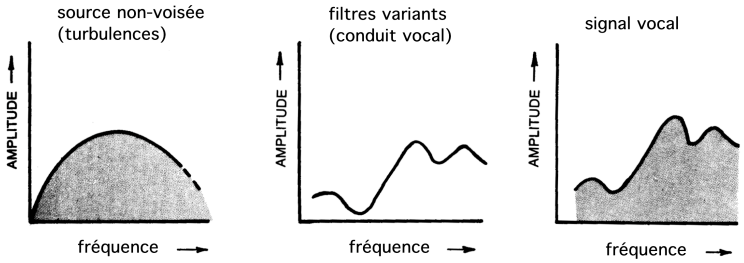


FIG. 1.2 – Schéma de production d'un signal vocal non-voisé. (Everest, 1989)

1.2 Synthèse de la voix chantée : Quelques modèles importants

1.2.1 Le Voder

En 1939, dans les laboratoires de Bell, Homer Dudley met au point le *Voder*, une machine servant à modéliser le signal de la voix. Comme pour beaucoup des recherches dans le domaine de la synthèse vocale, le *Voder* était conçu pour réduire la quantité de signal transmis sur les lignes de téléphone.

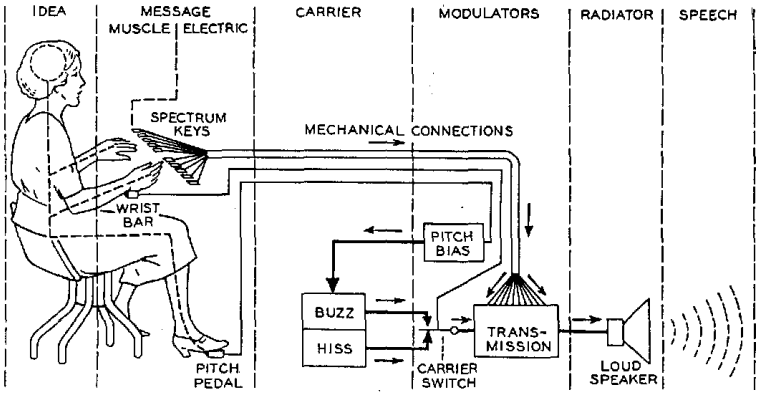


FIG. 1.3 – Circuit schématique du Voder. (Lee, nil)

Le *Voder* est un instrument qui doit être manipulé par un technicien hautement entraîné. Dix touches de clavier servent à contrôler l'amplitude de dix filtres passe-bande, responsables du contenu spectral du signal, c'est-à-dire des formants de la voix. Un levier situé à la hauteur du poignet permet d'alterner entre un signal d'excitation périodique (train d'impulsions) et un signal d'excitation aléatoire (bruit blanc). Une excitation périodique permet de recréer les voyelles et certaines consonnes tandis qu'une excitation aléatoire permet de reconstituer les fricatives, telles que le [f], le [s] et le [ch]. La fréquence du signal périodique est contrôlée par une pédale, permettant de créer des inflexions de voix réalistes (Bilmes, 2003). Les expérimentations de Homer Dudley sur les principes du *vocoder*[1] ont grandement

[1]Extension du Voder basée sur l'analyse du son vocal.

inspiré le milieu de la musique électronique.

1.2.2 Synthèse par Forme d'Onde Formantique

La synthèse par forme d'onde formantique a été développée au milieu des années 70 par Xavier Rodet, à l'IRCAM (Rodet, 1984). Le principe consiste à générer des flux de grains dont le contenu spectral correspond aux formants de la voix.

Les grains sont générés en appliquant une enveloppe d'amplitude sur un signal de type sinusoïdal, enregistré dans une table, produisant ainsi une sinusoïde amortie. Le signal obtenu est un spectre harmonique dont la fréquence fondamentale correspond à la fréquence de génération des grains et la largeur de bande à l'action combinée de la fréquence centrale et de la forme de l'enveloppe d'amplitude. Entre trois et cinq flux de grains FOF peuvent être superposés afin de reproduire les formants d'un son de voyelle. Cette technique permet d'obtenir une qualité sonore remarquable et offre l'avantage de contrôler la synthèse avec des paramètres très évocateurs de l'instrument réel, tels que la fréquence fondamentale et l'emplacement des formants dans le spectre (Rodet et al., 1984).

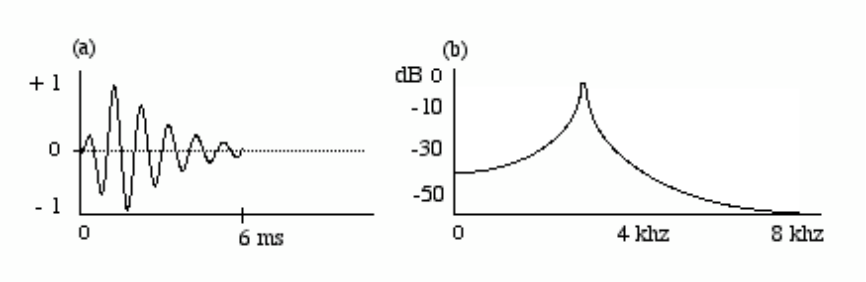


FIG. 1.4 – (a) Signal temporel d'une FOF. (b) Le spectre d'amplitude obtenu. (Iturbide, nil)

1.2.3 Modélisation physique du conduit vocal

Le modèle de conduit vocal mis au point par John Kelly et Carol Lochbaum est constitué d'une série de tuyaux recréant la forme du conduit vocal. Leurs travaux furent publiés sous forme d'article paru en 1962.

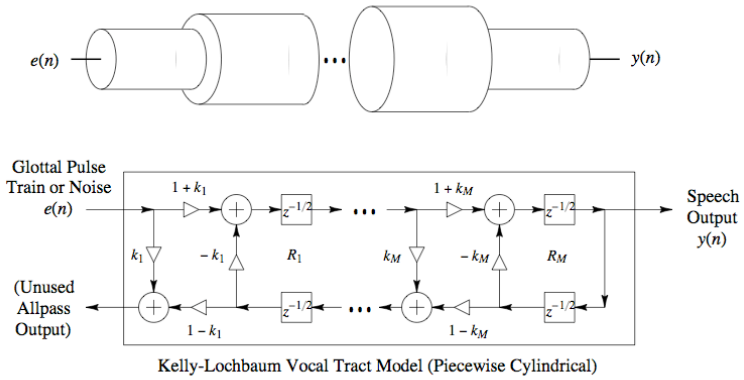


FIG. 1.5 – Modèle acoustique d'un conduit vocal, constitué d'une séquence de sections de cylindre et modèle du filtre numérique correspondant, d'après Kelly et Lochbaum, 1962. (Smith, 2005).

Ce modèle est en quelque sorte l'ancêtre des modèles physiques par guides d'ondes, et a été utilisé pour synthétiser la célèbre reprise synthétisée de la chanson «Bicycle built for two», mise en musique par Max Mathews. Cette pièce est probablement le premier exemple sonore, toutes méthodes confondues, d'un modèle physique de synthèse de la voix chantée (Smith, 2005).

1.2.4 SPASM

SPASM est un modèle physique de la voix chantée développé au début des années 90 par Perry Cook. Ce modèle, basé sur les travaux de Kelly et Lochbaum, utilise un réseau de guides d'ondes pour synthétiser les différentes parties du conduit vocal. La différence principale entre les deux modèles se situe au ni-

veau des contrôles. L'excitation est une combinaison d'un signal harmonique (train d'impulsions) et d'un signal aléatoire (bruit blanc). Le contrôle des transformations de chaque section du conduit vocal permet de synthétiser un grand nombre de phonèmes, notamment des voyelles et des consonnes nasales, grâce à la jonction entre la cavité buccale et le conduit nasal.

La bifurcation du conduit vocal vers le conduit nasal située au voile du palais peut être modélisée avec une jonction à trois passages. Une partie de l'énergie en provenance de la glotte sera dérivée vers la cavité nasale, une autre partie continuera vers les lèvres et le reste sera réfléchi et retournera vers la glotte (Cook, 1991).

1.2.5 Onde glottique dérivée

Au début des années 2000, Hui-Ling Lu, du CCRMA à Stanford, propose un modèle d'analyse-resynthèse de la voix de haute qualité. Ce modèle est construit en mode source-filtre, et met l'accent sur la génération de l'onde glottique.

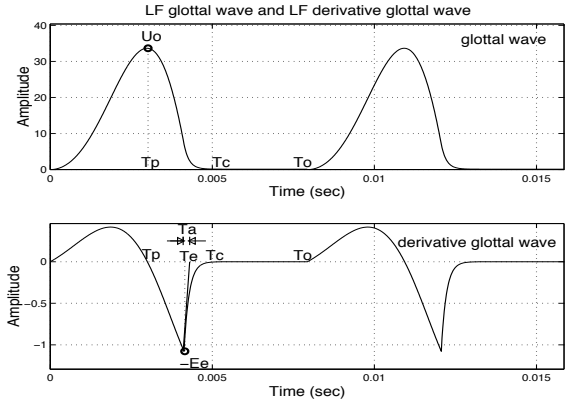


FIG. 1.6 – Illustration de l'onde glottique et de l'onde glottique dérivée. (Smith, 2005)

Un des apports importants de ce projet est le développement d'une procédure d'analyse estimant les paramètres nécessaires à la reproduction du timbre de voix désiré. Les paramètres permettant le contrôle de l'onde glottique ainsi que de la

composante bruitée de l'excitation sont obtenus par la transformée en ondelette d'enregistrements de voix. Ce système permet une synthèse plus fidèle de la source d'excitation et, par conséquent, une reproduction de voyelle très réaliste (Lu, 2002).

1.2.6 CALM Synthesizer

En 2006, Nicolas D'Alessandro, de la Faculté Polytechnique de Mons, présentait un nouveau modèle de synthèse de la voix, basé sur le modèle CALM[2] (Henrich et al., 2003), destiné à une utilisation en temps réel via différentes interfaces de contrôle, telles qu'une tablette graphique.

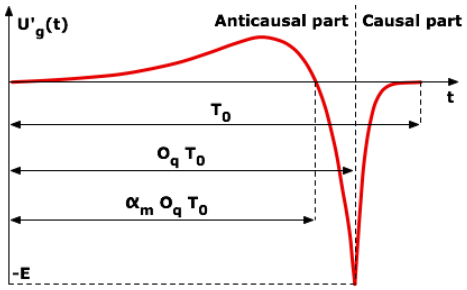


FIG. 1.7 – Représentation dans le domaine temporel de la partie anticausale et de la partie causale de la dérivée d'une impulsion glottique. (D'Alessandro et al., 2006)

Ce modèle est aussi de type source-filtre et utilise une toute nouvelle approche pour décrire l'onde glottique. Le modèle CALM utilise des équations permettant de relier les paramètres importants de l'impulsion glottique, en fonction des attributs spectraux de l'onde, aux paramètres des équations générant les filtres résonants anticausal et causal. Cette cascade de filtres permet de modéliser fidèlement le *formant glottique* ainsi que la pente spectrale, due entre autres, au rayonnement du signal au niveau des lèvres. Dans ce modèle, des paramètres tels que la tension,

[2]causal-anticausal linear model

le souffle, l'effort et l'articulation peuvent être spécifiés via une interface de contrôle et facilement reliés aux paramètres des filtres générant la source d'excitation.

1.3 Objectifs

Dans le cadre de ce projet de recherche doctorale, nous avions pour objectif de créer un modèle de synthèse de la voix chantée de bonne qualité, expressive et simple à manipuler en situation de composition. La synthèse en mode source-filtre s'est avérée répondre à tous ces critères. Extrêmement simple à contrôler et très stable, elle permet d'organiser les paramètres de manipulation de façon à ce qu'ils soient compréhensibles pour tous. Ce que l'utilisateur veut avoir à spécifier, c'est le phonème, la brillance et la raucité de la voix, c'est-à-dire des termes descripteurs qui sont utilisés naturellement pour le chant. Les contrôles du modèle élaboré au cours de ce projet offrent ce type de notation des paramètres, permettant une utilisation simple et expressive de la synthèse vocale.

Le modèle devait aussi permettre l'articulation de phonèmes contenant des consonnes, et non seulement la synthèse des voyelles, comme c'est le cas notamment de la synthèse par forme d'onde formantique.

Afin de pouvoir utiliser cette synthèse dans le cadre de compositions algorithmiques à plusieurs voix générées en temps réel, il était important que le modèle ne soit pas trop gourmand en puissance de processeur. La demande en quantité de calcul devait aussi rester stable en toute situation. Plusieurs types de synthèse, tels que la fof et la synthèse par guide d'onde, sont plus exigeants lorsqu'il s'agit de générer des fréquences aiguës. Le modèle source-filtre est léger et demande le même temps de calcul peu importe le registre, ce qui permet de générer plusieurs voix de polyphonie sur un seul processeur.

Une attention particulière a été portée sur le contrôle fin de la fréquence fondamentale, des modulations et des trajectoires de formants, afin de conférer un

caractère le plus naturel possible aux voix de synthèse.

L'implémentation de ce modèle a été réalisée avec l'environnement Csound, les contrôles et les algorithmes sont programmés dans le logiciel Ounk.

1.4 Le modèle source-filtre

Le modèle de synthèse source-filtre n'est pas nouveau en soi, mais il permet d'obtenir un son de voix naturel et facile à contrôler. Dans le cas d'une synthèse de la voix, la source est constituée d'une combinaison de signaux périodiques et aléatoires, simulant l'excitation de l'onde glottique. Le filtre est construit avec un banc de filtres passe-bande modélisant le conduit vocal.

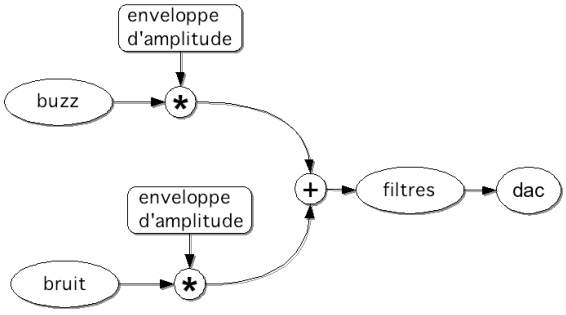


FIG. 1.8 – Schéma de base d'un modèle de synthèse de la voix en mode source-filtre.

1.5 Modélisation de l'excitation glottique

Le signal d'excitation, qui correspond au passage de l'air, en provenance des poumons, au travers des cordes vocales, est produit à partir de deux générateurs : un train d'impulsions et un générateur de bruit. Le train d'impulsions est utilisé pour recréer les vibrations des cordes vocales, contenant une fréquence fondamentale identifiable, comme lors de la production des voyelles. Le générateur de bruit est utilisé pour la production des sons non voisés, lors de l'articulation de certaines consonnes ([t], [k]) et des fricatives ([f], [s]). Afin de produire des sons de voix

réalistes, deux enveloppes d'amplitude effectuent le mixage nécessaire entre les signaux périodiques et aléatoires en fonction du phonème demandé. Le caractère naturel de la voix est rehaussé par l'addition d'un vibrato, entièrement contrôlable par l'utilisateur, ainsi que par l'ajout d'une modulation reproduisant la raucité de la voix.

1.5.1 Génération de la source harmonique

Afin de donner un caractère naturel aux sons de voix synthétisés, de légères variations ont été ajoutées à la source harmonique, brisant ainsi la régularité trop parfaite du train d'impulsions. Un son de synthèse pur est rapidement perçu comme étant une voix synthétique, car il manque les micro-modulations que l'on retrouve dans un signal produit par des cordes vocales réelles. Ce sont ces micro-modulations qui confèrent le caractère naturel à la voix humaine. Dans ce modèle, comme on peut le voir sur la figure 1.9, de légères variations aléatoires ont été appliquées sur la fréquence fondamentale, sur l'amplitude ainsi que sur la pente du filtre du train d'impulsions. Chaque variation possède sa vitesse et son amplitude propres. Un filtre passe-bas est appliqué directement sur la source afin de modéliser le filtrage dû au rayonnement de l'onde à la sortie de la bouche.

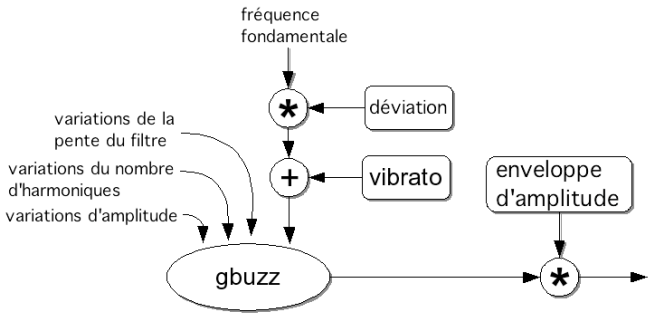


FIG. 1.9 – Implémentation de la source harmonique de l'excitation.

1.5.2 Contrôle du vibrato

Un vibrato contrôlé avec soin contribue grandement à la reconnaissance de la voix chantée. Une simulation vocale sans vibrato, ou sans micromodulation, est souvent associée à un son de synthèse sans aucune référence à la voix. Une modulation d'environ 1% de la fréquence fondamentale, avec une onde sinusoïdale ou triangulaire, donne un résultat très réaliste, pour tous les registres de la voix. Tout comme pour la génération du train d'impulsions, le vibrato ne doit pas être parfaitement périodique car il sera instantanément identifié comme provenant d'une machine. Pour régler ce problème, la fréquence et l'amplitude sont soumises à de légères variations aléatoires, afin de créer un vibrato qui ne soit ni trop mécanique ou trop régulier. En accord avec une étude sur la perception du vibrato (Verfaille et al., 2005), le vibrato module également la position des formants ainsi que l'amplitude générale du signal, ce qui rehausse la perception du caractère naturel de la voix. Le modèle prend également en compte le fait qu'un chanteur produit rarement le vibrato dès le début de la note. Tel qu'illustré sur la figure 1.10, une enveloppe d'amplitude permet d'introduire graduellement le vibrato après le départ de la note.

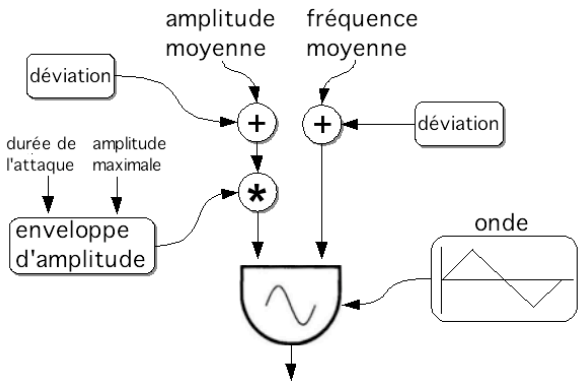


FIG. 1.10 – Structure du vibrato.

1.5.3 Raucité de la voix

Un autre aspect traduisant les composantes bruitées de la voix chantée est la présence d'une petite bande de bruit, à faible amplitude, provenant des turbulences de l'air au passage des cordes vocales. On nomme généralement ce phénomène la raucité de la voix. Pour la simuler, le train d'impulsions est modulé par un bruit rose de très faible amplitude, filtré par un filtre passe-bande entre 1 kHz et 4 kHz (D'Alessandro et al., 2006).

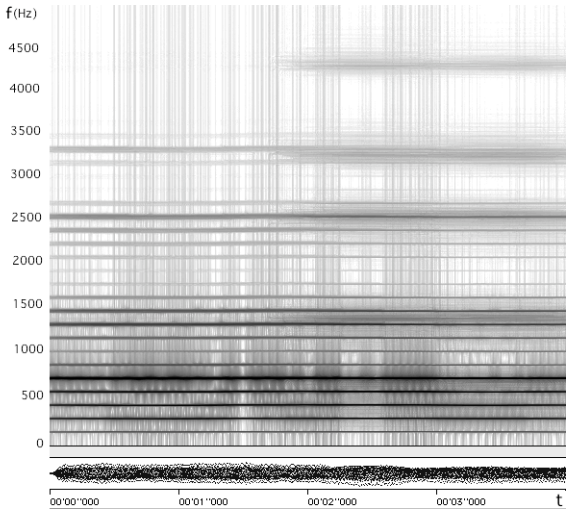


FIG. 1.11 – Sonagramme d'une voyelle synthétisée où la raucité a été introduite à la mi-parcours.

Le modèle offre la possibilité de modifier l'amplitude du bruit afin de créer différentes qualités de voix, de rauque à flûtée, ou de créer un effet de chuchotement lorsque l'amplitude du bruit est au maximum, masquant ainsi la composante harmonique du signal d'excitation.

1.5.4 Générateur de la source bruitée

La source bruitée, généralement présente lors de la production des consonnes plosives ou des fricatives, est générée par un bruit rose modulé par une enveloppe

d'amplitude similaire à l'enveloppe modulant la source harmonique. Ainsi, les deux sources peuvent être mixées avant d'être injectées dans la banque de filtres simulant le conduit vocal. Lors de la production de consonnes plosives, l'enveloppe d'amplitude de la source bruitée doit être finement ajustée puisqu'une impulsion bruitée apparaît très brièvement, souvent précédée et suivie d'un court silence de la source harmonique. La durée du silence entre l'impulsion et l'apparition de la composante harmonique, s'étendant de 5 à 30 ms, est appelé le *Voice Onset Time* (VOT). Ce paramètre est crucial dans la perception des différentes consonnes plosives. Par exemple, la durée du silence sera plus importante pour la production de la consonne [t] (environ 40 ms) que pour la production de la consonne [d] (environ 10 ms) (Lisker, 1975, Niyorgi and Ramesh, 1998). Le système auditif est extrêmement sensible à ces petites différences. Les enveloppes d'amplitude ont d'abord été déduites suite à des analyses effectuées sur des enregistrements de phonèmes chantés. Ensuite, à l'aide de tests d'écoute et de manipulation des paramètres, des ajustements fins ont été apportés aux trajectoires. Les enveloppes d'amplitude sont mémorisées dans des tables de données à l'intérieur du modèle. L'utilisateur n'a qu'à spécifier la consonne et la voyelle désirées, et les trajectoires d'amplitude seront automatiquement générées.

1.6 Modélisation du conduit vocal

Le filtrage produit par le conduit vocal est simulé par un banc de filtres passe-bande en parallèle, reproduisant des pics d'amplitude dans le spectre, que l'on nomme les formants, caractéristiques du signal vocal. Afin d'améliorer la précision des filtres et la qualité sonore du signal généré, chacun des formants est modélisé par une cascade de trois filtres résonants du second ordre. Cette chaîne de filtres passe-bande donne une pente de coupure plus nette, éliminant ainsi les composantes spectrales indésirables. Csound permet d'obtenir cette cascade de filtres résonants

à l'intérieur d'un seul opcode[3], pour lequel on spécifie le nombre de filtres désirés dans la série. En plus d'être extrêmement stable, cet opcode est plus léger en temps de calcul qu'une série de filtres indépendants, puisque le calcul des vecteurs d'échantillons audio se fait à l'intérieur de la mémoire cache du processeur. Trois paramètres sont nécessaires à la définition d'un formant : la fréquence centrale, l'amplitude et la largeur de bande. Ces paramètres ont été obtenus par l'analyse d'échantillons de voyelles prononcées par des chanteurs et chanteuses de tessitures différentes, ainsi que par des ajustements à l'écoute, et sont mémorisés dans des tables à l'intérieur du modèle. Selon le registre et la voyelle demandée, le modèle ira récupérer les valeurs des formants appropriées. Pour obtenir des voyelles aux timbres légèrement différents chaque fois qu'une note est chantée, une variation d'environ 2% est appliquée à la fréquence centrale des formants, éliminant ainsi l'effet robotique d'une voyelle qui serait répétée avec exactement la même forme de conduit vocal.

1.6.1 Rôle des formants

Cinq formants sont définis pour les quinze principales voyelles françaises de l'alphabet phonétique international (voir figure 1.12). Les deux premiers formants jouent un rôle important dans la perception de la voyelle produite. Le troisième et le quatrième formants, en combinaison avec les deux premiers, aident à la perception et à la catégorisation de certaines consonnes (Delattre et al., 1955, 1958). Le rôle du cinquième formant consiste à ajuster la qualité du timbre de la voix. Ce dernier pourrait être éliminé sans perdre l'intelligibilité du phonème prononcé.

Un deuxième groupe de filtres peut être utilisé afin de simuler la cavité nasale. Lorsqu'une voyelle ou une consonne nasale est produite, des annulations de fréquence sont modélisées par l'ajout de filtres à réponse impulsionnelle finie créant

[3]resonx. Opcode est le nom donné aux opérateurs dans le langage Csound.

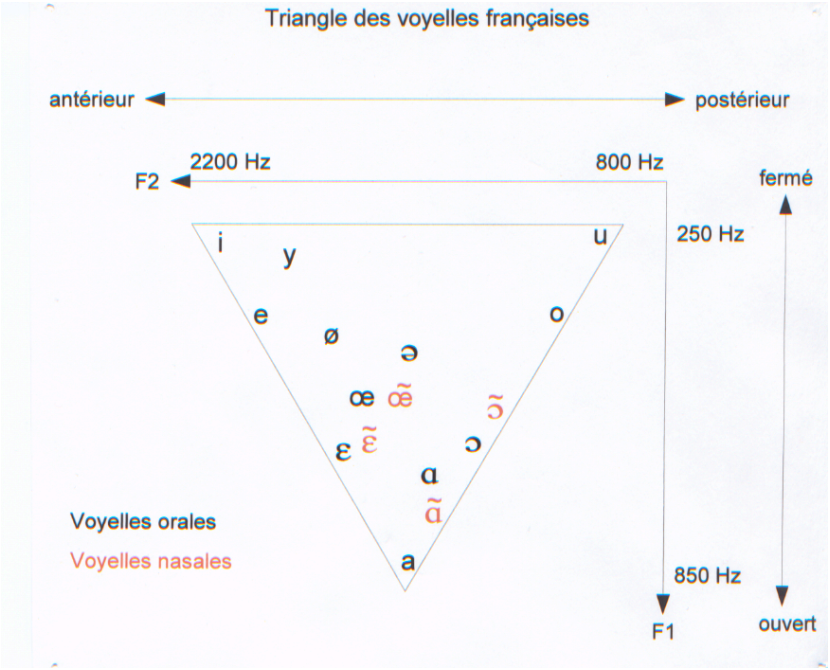


FIG. 1.12 – Les voyelles telles que définies dans le modèle représentées sur le triangle en fonction du point d'articulation et de l'ouverture de la bouche.

des anti-résonances dans le spectre d'amplitude (Delvaux et al., 2004). Au moment de la rédaction de cette thèse, cet aspect du modèle est encore à l'état d'ébauche.

Les analyses effectuées sur des échantillons sonores ont démontré que le spectre d'amplitude présente toujours une forte intensité autour de la fréquence fondamentale. Nous avons observé ce résultat même dans le cas où le premier formant se trouve à une fréquence plus élevée que la fréquence fondamentale. Dans un modèle source-filtre utilisant cinq formants en parallèle, comme c'est le cas du modèle présenté dans le cadre de cette recherche, le résultat sonore est obtenu par la sommation de la sortie de chacun des filtres. Si le premier formant se situe à une fréquence plus élevée que la fréquence fondamentale, celle-ci sera grandement atténuée, ce qui aura pour effet de générer un signal où manque la composante de base de la note. La solution a été de rajouter un autre filtre au modèle, en parallèle aux formants, afin de renforcer le registre autour de la fréquence fondamentale.

L'amplitude de ce filtre, appelé *résonance glottique* (D'Alessandro et al., 2006), peut être ajusté pour contrôler la profondeur du son de voix produit.

1.6.2 Registres

Selon la tessiture de la voix qui produit une sonorité de voyelle, les fréquences centrales des formants ne seront pas situées aux mêmes fréquences. Les rapports de fréquences centrales de formants pour une voyelle donnée seront aussi différents selon les registres. On ne peut donc pas tout simplement étirer ou contracter une configuration de base des fréquences centrales en fonction de la tessiture (Martin, 2000). Des recherches ont aussi démontré qu'il est possible de catégoriser les registres en fonction de l'emplacement des formants pour une fréquence fondamentale fixe (Coleman, 1971). Quatre registres sont définis dans le modèle : basse, ténor, alto et soprano. Pour chaque registre, des tables de paramètres sont mémorisées pour les valeurs de fréquences centrales, d'amplitudes et de largeurs de bande. Le modèle utilisera les tables correspondant à la tessiture de voix choisie par l'utilisateur.

Registre	F1	F2	F3	F4
basse	400	750	2400	2600
ténor	360	770	2530	3200
alto	420	850	3040	4160
soprano	500	950	3240	4160

TAB. 1.1 – Table des fréquences des formants pour la voyelle [o] en fonction du registre.

1.6.3 Correction de formant pour les voix de soprano

Un problème particulier se produit lorsqu'une note très aiguë est produite dans le registre de soprano. Si la fréquence fondamentale est plus élevée que la fréquence centrale du premier formant, il y aura moins d'énergie dans ce formant, créant

un son de synthèse de faible amplitude, au timbre mince, très pauvre en basse fréquence. Lors de la production d'une note aiguë, les chanteuses sopranos modifient la forme de leur conduit vocal afin d'ajuster le premier formant sur la fréquence fondamentale de la note chantée, lorsque cette dernière est la plus élevée (Sundberg, 1977, Joliveau et al., 2004). Cet ajustement donne du corps et de la chaleur au son, comme l'implémentation de cet ajustement dans le modèle le confirme. Il est à noter que cette transformation du conduit vocal altère l'intelligibilité des voyelles prononcées. De plus, à des fréquences fondamentales entre 700 Hz et 1000 Hz, les harmoniques sont si éloignées les unes des autres que l'enveloppe spectrale est de toute façon «sous-échantillonnée». Il est donc très difficile de percevoir clairement les voyelles prononcées. Nous avons donc jugé préférable d'accorder la priorité à la qualité du timbre sur l'intelligibilité de la voyelle.

1.7 Trajectoires des formants pour la synthèse des consonnes plosives

Un des aspects originaux de ce projet consiste en l'implémentation de trajectoires de formants pour la production des consonnes. Ce type d'implémentation fait souvent défaut dans les modèles de synthèse de la voix chantée en mode source-filtre. Ces trajectoires, lors de la production de consonnes, découlent d'une série de transformations de la forme du conduit vocal, se terminant sur les valeurs de formants de la voyelle cible[4] (Chafcouloff, 2004). Dans ce modèle, ces transformations sont simulées par des changements continus appliqués aux fréquences centrales des formants. Afin de reproduire correctement ce phénomène, des trajectoires particulières, en fonction des consonnes projetées, sont aussi appliquées à l'amplitude et à la largeur de bande des formants. Les paramètres définissant les contours de trajectoires sont mémorisés dans des tables, à l'intérieur du modèle, et sont automatiquement appelés selon la consonne et la voyelle coarticulées.

[4]Voir tableau des trajectoires en annexe I.

L'équation 1.1 présente la formule utilisée pour définir une trajectoire d'un point de départ f_1, correspondant à la fréquence centrale initiale du formant de la consonne, à un point d'arrivé f_2, correspondant à la fréquence du formant de la voyelle. α représente le coefficient de courbure et N le nombre total d'échantillons dans la transition.

$$F(n) = f_1 + (f_2 - f_1)\frac{1 - e^{-\alpha n/N-1}}{1 - e^{-\alpha}} \tag{1.1}$$

Les transitions de formant, lors de la production d'une consonne plosive, sont très rapides, de 30 à 80 ms de durée. Le système auditif est extrêmement sensible à ce type de transitions, qui constituent un des principaux indices de perception des consonnes plosives (Blumstein and Stevens, 1979, Jackson, 2001). Deux autres indices participent à la reconnaissance des consonnes : la présence de bruit et le VOT[5]. Une brève impulsion de bruit est toujours présente lors de la production des consonnes plosives non-voisées telles que [p], [t] ou [k]. Le VOT, c'est-à-dire un court instant de silence entre le bruit et le départ de la note, devra être ajusté de façon très précise en fonction de la consonne prononcée. Ces paramètres sont aussi mémorisés dans des tables et appelés au besoin.

La figure 1.13 illustre un exemple de trajectoires de fréquences centrales des trois premiers formants pour les syllabes [da] (en bleu) et [ba] (en rouge). Les trajectoires démarrent sur les valeurs définies pour la consonne et se déplacent rapidement vers les valeurs de formants de la voyelle [a], puis gardent ces valeurs jusqu'à la fin de la note. La forme du conduit vocal diffère selon que l'on produit un [b] ou un [d], ce qui a pour effet de modifier le point de départ des transitions de formant. On constate que la différence entre les deux consonnes réside dans le point de départ des deuxième et troisième formants. Alors que dans les deux cas, la fréquence centrale du premier formant part d'une valeur plus basse que la

[5] *Voice Onset Time*, voir page 15 pour la description.

fréquence du formant de la voyelle, dans le cas du [d], le point de départ des deux autres formants se situe au-dessus de la fréquence du formant de la voyelle.

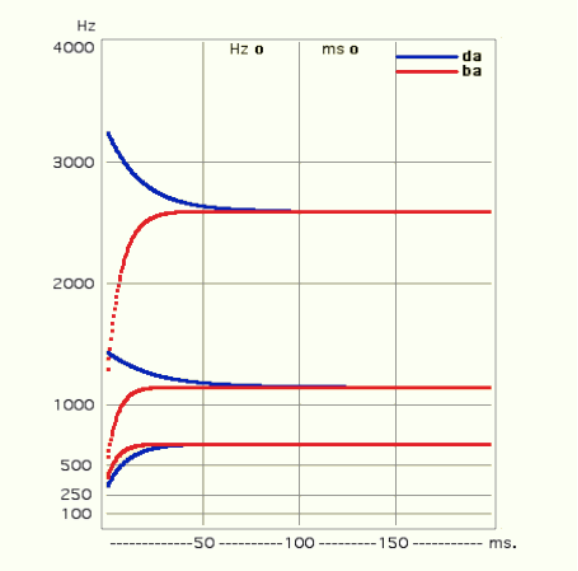


FIG. 1.13 – Trajectoires des trois premiers formants pour les syllabes [da] et [ba].

1.7.1 Locus acoustique

Selon une étude sur les indices de perception des consonnes plosives (Delattre et al., 1955), il est possible de réduire la base de données nécessaire à la définition des trajectoires de formant en assignant un point de départ commun pour toutes les consonnes ayant le même point d'articulation, indépendamment de la voyelle qui suit. Par exemple, les consonnes labiales [b], [p] et [m] sont toutes articulées au niveau des lèvres et peuvent donc toutes être synthétisées en démarrant la transition au même locus acoustique (même point de départ). Les consonnes alvéolaires [d], [t] et [n], qui sont articulées avec la langue contre le palais, auront elles aussi le même locus. La théorie du locus acoustique fonctionne particulièrement bien pour situer le point de départ du deuxième formant des consonnes car c'est celui qui est le plus influencé par la forme de la bouche. Une deuxième étude (Delattre et al., 1958) a

démontré que cette théorie est plus difficilement applicable au troisième formant. Des expérimentations avec le modèle tendent à corroborer cette étude. Afin de respecter la théorie du locus acoustique, les trajectoires d'amplitude doivent être précisément ajustées en introduisant une période de silence au début de la transition comme illustré sur la figure 1.14.

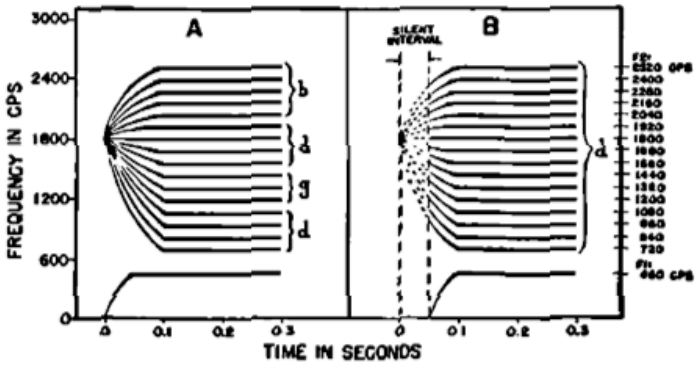


FIG. 1.14 – Position de départ du deuxième formant selon la théorie du locus acoustique, avec (B) et sans (A) un temps de silence dans la transition (Delattre et al., 1955).

1.7.2 Contrôle des paramètres pour une génération musicale expressive

Un modèle de synthèse de la voix, pour être efficace, doit permettre de créer une grande variété de timbres et d'articulations, en spécifiant seulement quelques paramètres de contrôle. Dans ce modèle, tous les paramètres nécessaires à la production d'une syllabe sont mémorisés dans des tables et sont appelés par groupe afin de générer la consonne, la voyelle qui suit ainsi que le comportement articulatoire. Seulement cinq paramètres principaux doivent être précisés, c'est-à-dire la durée de la note, la fréquence fondamentale, la consonne, la voyelle et le registre désiré. Une douzaine de paramètres supplémentaires sont optionnels et permettent de régler finement le timbre et le comportement du modèle.

La coarticulation est l'un des aspects les plus importants de la production vo-

cale. Une syllabe ne prend son sens qu'en fonction de la syllabe qui l'a précédée et de celle qui la suit. Généralement, il y a une continuité sonore entre deux sons articulés. Reproduire ce phénomène est essentiel pour la génération de phrasés réalistes. Il est possible, dans ce modèle, de spécifier qu'un événement doit être lié à l'événement suivant, en lui donnant une durée négative (normalement exprimée en seconde). Lorsque la prochaine note est appelée, le modèle n'exécute pas la phase d'initialisation et débute les trajectoires à partir de l'endroit où l'événement précédent était rendu (les valeurs de formant correspondant à une voyelle par exemple).

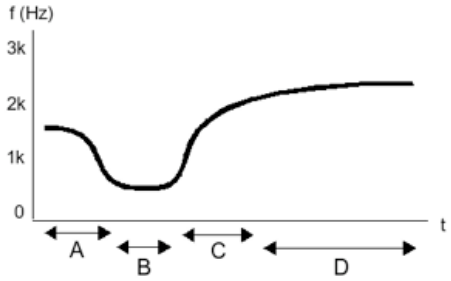


FIG. 1.15 – Une trajectoire typique de liaison entre deux événements pour un seul formant.

La figure 1.15 illustre une trajectoire typique pour la fréquence centrale d'un seul formant, débutant sur l'état stable d'une voyelle, suivi d'une syllabe comportant une consonne plosive. La partie A correspond à la chute de la voyelle précédente, d'une durée d'environ 30 ms. La partie B représente un temps de silence ou une très faible résonance durant la production de la consonne (très important pour la perception des consonnes plosives), juste avant l'attaque de la nouvelle note. Cette durée peut varier de 20 à 100 ms selon la consonne désirée. La partie C correspond à la trajectoire de formant de la consonne choisie et la partie D à la tenue stable définie par les paramètres de la voyelle ciblée. Tous les paramètres de formants (fréquence centrale, amplitude et largeur de bande) suivent des trajectoires similaires, générées en fonction de la consonne et de la voyelle à synthétiser.

1.8 État des travaux

1.8.1 Ce qui a été développé

Le modèle de synthèse de la voix en mode source-filtre développé au cours de ce projet a été conceptualisé pour générer une synthèse vocale au caractère naturel, par l'ajout de micro-variations modulant la source d'excitation, ainsi que par un contrôle précis des transitions de consonne à voyelle.

Un des avantages de ce modèle est que toute l'information nécessaire à la production de sons de voix réside dans un simple fichier texte. Ce modèle est donc très léger et très facile à partager.

Ce modèle de synthèse vocale a d'abord été programmé dans l'environnement Max/MSP avec l'objet csound~. Il a ensuite été converti en une fonction du logiciel Ounk. Nous verrons dans le chapitre 4 comment il peut bénéficier de la puissance du langage de programmation Python et de l'environnement de composition Ounk afin de construire des structures musicales riches et variées.

1.8.2 Ce qui reste à faire

Afin de pouvoir synthétiser des phrases complètes, il reste à compléter la base de données pour les paramètres de certaines consonnes qui ne sont pas encore implémentées.

Un autre aspect qui serait intéressant à développer, consisterait à offrir un choix de différents timbres de voix. Le timbre de la voix est défini par une douzaine de paramètres optionnels, qui devraient être regroupés en banques de valeurs, synthétisant des types de voix particuliers.

CHAPITRE 2

MODÈLE DE SYNTHÈSE DU DIDJERIDU

Le didjeridu, bien que d'une facture relativement simple, est un instrument qui permet de créer une grande variété de timbres très riches. L'idée d'en faire un modèle de synthèse est venu d'un questionnement sur l'intérêt de concevoir un didjeridu dont les caractéristiques physiques évoluent au cours du temps. Par exemple, avec un modèle de synthèse, il serait possible de changer de façon continue la forme de l'instrument. Ce type de transformation est évidemment impossible avec un réel didjeridu.

Le contenu de ce chapitre est divisé en deux sections. Dans la première partie, les caractéristiques acoustiques de l'instrument sont présentées. Cette introduction théorique est complétée par des analyses effectuées sur des échantillons réels des différents modes de jeu du didjeridu. Les échantillons ont été produits sur un instrument accordé en ré par Alexandre Lacroix, ancien membre du groupe Montréalais *les Globe Glotters*. La seconde section explique le développement du modèle de synthèse créé dans le cadre de ce projet.

2.1 Acoustique et analyses

2.1.1 Paramètres structurels du didjeridu

Le didjeridu est considéré comme un cône tronqué et fermé par la bouche de l'instrumentiste, à un des deux bouts. Contrairement au cylindre parfait, dont le diamètre est constant, les modes supérieurs du cône tronqué ne sont pas en rapport harmonique avec le premier mode. Rappelons que la série harmonique est constituée de composantes à des fréquences qui sont des multiples entiers de la fréquence de la note fondamentale (Voir tableau 2.1).

mode	écart en demi-tons	intervalle	note
1	0	fondamentale	do 3
2	+ 12	octave	do 4
3	+ 19	douzième	sol 4
4	+ 24	2 octaves	do 5
5	+ 28	2 octaves + tierce	mi 5
6	+ 31	2 octaves + quinte	sol 5
7	+ 34	2 octaves + septième mineure	si bémol 5
8	+ 36	3 octaves	do 6

TAB. 2.1 – La série harmonique.

Le didjeridu étant un tuyau fermé à un des deux bouts, les modes de résonance suivent une série composée des harmoniques impairs. Par contre, dû à la forme conique du tuyau, le premier mode est particulièrement élevé en fréquence (Fletcher et al., 2001b). Comme le premier mode demeure la fondamentale perçue, le deuxième mode paraît considérablement au-dessous de l'intervalle de douzième correspondant au troisième harmonique de la série du fondamental.

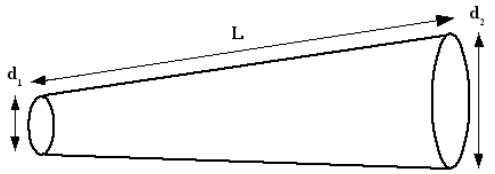


FIG. 2.1 – Dimensions dont dépend l'accord d'un tuyau de forme conique.

La fréquence des modes de résonance d'un tuyau tronqué dépend de la longueur du tuyau, L, ainsi que du diamètre de chaque extrémité, d_1 et d_2 :

$$f_n = (n - 0.5)\frac{c}{4L'}\left[1 + \left[1 + \frac{4(d_2 - d_1)}{\pi^2 d_1 (n - 0.5)^2}\right]^{.5}\right] \tag{2.1}$$

où n est l'ordre du mode recherché, c représente la vitesse de propagation du son (environ 340 m/s dans l'air) et L' est la longueur effective, calculée pour compenser la largeur de l'ouverture de l'instrument d'où le son sort et qui s'exprime comme

suit :

$$L' = L + 0.3\,d_2 \tag{2.2}$$

Si $d_1 = d_2$, l'élément de droite de l'équation 2.1 s'annule et nous obtenons l'équation pour trouver les harmoniques impairs, et justes, du cylindre fermé à une extrémité :

$$f_n = (n - 0.5)\frac{c}{4L'}(1 + 1) \tag{2.3}$$

$$= (n - 0.5)\frac{c}{2L'} \tag{2.4}$$

Le tableau 2.2 illustre les fréquences théoriques d'un tuyau cylindrique d'une longueur de 0,6 m, fermé à une extrémité, pour $c = 340$ m/s :

mode (n)	(f_n =)	fréquence (Hz)	rapport $\frac{f_n}{f_0}$
1	$c/4L$	141.66	1
2	$3c/4L$	425	3
3	$5c/4L$	708.33	5
4	$7c/4L$	991.66	7
5	$9c/4L$	1275	9

TAB. 2.2 – Calcul de la fréquence des modes d'un cylindre fermé à une extrémité.

Cette théorie est confirmée par un calcul théorique, effectué avec la formule déterminant les résonances du didjeridu (Eq. 2.1), pour deux cônes ayant la même longueur et le même diamètre d'entrée mais des diamètres de sortie d_2 différents. Pour un amincissement du cône, alors que la fréquence centrale du premier mode est considérablement abaissée, plus on avance dans l'ordre des modes, moins la différence est grande. En changeant l'ouverture de sortie pour que le cône se rapproche du cylindre, on constate que les modes supérieurs sont en rapport presque juste avec le fondamental (Bélanger and Traube, 2005). Ceci implique que la qualité

harmonique de l'instrument en rapport avec sa résonance fondamentale ne dépend pas uniquement de sa longueur, mais aussi de l'élargissement du cône. C'est un phénomène qui sera intéressant à explorer lors de la modélisation physique. Le tableau 2.3 présente les résultats de ce calcul théorique pour deux didjeridus aux diamètres de sortie différents :

$i_1 : L = 1.29$ m, $d_1 = 0.032$ m, $d_2 = 0.063$ m

$i_2 : L = 1.29$ m, $d_1 = 0.032$ m, $d_2 = 0.042517$ m

			différence	
modes	fréquence i_1	fréquence i_2	Hertz	cents
1	84.5 Hz	72.67 Hz	11.83	261
2	203 Hz	197.7 Hz	5.3	46
3	329.7 Hz	326.4 Hz	3.3	17
4	458.2 Hz	455.8 Hz	2.4	9
5	587.3 Hz	585.4 Hz	1.9	5
6	716.65 Hz	715.1 Hz	1.55	3
7	846.2 Hz	844.9 Hz	1.3	2.6
8	975.8 Hz	974.7 Hz	1.1	2
9	1105.5 Hz	1104.5 Hz	1	1.5
10	1235.2 Hz	1234.3 Hz	.9	1.25

TAB. 2.3 – Influence de la conicité sur la note fondamentale d'un cône tronqué.

La réponse impulsionnelle d'un didjeridu, possédant les mêmes dimensions que le deuxième modèle de l'exemple ci-dessus, a été prélevée en frappant avec la paume de la main sur la plus petite extrémité de l'instrument. Les valeurs que l'on observe à l'analyse spectrale ne concordent pas tout à fait avec le modèle théorique parce que la formule ne représente qu'un modèle idéal, que les ouvertures de l'instrument ne sont pas tout à fait circulaires, qu'un didjeridu réel n'est jamais parfaitement lisse et que l'excitation générée avec la paume de la main n'est pas une impulsion parfaite. Par contre, cette analyse démontre clairement que les modes supérieurs sont plus bas que les harmoniques du premier mode, identifiés par les lignes verticales sur la figure 2.2. Cette série, en ne prenant que les harmoniques impairs, caractériserait

un cylindre fermé à une extrémité.

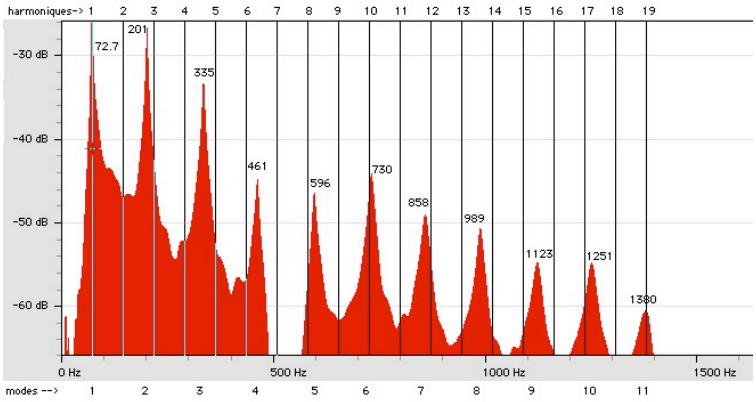


FIG. 2.2 – Les lignes verticales indiquent l'emplacement des harmoniques pour une série dont le fondamental serait le premier mode du tuyau. On constate que les modes supérieurs sont de plus en plus graves, par rapport aux harmoniques impairs théoriques.

Ceci étant dit, précisons que le spectre du didjeridu est strictement harmonique, car la source d'excitation, c'est-à-dire la vibration des lèvres, est harmonique. Les modes de résonance du tuyau ne font que filtrer le signal émis par la source. Le fait que les modes supérieurs du tuyau ne soient pas harmoniquement justes influe seulement sur la puissance de certains harmoniques, et non sur la justesse de ces harmoniques. Si le second mode est excité à l'instar du fondamental, il apparaît avec sa propre série harmonique (Fletcher, 1996). En fait, les modes de résonance du tuyau exercent une influence particulière au moment des transitions, là où se trouve le bruit, car c'est aux moments où l'excitation est constituée de bruit que l'on entend réellement les multiples résonances de l'instrument. À l'aide d'une FFT à court terme aux fenêtres très larges (4096 échantillons), afin d'obtenir une bonne résolution fréquentielle, on peut vérifier que le spectre d'un bourdon sur un didjeridu en ré est bel et bien harmoniquement juste. Sur le sonagramme de la figure 2.3, chaque ligne horizontale représente un harmonique du signal et la puissance de l'harmonique est indiquée par le niveau de gris de la ligne (plus elle est foncée

et plus il y a d'énergie à cette hauteur). On peut constater, en mettant en lien le spectre d'amplitude de la réponse impulsionnelle à la figure 2.2 avec le sonagramme de la figure 2.3, qui proviennent tous deux du même instrument, que les résonances du didjeridu influencent bel et bien l'amplitude des harmoniques de l'excitation. Ainsi, le deuxième harmonique dont la fréquence est de 147 Hz tombe entre le premier et le deuxième mode de l'instrument et ne possède pratiquement pas d'énergie, alors que le dix-neuvième harmonique, à une fréquence de 1387 Hz, coïncide avec le onzième mode du tuyau et possède par conséquent une plus grande puissance.

FIG. 2.3 – Sur l'analyse spectrographique d'un échantillon de bourdon en ré, on constate que les lignes les plus foncées, correspondant aux harmoniques qui possèdent le plus d'énergie, coïncident avec les modes de résonance du tuyau (voir Fig. 2.2).

2.1.2 Paramètres du geste instrumental

2.1.2.1 Vibration des lèvres

Un instrument à colonne d'air a besoin d'un signal d'excitation pour entrer en vibration. Ce signal peut être très bref, de type percussif, ou soutenu, comme c'est souvent le cas pour les instruments à vent. Le didjeridu étant un instrument

sans embouchure, le musicien appuie sa propre bouche pour fermer la plus petite extrémité de l'instrument, et fait vibrer ses lèvres, qui agissent comme une valve qui s'ouvre et se ferme périodiquement, pour créer le signal d'excitation. Il exerce une pression, avec son souffle, sur ses lèvres, et lorsque la pression dépasse un certain seuil, déterminé par la tension musculaire des lèvres, qui elle-même dépend de la hauteur de la note jouée, elles entrent en vibration et génèrent un signal oscillatoire, presque sinusoïdal, qui fait vibrer la colonne d'air de l'instrument. Le joueur accorde la tension de ses lèvres pour que la vitesse de vibration ainsi créée soit proche de la fréquence du premier mode du tuyau. S'il veut exciter le second mode, il doit augmenter la tension de ses lèvres, contraignant ainsi la pression de son souffle à dépasser un seuil plus élevé. C'est pourquoi il ne peut s'attarder longtemps sur les modes supérieurs car la pression à fournir est très exigeante. Voici la formule qui permet de déterminer, en fonction de la pression du souffle et de la résistance des lèvres et de l'instrument, la forme du flux d'air qui vient exciter la colonne d'air :

$$U = \frac{p_0}{R} - \frac{\frac{p_0^2}{R^3}}{(a_0 + a\sin(2\pi f t))^2} \tag{2.5}$$

où p_0 représente la pression du souffle du musicien, et R la résistance à l'entrée de l'instrument. a et a_0 sont des constantes d'amplitude de la fonction sinusoïdale (Fletcher, 1996). L'onde générée par la vibration des lèvres est de type sinusoïdal lorsque la pression dépasse tout juste le seuil où les lèvres entrent en vibration. Plus la pression est forte, plus la forme d'onde tend vers le trapèze, la partie large du trapèze représentant le temps où les lèvres sont complètement fermées. Des observations effectuées à l'aide d'un stroboscope sur la vibration des lèvres d'un joueur de didjeridu, montrent que les lèvres sont complètement fermées durant une grande fraction de chaque cycle (Fletcher et al., 2001b).

Une forme d'onde trapézoïdale contient beaucoup d'énergie aux harmoniques supérieurs, ce qui permet à un bon joueur d'utiliser les variations de pression pour

modifier le contenu spectral de l'excitation. Le fait qu'il y ait beaucoup d'énergie dans les hautes fréquences permet d'entendre clairement les changements de configuration du conduit vocal, qui se traduisent par un déplacement des formants.

Les lèvres ne peuvent atteindre un état de vibration stable, à la bonne fréquence, instantanément. Elles atteignent leur fréquence de vibration en suivant une rampe partant de zéro et atteignant la fréquence désirée en une centaine de millisecondes, approximativement. C'est la phase d'attaque du son de didjeridu. À tous les instants de la montée, la série harmonique générée par cette vibration change et provoque beaucoup de variations aux fréquences où les harmoniques de la source coïncident avec les résonances du tuyau, ce qui se traduit auditivement par un caractère très bruité. La figure 2.4 présente un spectre d'attaque provenant de l'analyse effectuée sur les 100 premières millisecondes d'un son de didjeridu.

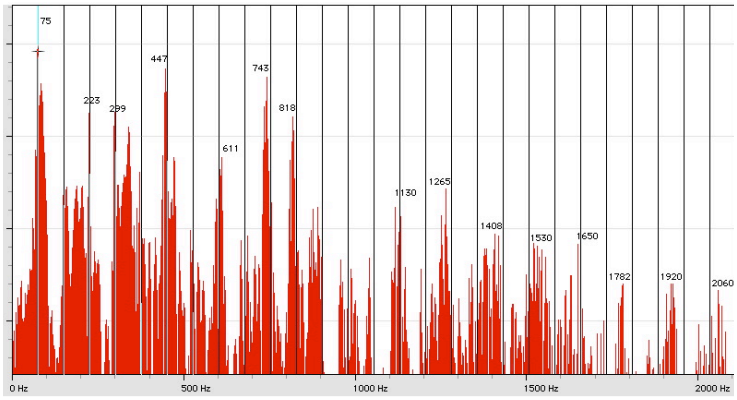


FIG. 2.4 – Spectre des 100 premières millisecondes. Malgré le filtrage sélectif du tuyau, on constate l'instabilité de la vibration.

Le fait est qu'un musicien qui contrôle bien son jeu peut, à n'importe quel moment, créer ce type de variations en modifiant la fréquence de vibration de ses lèvres pour générer de nouvelles combinaisons spectrales. C'est ce qui crée tout l'intérêt du didjeridu : la gamme des effets spéciaux est très vaste et relativement facile à contrôler. Comme l'excitation et le conduit vocal sont intimement liés l'un

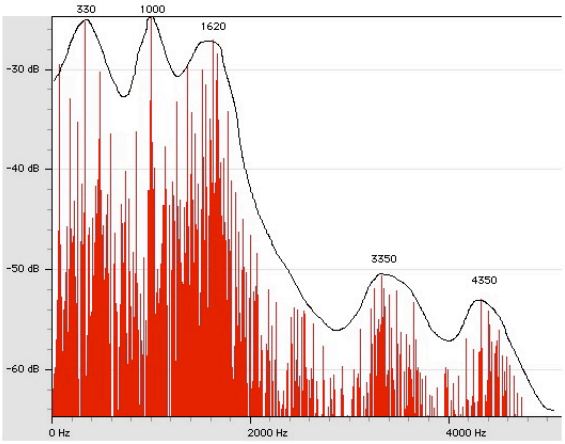


FIG. 2.5 – Spectre d'amplitude d'un son obtenu en désaccordant la fréquence de vibration des lèvres par rapport au premier mode de l'instrument. Il diffère de l'attaque en ce sens que l'on aperçoit des formants qui sont probablement dus au conduit vocal du joueur. La voyelle prononcée est [a].

à l'autre, le musicien peut varier la vitesse de vibration de ses lèvres et modifier, en même temps, la forme du conduit vocal pour donner un son très riche en partiels et filtré subtilement par l'action conjointe du conduit vocal et du tuyau. La figure 2.5 donne un exemple, sur une période de 400 ms, du spectre généré par l'action combinée du mouvement des lèvres et des résonances du conduit vocal.

2.1.2.2 Influence de la cavité buccale

La génération de sons d'un instrument à vent consiste en un contrôleur de flux (les lèvres) couplé à deux colonnes d'air, celle de l'instrument et celle du musicien. Le mouvement oscillatoire des lèvres est influencé par les résonances harmoniques de ces deux colonnes d'air. Les musiciens utilisent le résonateur que forme leur colonne d'air pour corriger la justesse de la note, ou renforcer et embellir le son de l'instrument en s'accordant sur la fréquence de la note jouée.

Il est possible de faire varier la fréquence du résonateur, en déplaçant la langue, sur un registre entre 500 et 3000 Hz. Une des caractéristiques particulières du son du didjeridu, c'est que les régions formantiques de la cavité buccale du musicien, et

toutes les modifications qu'il sait y apporter, provoquent des changements continus dans le spectre acoustique de la source. Les formants sont plus prononcés dans le cas du didjeridu que dans celui des cuivres car l'embouchure des trompettes, tuba et autres, forme une cavité tampon entre la bouche et l'instrument et amoindrit les pics formantiques. La voix n'est pas seulement un simple résonateur, mais un générateur de sons complexes permettant une grande variabilité de contrôle. Les cris, la parole et le chant modifient la colonne d'air de façon extrêmement subtile et offrent une gamme d'effets spéciaux très intéressants lorsqu'utilisés en jouant du didjeridu. L'oscillation des lèvres en soit ne provoque qu'un son constant sans grande variation, mais lorsque le musicien utilise des sons vocaux tout en jouant, une modulation d'amplitude s'opère entre la voix et l'excitation non linéaire des lèvres et celle-ci est ensuite injectée dans l'instrument. La forme d'onde de l'excitation est alors beaucoup plus complexe, comportant une grande quantité de variations, le tout filtré par le résonateur qu'est le didjeridu.

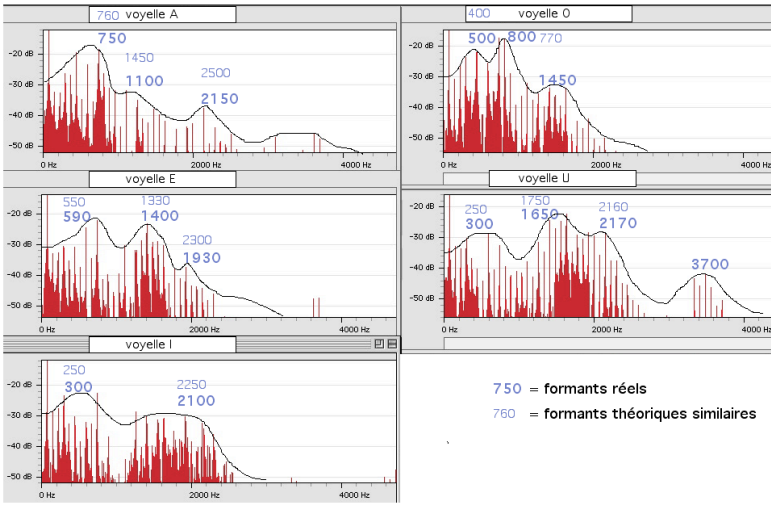


FIG. 2.6 – Spectres d'amplitude d'un bourdon avec une voyelle prononcée. Les nombres en petits caractères représentent les formants théoriques lorsqu'ils coïncident avec les maxima (caractères gras) observés à l'analyse.

Nous avons demandé au musicien de produire un simple bourdon en prononçant

la suite de voyelles [a - e - i - o - u] afin de comparer les fréquences des formants. La suite de spectres d'amplitude pour chaque voyelle de la figure 2.6 démontre clairement qu'il y a concordance entre le son perçu et la voyelle prononcée lors du jeu. Les chiffres en caractère gras correspondent aux formants observés à l'analyse. Lorsque l'analyse concorde avec les formants théoriques, ceux-ci sont inscrits en petit caractère sur la figure. On constate que pour chacune des voyelles, au moins deux formants sont similaires aux formants théoriques d'un ténor francophone.

2.1.2.3 Ajout du chant

Lorsque le joueur chante dans l'instrument tout en produisant un bourdon, c'est-à-dire une note pédale à fréquence fixe, le flux d'air venant des poumons est modulé par les cordes vocales (à la fréquence f_1), filtré ensuite par le conduit vocal et modulé à nouveau par la vibration des lèvres (généralement une fréquence plus basse, f_2), ce qui produit les partiels de fréquence $mf_1 \pm nf_2$, m et n étant des entiers, résultant d'un phénomène d'intermodulation. Il est ainsi possible de générer des sous-harmoniques en chantant des notes selon certains rapports en fonction de la fréquence du bourdon.

Un exemple typique a lieu lorsque le joueur chante une dixième au-dessus du bourdon (rapport de 5/2). Apparaît alors une note un octave sous le bourdon naturel de l'instrument (Fletcher, 1996).

$$\begin{aligned} f_1 &= 2.5f_2 \\ f_{m,n} &= mf_1 \pm nf_2 \\ f_{1,2} &= f_1 - 2f_2 \\ &= 2.5f_2 - 2f_2 \\ &= 0.5f_2 \end{aligned}$$

Chanter une note de la série harmonique ne provoque pas de grands change-

ments, si ce n'est que de renforcer la région du spectre où se trouve cette harmonique. Par contre, si le musicien possède une bonne maîtrise de sa voix, il peut s'éloigner légèrement de la note fondamentale, ce qui fait varier intensément le contenu spectral du son, puisque le musicien modifie sa colonne et s'accorde sur des fréquences très différentes du fondamental de l'instrument. C'est une autre façon de provoquer des phénomènes de battements.

La figure 2.8 représente l'analyse spectrale des quatre régions indiquées sur le sonagramme de la figure 2.7. La région A correspond à la partie stationnaire, le bourdon juste avant les variations. En B, nous avons la montée lorsque le joueur commence à chanter dans l'instrument. C et D nous donnent deux spectres obtenus lorsque le joueur stabilise sa voix sur une note qui n'est pas celle du drone. Cette analyse a été effectuée sur un didjeridu accordé en La.

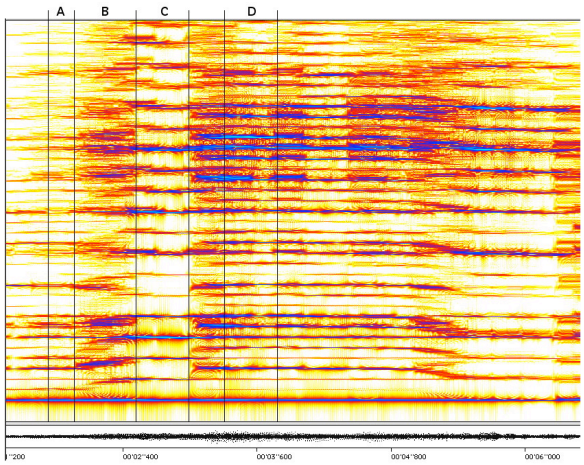


FIG. 2.7 – On constate une montée et une descente lorsque la voix du musicien se détache de la fréquence de l'excitation à laquelle elle vient s'ajouter. A = partie stationnaire. B = montée de la note chantée. C et D = parties stationnaires avec chant.

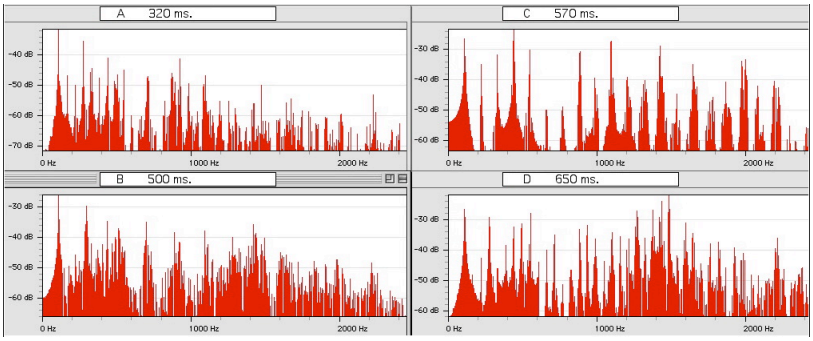


FIG. 2.8 – Spectres d'amplitude correspondant aux régions A, B, C et D, respectivement, du sonagramme de la figure 2.7.

2.1.2.4 Respiration circulaire

Pour exécuter la respiration circulaire, le joueur expire l'air de façon normale jusqu'au moment où il a besoin de faire une nouvelle provision d'air. Il emplit alors ses joues d'air et isole sa bouche des voies respiratoires par un rapprochement de la langue et du palais. Alors qu'il respire brièvement par le nez, il continue à projeter de l'air en contractant ses joues. Il reprend ensuite sa position normale et le cycle continue. Les formants changent de façon importante lorsque la bouche est isolée des voies respiratoires.

Durant le cycle normal, tout le conduit vocal est connecté aux lèvres tandis que durant la partie où il respire, seule la bouche exerce une influence sur le son (Fletcher et al., 2001a).

Les effets de la respiration circulaire se manifestent à plusieurs niveaux : celle-ci provoque une pulsation du bourdon, une variation rythmique sur la pression du souffle (contenu harmonique de l'excitation), ainsi qu'une variation rythmique de la forme du conduit vocal (mouvement des formants).

En appliquant le procédé de la respiration circulaire, le musicien peut tenir une note pendant plusieurs minutes sans interrompre le son. Il doit cependant respirer très rapidement par petits coups secs, ce qui provoque inévitablement un bruit de respiration qui s'entend très bien, puisqu'il se manifeste dans une région spectrale

où le didjeridu n'agit pratiquement plus, c'est-à-dire les hautes fréquences. Les bons joueurs de didjeridu respirent de façon régulière pour marquer rythmiquement leur musique. Le sonagramme de la figure 2.9 démontre bien ce qui se passe. En se contractant, les joues diminuent le volume de la cavité buccale, lui conférant ainsi une fréquence de résonance plus élevée, ce qui vient renforcer certains harmoniques au moment de la respiration.

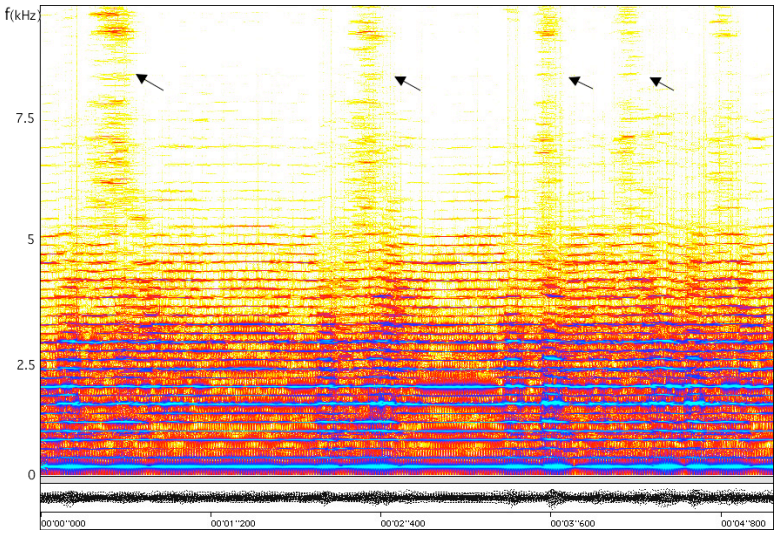


FIG. 2.9 – Rythme respiratoire aux fréquences élevées et variations d'énergie aux différents harmoniques au moment de la respiration.

2.1.2.5 Excitation des modes supérieurs

La note fondamentale constitue le premier mode de l'instrument, c'est-à-dire le bourdon, la note que le musicien tient presque tout le temps. Les analyses spectrographique démontrent que lorsque le musicien augmente la pression de son souffle pour exciter les modes supérieurs, il n'y a pas de changement harmonique radical. Le premier mode reste toujours présent mais, il se crée une augmentation notable d'énergie entre le septième et le trentième harmonique, avec beaucoup de variations internes. Le fait que les modes de résonance de l'instrument ne soient pas en

relation harmonique importe peu, puisque les variations se produisent à l'intérieur de la série harmonique du premier mode. En variant la pression exercée sur ses lèvres, le musicien ne fait que varier les amplitudes de chacun des harmoniques présents dans le signal, ce qui crée les variations de hauteur spectrale perçues par l'auditeur. La pression à exercer pour exciter les modes supérieurs est très élevée, ce qui limite le musicien à ne s'en servir que pour de courtes périodes, car il doit déployer beaucoup d'énergie pour maintenir cette vibration. Le sonagramme de la figure 2.10 donne une allure générale de ce qui arrive lorsque les modes supérieurs sont excités. Il y a un cycle de forte concentration d'énergie entre les harmoniques 7 et 30.

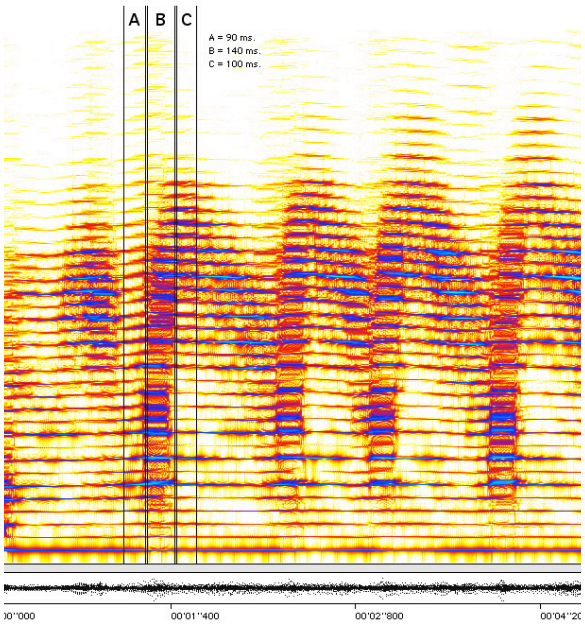


FIG. 2.10 – Excitation des modes supérieurs : la note fondamentale reste toujours présente mais l'énergie se déploie entre les harmoniques 7 et 30.

La figure 2.11 présente les spectres d'amplitude des régions A, B et C de la figure 2.10, correspondant aux trois états du mouvement en jeu. La section A, d'une durée de 90 ms, représente l'état stationnaire du bourdon. La section B, durant 140 ms,

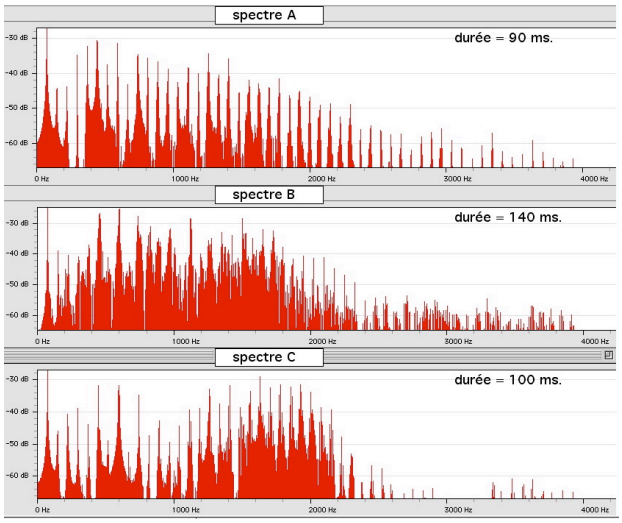


FIG. 2.11 – Spectres A,B et C, correspondant respectivement à un drone stationnaire, un changement vers l'excitation des modes supérieurs et une excitation stabilisée sur les modes supérieurs.

est la conséquence d'une montée en fréquence de la vibration des lèvres, un peu à l'image de l'attaque d'une note, où se produit la rencontre de plusieurs spectres consécutifs avec le spectre du tuyau. La section C correspond à l'état stationnaire sur les modes supérieurs, qui dure au plus 100 ms, avant de redescendre lentement vers un état stable sur le bourdon sans s'y rendre tout à fait puisque les modes supérieurs sont à nouveau attaqués et le cycle recommence.

2.1.2.6 Cris

Un dernier mode de jeu consiste à crier dans l'instrument tout en produisant le bourdon. Crier dans un didjeridu demande beaucoup d'énergie car il faut utiliser au maximum l'air accumulé et se dépêcher de faire de nouvelles réserves, mais c'est sans aucun doute l'effet le plus intéressant d'un bon joueur de didjeridu. Nous sommes sensibles à ce type de sons car ils sont très organiques et évoquent souvent le cri des animaux. La gamme de sons possibles est très large. Les cris font apparaître des zones d'énergie tout à fait nouvelles dans le spectre, au-dessus

des résonances normalement excitées par la vibration des lèvres. Ils confèrent aussi beaucoup d'énergie aux harmoniques déjà présents, ce qui se traduit par une montée d'intensité spectaculaire. Le sonagramme de la figure 2.12 illustre les changements dans le spectre du didjeridu lorsque le musicien pousse un cri dans l'instrument.

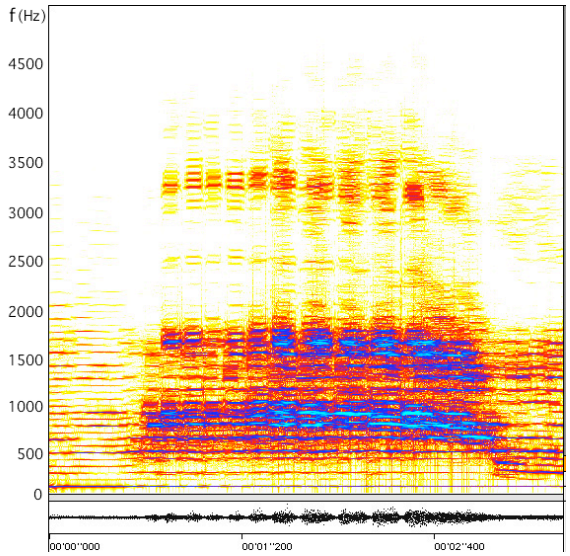


FIG. 2.12 – Crier dans le didjeridu tout en produisant un bourdon constant provoque une montée subite d'énergie dans le registre médium (500 à 2000 Hz), ainsi qu'une nouvelle zone d'énergie dans les hautes fréquences, qui apparaît uniquement lorsque l'excitation est très puissante.

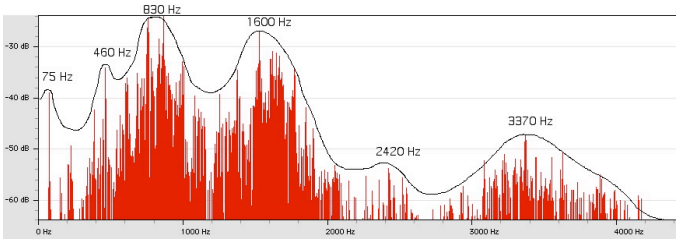


FIG. 2.13 – Spectre d'amplitude du cri sur la voyelle [a] de la figure 2.12. On remarque des formants bien prononcés.

À l'écoute du son produit par ce cri dans le didjeridu, on peut avoir une forte impression que le joueur prononce la voyelle [a]. Effectivement, les formants théoriques

pour la voyelle [a], chantée par un ténor francophone, se trouvent autour de 760, 1450, 2600 et 3300 Hertz. Ce qui correspond aux formants observés sur le spectre d'amplitude de la figure 2.13.

2.2 Synthèse

Le modèle créé et utilisé pour ce projet consiste en une synthèse modale du didjeridu. Ce modèle est construit en deux parties. La première est la source d'excitation avec un contrôle sur la fréquence fondamentale et la pression du souffle. La deuxième est constituée d'un banc de filtres en parallèle, simulant les résonances de l'instrument, avec contrôle sur la fréquence du premier mode et sur la longueur du tuyau (voir fig. 2.14). Nous verrons tout d'abord le modèle d'excitation, ensuite la modélisation du tuyau et de ses modes de résonances.

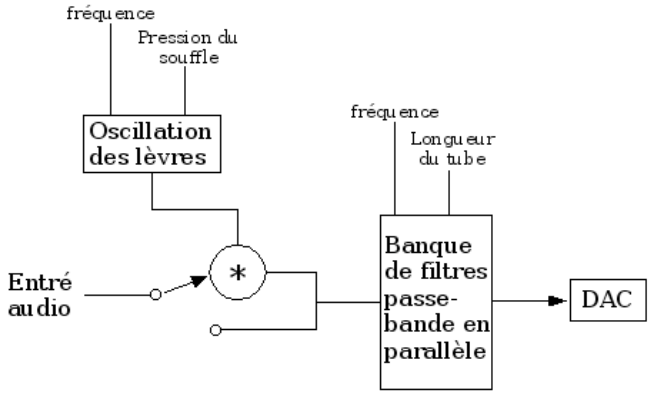


FIG. 2.14 – Diagramme simplifié d'une synthèse source-filtre du didjeridu.

2.2.1 Source d'excitation : vibration des lèvres

Le mouvement des lèvres constitue un élément clé d'une bonne implémentation d'un modèle de didjeridu. Tous les effets spéciaux sont d'abord modulés par cette oscillation avant d'être injectés dans le tuyau. La formule donné par Fletcher (éq. 2.5) pour déterminer l'onde de vibration est très instable au niveau de l'amplitude

par rapport à la pression des lèvres, et il s'est avéré impossible de la contrôler sans générer plusieurs artefacts indésirables. Le défi était de trouver un algorithme générant un signal périodique dont la forme d'onde est modifiable au moyen d'un seul paramètre afin de simuler la variation de la brillance spectrale tel qu'observée en fonction de la pression de l'excitation.

Un modèle a été élaborée à partir d'une onde sinusoïdale et d'un module de distorsion non-linéaire. Lorsque la pression est faible, le train d'onde a la forme d'un sinus, avec une amplitude moyenne. Une fonction d'écrêtage est appliquée à l'onde en fonction de l'augmentation de la pression du souffle sur les lèvres. Un pic se forme et l'onde devient de plus en plus trapézoïdale (voir Fig. 2.15). Cette fonction simule bien la réalité et est extrêmement simple à manipuler.

Ce train d'onde constitue l'excitation du modèle. Tout comme le musicien ajuste la fréquence de vibration de ses lèvres sur la fréquence fondamentale du didjeridu, la fréquence de ce train d'onde est asservie à la fréquence fondamentale du tuyau. De légères variations aléatoires sont appliquées à la pression du souffle et à la fréquence de l'onde pour conférer une allure plus naturelle au modèle.

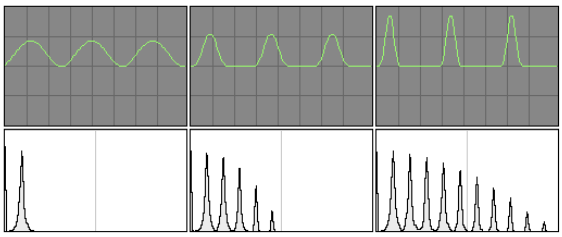


FIG. 2.15 – Train d'impulsions généré par les lèvres. En haut, la forme d'onde générée et en dessous, le spectre d'amplitude correspondant. De gauche à droite, pour des valeurs de pression de 1, 4 et 15 sur une échelle de 1 à 20.

2.2.2 Modélisation de la cavité buccale

Le spectre du souffle modulé en amplitude par la vibration des lèvres, avant d'être filtré par le tuyau, est influencé par l'allure du résonateur que constitue la cavité buccale. En modifiant le volume de sa bouche, le musicien déplace un formant très prononcé dans le spectre de l'excitation. C'est, notamment, le déplacement de ce formant que l'on entend lorsque le musicien respire tout en poussant l'air avec ses joues. Cet effet est souvent utilisé pour rythmer la musique. L'implémentation est très simple, elle consiste à rajouter une bosse dans le spectre de l'excitation, avec un filtre d'égalisation, avant de l'envoyer dans le tuyau. D'autres formants, à l'influence plus subtile, peuvent aussi être rajoutés afin de créer des bosses dans le registre aigu du spectre de l'excitation.

2.2.3 Effets spéciaux (excitations externes)

Nous avons maintenant la base de ce qui constitue l'excitation simple du didjeridu, un bourdon avec un formant très prononcé. Afin de créer les effets spéciaux propres au style de jeu, il faut modifier le signal excitateur des lèvres en le modulant avec un autre signal complexe. La voix, par exemple, est le signal que le musicien va utiliser pour créer des variations dans son jeu. Le modèle de synthèse, présenté ici, possède une entrée audio par laquelle on introduit un signal externe dans le mécanisme de production sonore de l'instrument. On peut y brancher un micro et chanter dans l'instrument, ce qui donne un son de didjeridu très réaliste. Pour une synthèse complète du joueur de didjeridu, on peut aussi y injecter une synthèse vocale ! Ça peut être un fichier sonore, du bruit filtré, de la granulation... Tout ce qui entre par cette entrée est modulé en amplitude avec le mouvement oscillatoire des lèvres pour créer un signal au spectre très riche, qui se contrôle très bien lorsqu'utilisé avec une voix chantée.

Cette excitation est celle utilisée pour faire vibrer le modèle de tuyau qui sera

maintenant développé.

2.2.4 Modes de résonance du tuyau

Dans la conception de ce modèle de synthèse, il y avait plusieurs choix à faire. En ce qui concerne les modes de résonance du tuyau, c'est-à-dire les bosses dans le spectre d'amplitude de l'instrument, il est permis à l'utilisateur de spécifier le nombre de modes désirés. Ces modes de résonance sont implémentés avec un banc de filtres résonants en parallèle, dont les fréquences centrales, les amplitudes et les largeurs de bande sont variables. Le compromis se trouve donc entre brillance et exigence en temps de calcul. Un tuyau comportant entre 24 et 32 modes de résonances génère un son de didjeridu réaliste. Il est possible d'ajouter ou d'enlever des résonances afin de créer des effets de filtrage s'éloignant du son naturel du didjeridu.

Chacun des modes est constitué d'une cascade de deux filtres récursifs du second ordre. Cela a pour effet d'accentuer la résonance du filtre et d'éliminer les composantes spectrales indésirables qui étaient présentes avec un seul filtre. Les fréquences centrales sont déterminées selon une formule qui tient compte de la longueur du tuyau et d'un facteur d'élargissement du cône pour une fréquence fondamentale donnée. Une version antérieure du modèle permettait de varier la fréquence fondamentale en modifiant la longueur du tube, mais l'effet n'était pas convainquant.

Le diamètre d_1 est fixé à l'initialisation du programme. Le diamètre d_2 varie automatiquement en fonction de la longueur et de la fréquence fondamentale de l'instrument. De la formule donnant la fréquence fondamentale, f_0, en fonction de d_1, d_2 et L', (eq. 2.1) nous avons déduit la formule donnant le diamètre d_2 en fonction de d_1, L' et f_0 :

$$d_2 = d_1 + \frac{\pi^2 d_1}{16}\left[\left(\frac{8 f_0 L'}{c} - 1\right)^2 - 1\right] \tag{2.6}$$

On décide donc d'une fréquence fondamentale, entre 60 et 100 Hz pour un instrument réaliste, et le modèle nous indique quelle serait la longueur d'un cylindre ayant cette fondamentale. Ensuite, on peut varier la longueur à volonté et le diamètre de sortie du cône, d_2, se réajuste constamment pour conserver la fréquence fondamentale intacte, préservant ainsi l'effet du bourdon. Ce qui est intéressant avec cette méthode, c'est que tous les formants se déplacent sur l'axe des fréquences, tandis que le premier reste en place, produisant un balayage spectral fort appréciable et beaucoup plus subtil qu'un déplacement du bourdon. On modifie le timbre tout en préservant la hauteur de la note.

Les valeurs d'amplitude sont initialisées afin d'obtenir un instrument réaliste, c'est-à-dire selon une courbe plus ou moins exponentielle, mais il est possible d'envoyer vingt-quatre valeurs sous forme de liste et de les varier en temps réel pour obtenir des sonorités surprenantes. Cette option permet de modeler le son avec précision et d'obtenir des textures qui s'éloignent considérablement du didjeridu.

Un autre effet intéressant, est obtenu en appliquant un *jitter* sur la largeur de bande des filtres. La largeur de bande est par défaut très serrée, ce qui donne un effet de vocodeur à l'instrument. On peut contrôler le degré de variance du jitter ainsi que la vitesse des changements, avec pour effet sonore un espèce de bouillonnement du tube subtil et agréable.

L'excitation est envoyée dans chacun des filtres modélisant les modes de résonance du tuyau, et les sorties de tous ces filtres sont additionnées et acheminées vers la sortie audio du modèle.

2.3 Conclusion

La modélisation d'un instrument comme le didjeridu permet de créer des sons de synthèse d'une grande richesse sur le plan du timbre, tout en offrant l'avantage de pouvoir modifier les caractéristiques physiques de l'instrument en temps-réel. Il est donc possible, et relativement facile, de faire une transition fluide entre deux didjeridus de différentes dimensions. Il est aussi possible de simuler un instrument dont les caractéristiques physiques sont pratiquement impossible à réaliser. Par exemple, avec un modèle physique, rien n'empêche de créer un didjeridu de 10 mètres de long, et de simuler des lèvres de géant, nécessaires pour faire vibrer une telle colonne d'air.

Un effet intéressant, possible avec un modèle de synthèse, consiste à ajuster automatiquement la longueur du tuyau, en fonction d'un changement de diamètre de l'ouverture de sortie, pour conserver un premier mode de fréquence constante. Le bourdon garde ainsi toujours la même fréquence, mais les modes supérieurs subissent une variation de hauteur due au changement de forme de l'instrument. Cet ajustement permet de créer plusieurs spectres de partiels pour un bourdon constant.

Un autre avantage de la modélisation réside dans le fait que les paramètres de la synthèse font référence aux caractéristiques physiques du corps sonore, ainsi qu'aux modes de jeu propres à l'instrument simulé. Les variables sont donc très évocatrices et permettent de prévoir précisément le type de résultat sonore provoqué par des changements de paramètres. La suite de ce travail de recherche sur les modes de jeu du didjeridu, et leur contribution dans le timbre de l'instrument, consiste donc à élaborer une suite d'algorithmes qui simuleraient de façon réaliste, ou fantaisiste, les différentes techniques de jeu du didjeridu. Ces algorithmes viendront modifier judicieusement les différents paramètres du modèle de synthèse pour créer un joueur virtuel capable d'improviser intelligemment.

CHAPITRE 3

OUNK

3.1 Objectifs

La création du logiciel Ounk (Bélanger, 2008) a été motivée par le besoin d'un environnement de programmation musicale offrant une excellente qualité sonore accompagnée d'un langage de haut niveau, simple et efficace. L'environnement doit permettre à l'utilisateur d'arriver rapidement au résultat sonore souhaité. Le langage Csound[1] comme moteur audio fut choisi pour plusieurs raisons. Premièrement, Csound est un langage mature ayant une communauté de programmeurs très active. En plus d'offrir une très large librairie d'opérateurs, la précision de l'interpolation produit un rendu sonore d'excellente qualité. Csound permet aussi de calculer les échantillons en temps réel ou en temps différé, en créant un fichier son sur le disque dur, ce qui permet de construire des structures sonores dont la complexité n'est pas limitée à la puissance du processeur.

Par contre, la syntaxe du logiciel Csound est un peu désuète et rébarbative. Ounk nécessitait une syntaxe moderne, claire et efficace pour l'écriture des scripts. Le choix du langage de programmation Python[2] s'est imposé principalement pour la simplicité de sa syntaxe, sa librairie de fonctions riche et variée, le fait qu'il est multiplateforme et les facilités d'interfaçage qu'il fournit. Développé au début des années 90, par Guido Van Rossum, Python est un langage dit de haut niveau relativement jeune, en constant développement et bénéficiant du soutien d'une communauté très active. Depuis quelques années, il est de plus en plus utilisé dans le monde de la programmation de logiciels et d'interfaces web. Python est un excellent langage pour construire des algorithmes puissants et versatiles.

[1] http ://www.csounds.com
[2] http ://www.python.org

Le tandem Python et Csound combine la puissance du langage de programmation Python à la qualité du moteur audio Csound pour offrir un environnement de programmation musicale agréable à utiliser, que ce soit pour la construction d'interfaces graphiques, l'exploration des techniques de traitement sonore, le multitâche ou la communication avec d'autres logiciels.

3.2 Structure

L'interface graphique de Ounk consiste principalement en un éditeur de texte facilitant l'écriture des scripts ainsi que la gestion de projets musicaux complexes. Un projet, comprenant plusieurs fichiers ressources tels que des sons, des bases de données ou des fichiers MIDI, peut être géré via l'interface graphique, qui assurera les liens entres les différentes composantes du projet. L'éditeur offre des fonctions communes aux éditeurs de texte voués à la programmation, par exemple la colorisation des mots-clés (avec une adaptation particulière pour les langages Python et Csound), l'auto-complétion et la mise en commentaire de morceaux de code. La librairie de fonctions, classées par catégories, est facilement accessible et un panneau de documentation fournit des informations essentielles à l'utilisation des fonctions propres à Ounk.

Plusieurs fenêtres peuvent être ouvertes simultanément dans l'éditeur. Chacune possède sa propre console où seront interprétées les commandes du script. C'est aussi par le biais de cette console que l'utilisateur pourra interagir avec un script en cours d'exécution.

L'usage principal du logiciel consistera donc à écrire des scripts musicaux en utilisant la librairie de fonctions fournie par l'environnement. Chaque fonction propre à Ounk a pour tâche d'écrire une séquence d'instructions Csound effectuant un processus algorithmique ou sonore précis. Au moment de lancer le script, chaque commande sera envoyée à l'interpréteur Python qui construira un programme Csound

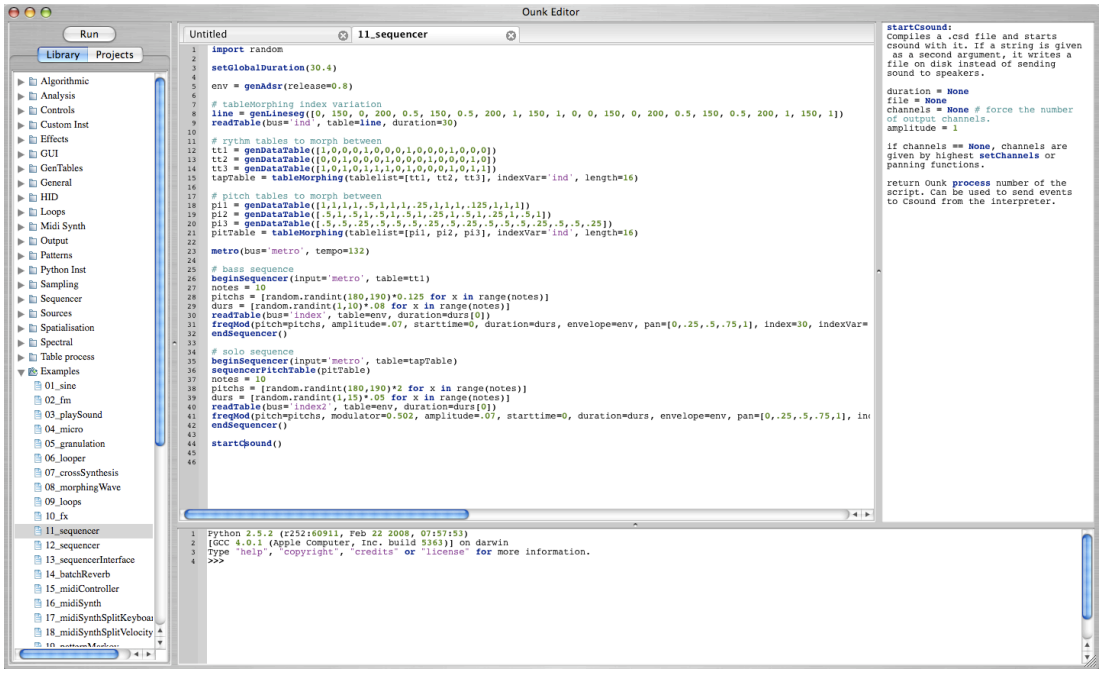


FIG. 3.1 – Interface d'édition du logiciel *Ounk*.

contenant toute l'information nécessaire à l'accomplissement de l'idée musicale exprimée. Le logiciel Csound sera ensuite appelé pour exécuter la génération des échantillons sonores. C'est la fonction *startCsound* qui met tous les morceaux de code en place et qui appelle Csound. Cette fonction doit donc être présente dans tous les scripts Ounk générant du son. Une liste des fonctions, regroupées par catégorie, peut être consultée à l'annexe II.

3.3 Particularités du langage

Le principal objectif de ce projet était de construire un environnement simple et cohérent sans limiter le pouvoir d'action de l'utilisateur. Nous verrons les principaux automatismes mis en place pour assurer une communication limpide entre le script écrit en Python et l'exécution par le logiciel Csound.

3.3.1 Attributs audio

Csound doit être configuré en fonction du système sur lequel il est installé afin de fonctionner correctement. Il faut spécifier, entre autre, l'interface audio qui doit être utilisée. D'autres paramètres importants sont la fréquence d'échantillonnage et la fréquence de la boucle de contrôle qui indique à quel taux les données de contrôle seront rafraîchies. Un fichier de configuration peut être édité afin d'optimiser les performances en fonction du système utilisé. Ce fichier est accessible via le menu 'File', puis 'Preferences...'. Des valeurs par défaut sont fournies en fonction des différentes plates-formes. Il est aussi possible de modifier la configuration audio localement à l'intérieur d'un script. Les valeurs par défaut ne seront pas modifiées et le comportement des autres scripts ne sera donc pas affecté. Les fonctions *setAudioAttributes* et *setAudioDevice* permettent de modifier localement la configuration audio de Csound.

```
setAudioAttributes(samplingrate = 48000,
                   controlrate = 4800,
                   softbuffer = 256,
                   hardbuffer = 512,
                   audioDriver = 'jack')
setAudioDevice(inumber = '', onumber = 3)
```

3.3.2 Gestion des canaux de sortie

Ounk permet de contrôler des processus audio multicanaux. La fonction *setChannels* permet de définir le nombre de canaux à utiliser et chaque fonction en tiendra compte dans la gestion des entrées et des sorties audio. Les fonctions traitant les flux d'échantillons audio de Ounk sont conçues de façon à générer automatiquement le nombre de canaux audio requis, ce qui, par exemple, permet de convertir aisément une musique stéréophonique en une musique octophonique. La valeur de panoramisation[3] est toujours spécifiée entre 0 et 1, et est automatique-

[3]Le paramètre «pan» est présent pour toutes les fonctions générant une sortie audio.

ment échelonnée en fonction du nombre de canaux de sortie requis par la fonction. La fonction *setChannels* peut être appelée plus d'une fois dans un même script, permettant de calculer certaines fonctions en monophonie et d'économiser des cycles de calcul. Pour acheminer un signal monophonique sur un canal spécifique, on utilisera la fonction *directOut* en indiquant le numéro du canal au paramètre «offset». Un groupe de fonctions de panoramisation, permettant entre autre d'élargir le nombre de haut-parleurs affectés par le signal, est disponible pour gérer l'emplacement des sons dans l'espace. Le nombre de canaux le plus élevé parmi les appels de *setChannels* ou les fonctions de panoramisation déterminera le nombre de canaux de sortie de Csound.

```
# modulation en anneau générée en monophonie
setChannels(1)
sine(pitch = 100, out = 's1')
sine(pitch = 25, amplitude = 0.0004, out = 's2')
ringMod(in1 = 's1', in2 = 's2', out = 'ring')
# panoramisation sur les 2 premiers canaux
pan1to2(input = 'ring', pan = .25)
# diffusion d'un son en quadraphonie sur les 4 premiers canaux
setChannels(4)
playSound(sound = 'mon_son_quad.aif', pan = 0)
```

3.3.3 Gestion du temps

Chaque fonction générant un processus Csound possède les paramètres «start-time» et «duration», avec comme valeurs par défaut 0 et **None**. **None** est la valeur nulle en Python. Les fonctions sont donc définies pour démarrer au début de l'exécution du script et n'ont pas de durée prédéterminée, ce qui n'est pas strictement légal, puisque Csound doit savoir combien de temps un processus doit être activé. Afin de ne pas avoir à spécifier une durée pour chaque fonction, une durée globale est précisée en début de script, et chaque fonction ayant la valeur **None** au paramètre «duration» prendra la valeur de la durée globale. Par défaut, cette valeur est de -1, ce qui indique à Csound de jouer indéfiniment. Dans cette situa-

tion, il revient à l'utilisateur d'interrompre l'exécution du script lui-même. On peut modifier la durée globale, spécifiée en secondes, avec la fonction *setGlobalDuration*. Dans l'exemple ci-dessous, le son «baseballmajeur.aif» jouera en boucle pour toute la durée de l'exécution tandis qu'une centaine de sinus feront des interventions de 250 ms avec des temps de départ choisis aléatoirement.

```
setGlobalDuration(60)
playSound(sound = 'baseballmajeur.aif', loop = True)
for i in range(100):
    start = random.randint(0,59)
    pitch = random.randint(250,4000)
    sine(pitch = pitch, starttime = start, duration = .25)
```

3.3.4 Utilisation de la liste comme valeur de paramètre

Chaque fonction de la librairie Ounk demande un certain nombre de paramètres pour définir son comportement. Par exemple, le paramètre «sound» de la fonction *playSound* indiquera le lien vers un fichier son sur le disque dur. Ainsi, la ligne suivante créera un instrument qui jouera le son «baseballmajeur.aif» en boucle pour toute la durée de la performance.

```
playSound(sound='/home/olipet/sons/baseballmajeur.aif', loop=True)
```

Une des caractéristiques de Ounk en faisant un environnement de composition puissant est la possibilité d'utiliser une liste comme valeur de paramètre. C'est une méthode généralisée s'appliquant à tous les paramètres afin de décupler les processus sonores sans avoir à répéter sans cesse le même code. Cette technique est grandement inspirée du «multichannel expansion», présent dans le logiciel de programmation musicale *SuperCollider*, permettant de spécifier une liste de valeurs à un paramètre d'une fonction. Chaque élément de la liste créera une instance de la fonction en utilisant sa propre valeur comme paramètre. Dans *SuperCollider*, les différentes instances seront automatiquement acheminés vers des canaux de sortie consécutifs (McCartney, 1998).

Ainsi, pour faire jouer en même temps tous les sons contenus dans un dossier, on exécuterait le code suivant :

```
# donne une liste de tous les fichiers dans un dossier
snds = os.listdir('/home/olipet/sons')
playSound(sound = snds, loop = True)
```

Une fonction qui reçoit une liste comme valeur d'un de ses paramètres créera autant d'instances de l'instrument qu'il est nécessaire pour calculer le rendu sonore. Si, pour une même fonction, plus d'un paramètre reçoit une liste en entrée, la plus longue liste sera utilisée pour générer le nombre d'instances nécessaires. Les valeurs des listes plus courtes seront utilisées dans l'ordre, avec bouclage lorsque la fin de la liste est atteinte. Utiliser des listes de différentes longueurs pour les paramètres d'une fonction produira un résultat sonore riche et varié puisque les combinaisons de paramètres seront différentes pour chaque instance de l'instrument.

Pour toutes les fonctions générant des échantillons audio, seul le paramètre «out» n'accepte pas de liste. Ce paramètre permet de diriger le signal vers d'autres fonctions et ne peut être décuplé à l'intérieur d'un appel de fonction. Pour assigner la sortie d'une fonction sur plusieurs canaux audio, il suffit de réécrire la fonction, ou de la placer dans une boucle, en changeant la valeur du paramètre «out».

3.3.5 Passage des valeurs de contrôles

Comme beaucoup de moteurs audio, Csound gère les échantillons et les données de contrôle en deux vitesses différentes. La boucle de contrôle est généralement plus lente que la fréquence d'échantillonnage, afin d'économiser des cycles de calcul. On utilisera la boucle de contrôle pour faire évoluer les valeurs de certains paramètres en cours d'exécution. Par exemple, un vibrato sera généré au taux de contrôle et viendra modifier la fréquence d'un oscillateur. Ounk permet ces changements de valeur dans le temps. Chaque paramètre de fonction qui accepte les changements continus est accompagné d'un autre paramètre portant le même nom suivi

de «Var». Par exemple, pour modifier en continu la fréquence d'un oscillateur, on associera le canal de sortie d'un générateur de signaux de contrôle, soit son paramètre «bus», avec le paramètre «pitchVar» de la fonction *sine*.

```
randomi(bus = 'pit', mini = .9, maxi = 1.1, rate = 2)
sine(pitch = 250, pitchVar = 'pit')
```

La valeur continue qui évolue sur le canal «pitchVar» sera alors multipliée à la valeur fixe déclarée au paramètre «pitch». Le résultat sera rafraîchi à chaque tour de la boucle de contrôle. Une seule exception à cette règle : si un canal est déclaré au paramètre «panVar», la valeur du paramètre «pan» ne sera pas utilisé. Cette exception a pour but de préserver la cohérence de la panoramisation lorsque le nombre de canaux de sortie est modifié.

Le paramètre «bus» des générateurs de contrôle accepte les listes en entrée, ce qui permet de créer une banque de générateurs en une seule ligne. Par exemple :

```
canaux = range(6) # => [0,1,2,3,4,5]
freqs = [random.uniform(2,5) for i in canaux]
randomi(bus = canaux, mini = .9, maxi = 1.1, rate = freqs)
```

Dans l'exemple ci-dessus, une liste de 6 valeurs est affectée à la variable *canaux*. Ensuite, une deuxième liste est créée avec 6 valeurs pigées aléatoirement entre 2 et 5 et est placée en mémoire dans la variable *freqs*. En affectant ces deux listes aux paramètres «bus» et «rate» de la fonction *randomi*, 6 générateurs aléatoires, aux fréquences différentes, seront créés.

3.3.6 Passage des échantillons audio

Les échantillons audio sortant d'une fonction sont dirigés sur un canal défini par la valeur du paramètre «out». Par défaut, ce paramètre est fixé à la valeur «dac», ce qui signifie que le signal sonore sera acheminé de la sortie de Csound à la carte de son. On peut assigner une valeur de notre choix à ce paramètre (de préférence un chaîne de caractères), et récupérer le signal dans une autre fonction

en assignant la même valeur au paramètre «input». Les fonctions *route* et *toDac* permettent d'organiser des circuits complexes.

```
playSound(sound='ounkmaster.aif', out='snd')
lowpass(input='snd', cutoff=500)
```

3.3.7 Communication à l'aide du protocole *Open Sound Control*

Le protocole de communication *Open Sound Control* est très répandu et supporté par une grande quantité de logiciels et de langages de programmation. Il permet la communication entre différents ordinateurs ainsi qu'entre différents logiciels actifs sur une même machine (Wright and Freed, 1997). Ounk fait usage du protocole OSC, soit pour communiquer avec l'extérieur, soit pour passer des messages à une instance active de Csound par le biais de l'interpréteur Python. Ainsi, il est possible de construire un algorithme évoluant dans l'environnement Python et d'envoyer les événements résultants à un instrument Csound préalablement défini. Ce système offre beaucoup de possibilités pour la composition algorithmique. Nous verrons plus loin l'utilisation du module *pythonInst*, qui permet de créer un instrument Csound répondant à des événements externes, et du module *pattern*, qui est un ensemble de fonctions facilitant la construction de structures algorithmiques.

Csound peut recevoir des signaux de contrôle via le protocole OSC, et ce, peu importe leur provenance. Il suffit de connaître l'adresse IP[4] de la machine réceptrice. En situation de performance, la communication OSC entre différents environnements tels que Csound, Max/MSP, ChucK ou SuperCollider est simple, solide, performante et utile pour les interprètes et les compositeurs.

```
# réception des messages OSC et attribution de canaux de contrôle
oscReceive(bus=['sin','fm','snd'], adress=['/sin','/fm','/snd'],
           port=9000, portamento=.25)
sine(pitch=50, amplitudeVar='sin')
freqMod(pitch=100, modulator=.501, index=4, amplitudeVar='fm')
```

[4]Internet Protocol : protocole utilisé pour la transmission de data via un réseau internet.

```
playSound(sound='ounkmaster.aif', loop=True, amplitudeVar='snd')
startCsound()


def sin(x):
sendOscControl(value=x, adress='/sin', port=9000)

def fm(x):
sendOscControl(value=x, adress='/fm', port=9000)

def snd(x):
sendOscControl(value=x, adress='/snd', port=9000)
```

L'exemple ci-dessus illustre la manipulation de paramètres d'un instrument via des commandes OSC en provenance de Python. Le mixage de trois sources se fait par le biais de l'interpréteur, en appelant des fonctions prédéfinies spécifiant le volume de chaque source.

3.3.8 Rendu en temps réel ou différé

Csound permet le rendu en temps réel, c'est-à-dire que les échantillons calculés sont envoyés directement à la carte de son, ou le rendu en temps différé, où un fichier son est créé sur le disque dur. Le rendu en temps réel est soumis à la puissance du processeur tandis que le rendu en temps différé prend le temps nécessaire pour effectuer ses calculs et créer le fichier son. Dans l'environnement Ounk, il suffit de spécifier un nom, au paramètre «file» de la fonction *startCsound*, pour faire basculer le mode de rendu en temps différé.

Voici une technique de travail largement utilisée dans la création de mes musiques. Au début du script, une variable «RES», pouvant prendre les valeurs «low» ou «high», est définie. De la valeur de cette variable sera déduite la résolution sonore de Csound, c'est-à-dire le taux d'échantillonnage et le taux de contrôle spécifiés avec la fonction *setAudioAttributes*, le mode de rendu et, potentiellement, la qualité de la réverbération utilisée par le script. Donc, lors de la conception de la musique,

la variable est à «low», permettant une écoute rapide du résultat. Lorsqu'une version haute fidélité est désirée, la variable est changée pour «high» et on laisse la machine prendre son temps pour effectuer le calcul. Cette technique permet de construire des musiques sans se soucier des capacités de l'ordinateur, et l'on peut composer sans contrainte. Par exemple :

```
RES = 'low'

if RES == 'low':
    setAudioAttributes(samplingrate=22050, controlrate=2205)
else:
    setAudioAttributes(samplingrate=48000, controlrate=12000)

...

if RES == 'low':
    startCsound()
else:
    startCsound(file='Modulo')
```

3.3.9 Gestion des processeurs multiples

Au moment où ces lignes sont écrites, le sujet de l'heure en informatique musicale est l'utilisation des ordinateurs multi-processeurs pour créer de la musique. L'intégration de multiples processeurs dans les logiciels audio est beaucoup plus complexe qu'on ne le suppose. Les processus audio intimement reliés entre eux, le partage de la mémoire ainsi que le délai temporel introduit par la communication entre les processeurs sont autant de problèmes ne bénéficiant pas encore de solution optimale (Wang, 2008, Wessel, 2008). La communauté Csound a entrepris le développement d'une nouvelle version incorporant le *multi-threading*[5], ce qui permettrait le partage des tâches entre différents processeurs. Présentement, cette version n'est pas encore au point. Une solution intermédiaire a été mise en place dans l'environnement Ounk afin de tirer avantage des capacités qu'offrent

[5]Le partage des calculs d'une tâche sur plusieurs processeurs travaillant en parallèle.

les ordinateurs d'aujourd'hui. Lorsque Csound est lancé plusieurs fois sur une machine, chaque instance du logiciel tente de prendre son propre processeur. Sur un MacPro à huit processeurs, par exemple, il est possible de lancer huit fois Csound et d'utiliser au maximum la puissance de calcul de la machine. Ounk a été conçu de façon à pouvoir démarrer plusieurs scripts en parallèle, utilisant chacun leur propre instance de Csound, donc leur propre processeur. Le temps de démarrage de Csound n'étant jamais exactement le même, il est évidemment difficile d'assurer une parfaite synchronie entre les scripts. Il est alors possible d'envoyer des messages de contrôle, ou même des échantillons audio, d'un script à l'autre par le biais du protocole OSC, ce qui assure un certain contrôle sur la synchronisation des événements.

3.3.10 Gestion des répertoires

Ounk propose des répertoires par défaut où seront directement accessibles certains documents tels les fichiers son, les fichiers MIDI, les réponses impulsionnelles ou les *soundfonts*. Ces dossiers sont au même niveau hiérarchique que l'application Ounk. Tous les documents contenus dans ces dossiers peuvent être appelés sans donner le lien complet sur le disque, Ounk sachant déjà où chercher ce type de fichier. Il est aussi possible de changer les liens vers d'autres dossiers afin de respecter une organisation des fichiers déjà en place sur le système. Il suffit d'indiquer un nouveau lien avec les fonctions *set...Path*.

```
setSoundPath('/Users/Peanut/MyFavoritesSounds/')
setMidiPath('/Users/Peanut/MyFavoritesMIDI/')
```

3.4 Environnements

Ounk offre différents types d'environnements facilitant la construction d'instruments élaborés. Un environnement, dans Ounk, est un morceau de code où les fonctions qui y sont déclarées répondront d'une manière particulière en fonction

de certains types de messages, par exemple, une note MIDI en provenance d'un clavier. Les différents environnements sont expliqués en détail ci-dessous. La description de la tâche d'un environnement commence après une fonction *beginXXX* et se termine avec la fonction *endXXX* correspondante.

3.4.1 Instrument MIDI

Toutes les commandes se trouvant entre les fonctions *beginMidiSynth* et *endMidiSynth* seront en attente d'un événement MIDI pour entrer en action. La fréquence des fonctions ayant un paramètre «pitch» sera alors transposée par le rapport de la note MIDI jouée et de la valeur de référence donnée au paramètre «centralkey» de la fonction *beginMidiSynth*. L'amplitude ainsi que la durée seront aussi déterminées par les événements MIDI reçus. Ces actions sont transparentes et entièrement prises en charge par Ounk, ce qui permet de passer rapidement du mode normal à un instrument MIDI, ou vice versa, simplement en plaçant en commentaire les fonctions créant l'instrument MIDI. La fonction *beginMidiSynth* permet aussi de spécifier le temps de relâche du son, en seconde, après la réception du «noteoff»[6], ainsi que la valeur de transposition, en demi-ton, appliquée au contrôleur «pitch bend». Ce dernier ainsi que le contrôleur numéro 7, associé au volume, sont toujours actifs dans un instrument MIDI. Il est aussi possible de créer plus d'un instrument MIDI et de leur assigner des canaux différents.

```
beginMidiSynth(channel=1, centralkey=45, release=1, pitchbend=2)
freqMod(pitch=220, modulator=.501, index=4, out='fm')
lowpass(input='fm', cutoff=4000)
endMidiSynth()
```

Le script ci-dessus construit un instrument MIDI constitué d'une modulation de fréquence suivie d'un filtre passe-bas. Dû au fait que la fréquence donnée au paramètre «pitch» de la fonction *freqMod* équivaut à la note MIDI donnée au

[6]Relâche de la note ou une note avec une vélocité de 0.

paramètre «centralkey» de la fonction *beginMidiSynth*, la hauteur entendue correspondra à la hauteur MIDI demandée.

Les assignations audio à l'intérieur d'un instrument MIDI, spécifiées par les paramètres «out», ne sont valables que pour cet instrument. La fonction *endMidiSynth* a son propre paramètre «out» pour assigner le son de l'instrument à d'autres fonctions de Ounk, afin d'appliquer des traitements sur le son global de l'instrument.

La fonction *midiSynthCtl* permet de récupérer n'importe quel contrôleur MIDI externe à l'intérieur d'un instrument MIDI, tandis que la fonction *midiSynthGetBus* permet de récupérer des données de contrôle provenant d'une fonction Ounk préalablement définie.

```
randomi(bus='mod', mini=.99, maxi=1.01, rate=8)
beginMidiSynth()
midiSynthCtl(bus='ind', ctlnumber=74, minscale=0, maxscale=10)
midiSynthGetBus(bus='modu', input='mod')
freqMod(pitch=220, modulatorVar='modu', indexVar='ind')
endMidiSynth()
```

Ce script utilise la fonction *midiSynthCtl* pour récupérer le contrôleur MIDI numéro 74 à l'intérieur d'un instrument MIDI et l'assigner à l'index de modulation d'une synthèse par modulation de fréquence. La fonction *midiSynthGetBus* permet d'utiliser le canal de contrôle «mod», définit à l'extérieur de l'instrument avec la fonction *randomi*, et de l'appliquer au multiplicateur de la fréquence de l'oscillateur modulant.

Deux dernières fonctions complètent cette librairie et offrent la possibilité de créer rapidement des instruments MIDI originaux, *splitKeyboard* et *splitVelocity*. La première permet de définir des régions sur l'étendue du registre de hauteurs MIDI, tandis que la seconde sépare le registre d'amplitude en plusieurs couches. Des processus différents peuvent alors être assignés en fonction de la hauteur et de la vélocité des notes jouées. Voici un script distribuant sur l'étendue du clavier

les échantillons du piano préparé de John Cage, avec trois niveaux de vélocité. Les échantillons sonores sont sauvegardés dans un même dossier et sont nommés en suivant la logique suivante :

numéro de la touche + trait souligné + indication de vélocité + extension

```
setSoundPath('/Users/olipet/sampler')
beginMidiSynth(release=.05)
# boucle sur les 88 premières notes du clavier
for i in range(88):
    # chaque région comprend une seule touche
    splitKeyboard(firstnote=i, endnote=i, centralkey=i)
    # boucle sur les 3 vélocités
    for j in range(3):
        # p = (0 -> .3), mf = (.3 -> .6), ff = (.6 -> 1)
        splitVelocity(minthresh=[.6,.3,0][j], maxthresh=[1,.6,.3][j])
        # nom du fichier son (numéro_vélocité.aif)
        s = str(i) + '_' + ['ff','mf','p'][j] + '.aif'
        playSound(sound=s)
endMidiSynth()
```

3.4.2 *Step sequencer*

Le *Step sequencer* permet de construire des séquences qui seront jouées en boucle, à une vitesse déterminée par une fonction *metro*. Les commandes à exécuter en séquence doivent être placés entre les fonctions *beginSequencer* et *endSequencer*. La fonction *beginSequencer* contient quelques paramètres essentiels au bon fonctionnement de la séquence. D'abord, le paramètre «input» doit être associé au paramètre «bus» d'un métronome. Chaque clic du métronome fera avancer d'un pas la table d'amplitudes donnée au paramètre «table». La ou les tables d'amplitudes seront généralement créées par les fonctions *genDataTable* ou *genRhythmTable*, et serviront à déterminer l'amplitude des événements pour chaque pas de la séquence. Le paramètre «mode» de la fonction *beginSequencer* est effectif seulement lorsqu'une liste de tables a été spécifiée. Si la valeur est de 0, les tables seront lues de façon séquentielle. Si la valeur est supérieure à 0, elle correspond au pourcentage de

chance qu'une nouvelle table soit choisie au début de chaque séquence. Les tables sont alors choisies de façon aléatoire. Voici un exemple de séquenceur :

```
env = genAdsr(release=0.8)
amp = genDataTable([1,0,.5,0,1,0,.5,0,1,0,.5,0,1,0,.5,.5])
metro(bus='metro', tempo=132)
beginSequencer(input='metro', table=amp)
freqMod(pitch=250, modulator=.502, duration=.5, envelope=env)
endSequencer()
```

Dans ce script, une table d'amplitudes, pour une mesure à 16 pas, est créée avec la fonction *genDataTable* et est mise en mémoire dans la variable «amp». Le métronome fera avancer la séquence d'un pas dans la table à chaque nouveau clic. Une note sera jouée et la valeur pigée dans la table servira d'amplitude pour toutes les fonctions intégrées à la séquence.

La fonction *sequencerPitchTable* permet de spécifier une table de multiplicateurs de fréquence qui seront appliqués à tous les paramètres «pitch» présents dans la séquence. De façon plus générale, la fonction *seqParameterTable* permet de remplacer la valeur de n'importe quel paramètre d'une fonction faisant partie de la séquence. Le pas de la séquence servira à spécifier quelle valeur dans la table doit être utilisée pour remplacer la valeur initiale du paramètre.

```
ind = genDataTable([8,7,6,5,4,3,2,1,8,7,6,5,4,3,2,1])
pit = genDataTable([1,0,.5,0,1,0,.5,0,1,0,.5,0,1,0,.5,.5])
metro(bus='metro', tempo=132)
beginSequencer(input='metro', table=tt1)
# table de multiplicateurs, indexée par le pas, appliqués aux paramètres 'pitch'
sequencerPitchTable(pit)
beginSequencer(input='metro', table=tt1)
# table de valeurs, indexée par le pas, appliquées aux paramètres 'index'
seqParameterTable('index', ind)
freqMod(pitch=250, modulator=.502, duration=.5, index=4)
endSequencer()
```

La fonction *beginSequencer* contient des paramètres servant à démarrer ou arrêter le séquenceur, permettant le mixage en temps réel de plusieurs séquences. Le paramètre «active» détermine l'état initial de la séquence, c'est-à-dire, si elle démarre

dès le début de l'exécution du script ou si elle attend un événement déclencheur. Si la valeur donnée au paramètre «trigval» est envoyée sur le canal de contrôle spécifié au paramètre «trigbus», la séquence bascule d'un état à l'autre.

3.4.3 Instrument Python

L'environnement permettant de créer des «instruments Python» est une des caractéristiques les plus originales de Ounk. C'est par ce type d'instrument que l'on pourra réellement utiliser Python à ses pleines capacités algorithmiques. L'instrument Python est un groupe de fonctions, délimitées par les fonctions *beginPythonInst* et *endPythonInst*, qui seront en attente d'un appel de Python pour lancer leur processus. Cela permettra de construire des algorithmes entièrement dans l'environnement Python et d'envoyer des appels de notes à Csound durant l'exécution. La fonction *beginPythonInst* ne prend qu'un seul argument, qui est le numéro de l'instrument, une valeur arbitraire permettant de créer plusieurs instruments Python indépendants dans un même script. Voici un exemple très simple d'instrument Python en attente d'un événement pour jouer une note de synthèse par modulation de fréquence :

```
beginPythonInst(voice=1)
freqMod(pitch=250, modulator=.501, index=4, duration=1, out='fm')
endPythonInst()

lowpass(input='fm', cutoff=2000)

proc = startCsound()
```

À noter la valeur de retour de la fonction *startCsound*, jusqu'ici inutilisée. Cette valeur, affectée à la variable «proc», contient le numéro du processus Ounk du script. Comme plusieurs scripts Ounk peuvent être actifs en même temps, plusieurs instances de Csound peuvent être présentes sur le système. Le numéro de processus Ounk servira à identifier l'instance de Csound à laquelle s'adressent les appels d'événements. Chaque instrument de chaque script en cours est donc indépendant.

On utilisera la fonction *sendEvent*, expliquée ci-après, pour effectuer les appels d'événements. Cette fonction prend comme paramètre le numéro de l'instrument, le numéro de processus ainsi qu'un dictionnaire[7] où pourront être spécifiées, à chaque nouvelle note, de nouvelles valeurs pour n'importe quels paramètres des fonctions de l'instrument. Voici un appel à l'instrument ci-dessus où sont spécifiées les valeurs de fréquence et d'index de la fonction *freqMod* :

```
# création d'un dictionnaire avec comme clé le nom de la fonction visée
dict = {'freqMod': {}}
# affectation de valeurs
dict['freqMod']['pitch'] = 200
dict['freqMod']['index'] = 8
sendEvent(voice=1, dict=dict, process=proc)
```

Utilisée avec les fonctions de la catégorie *Patterns*, cette technique permet de construire des algorithmes complexes et flexibles. La fonction *pattern* crée un objet qui sert d'horloge dans l'environnement Python et qui appelle, à intervalles réguliers ou selon une rythmique particulière, une fonction où sera généralement construit le dictionnaire et effectué l'appel de notes.

```
def note()
    dict = {'freqMod': {}}
    dict['freqMod']['pitch'] = random.randint(100,200)
    sendEvent(voice=1, dict=dict, process=proc)

pat = pattern(time=.125, function=note, pattern=[1,1,2])
pat.start()
pat.play()
```

Ce code permet d'envoyer des événements à l'instrument définit précédemment. La fonction *pattern* crée une horloge qui appellera périodiquement, selon une vitesse et un schéma rythmique, la fonction *note*. À chaque fois que la fonction *note*

[7]Dans l'environnement Python, le dictionnaire est une table de données, définie entre accolades {}, sur base de paires clé - valeur. Les paires sont séparées par des virgules et la clé est séparée de sa valeur par le symbole deux-points ' :'.

est appelée, un dictionnaire est créé avec une nouvelle valeur de hauteur et un événement est demandé à l'instrument Python.

La catégorie *Patterns* contient un certain nombre de fonctions permettant de modifier le comportement de l'algorithme en cours d'exécution. Des fonctions témoins telles que *getBeat*, *getBar* et *getTime* permettent de suivre l'évolution temporelle du jeu, et de modifier la structure de l'algorithme en conséquence. Il est possible de modifier en tout temps la vitesse de l'horloge ou la séquence rythmique utilisée avec les fonctions *changeTime* et *changePattern*. Ce module contient aussi certaines fonctions spécialisées dans la génération de rythmes originaux et cohérents.

3.4.4 Boucleur

L'environnement *Boucleur* permet de mettre en boucle des procédés sans avoir à répéter plusieurs fois des groupes de commandes. En plaçant les éléments[8] à boucler entre les fonctions *beginLoop* et *endLoop*, ils seront automatiquement mis en boucle pour toute la durée spécifiée à la fonction *beginLoop*. La durée de la boucle de chaque fonction est dépendante de la durée spécifiée à son paramètre «duration». Ceci permet de créer, simplement, des effets rythmiques en bouclant des fonctions ayant différentes durées. Dans l'exemple ci-dessous, trois boucles de longueurs différentes seront créées.

```
env = genAdsr()
beginLoop(starttime=0, duration=30)
sine(pitch=[200,300,400], duration=[.25,.33,.5], envelope=env)
endLoop()
```

Deux autres paramètres à la fonction *beginLoop* permettent de modifier le comportement de la boucle. La valeur du paramètre «amplitude» agira sur l'amplitude globale de l'environnement, c'est-à-dire qu'elle affectera toutes les fonctions

[8]Toutes les fonctions possédant les paramètres «starttime» et «duration» peuvent être mises en boucle.

présentes. De façon similaire, la valeur du paramètre «legato» modifiera la durée entendue des événements, sans changer la durée utilisée pour calculer le temps de départ des événements. Si, au lieu d'une valeur seule, une liste de deux valeurs est donnée à ces paramètres, une ligne droite, en partant de la première valeur jusqu'à la deuxième, sera calculée sur toute la durée de la boucle. Ainsi, la valeur affectant soit l'amplitude, soit la durée des processus subira un changement progressif sur toute la durée de la boucle. Dans le prochain exemple, la boucle commence legato puis se transforme progressivement en staccato tout en disparaissant graduellement.

```
env = genAdsr()
beginLoop(duration=30, legato=[1,.5], amplitude=[1,0])
sine(pitch=[300,400], duration=[.25,.5], envelope=env)
endLoop()
```

3.4.5 Instrument «Csound»

Il est pratiquement impossible de produire des fonctions pour toutes les possibilités de chaînes de traitement qu'offre Csound. Une solution devait être envisagée qui ne limite pas la programmation aux seules fonctions offertes par Ounk. L'instrument «Csound» à été créé afin de permettre à l'utilisateur d'écrire ses propres instruments Csound et de les intégrer facilement à Ounk. Un instrument existant, sous la forme d'un fichier texte, peut être appelé avec la fonction *myInstrument*, et sera dès lors utilisé comme tout autre instrument produit par Ounk.

Une syntaxe spéciale permet de relier les paramètres d'initialisation des notes ainsi que les canaux de contrôle de l'instrument Csound aux variables et aux «bus» de Ounk. Le paramètre «argslist» de la fonction *myInstrument* attend une liste de valeurs qui seront assignées, dans l'instrument Csound, aux paramètres d'initialisation **p4** et les suivants. Les paramètres d'initialisation **p1**, **p2** et **p3**, qui représentent respectivement le numéro de l'instrument Csound, le temps de départ et la durée de la note, sont pris en charge par Ounk, notamment par la biais des paramètres «starttime» et «duration». Pour relier des canaux de contrôle, il

faut passer un dictionnaire au paramètre «vardict» de la fonction *myInstrument*. Dans ce dictionnaire, les clefs correspondront aux noms des variables à contrôler telles qu'elles apparaissent dans l'instrument Csound (sans le [k])[9], tandis que la valeur associée à cette clef sera le nom donné au paramètre «bus» d'une fonction de contrôle préalablement définie dans le script. Voici un exemple d'instrument Csound, soit un simple oscillateur avec une enveloppe et une variation de fréquence :

```
instr 1
kenv linseg 0,.1,1,p3-.1,0
asig oscili p4*kenv, p5*kpit, p6
out asig
endin
```

Et voici comment cet instrument peut être intégré à Ounk :

```
wave = genWaveform()
# variations de fréquence
randomi(bus='pitch', mini=.95, maxi=1.05, rate=5)
fichier = 'path_vers_mon_fichier.txt'
# p4 = 5000, p5 = 200, p6 = wave
myInstrument(fichier, duration=1, argslist=[5000,200,wave],
             vardict={'pit': 'pitch'}, out='toRev')
```

Dans cet exemple, la variable «kpit» de l'instrument Csound est asservie au canal de contrôle «pitch» et les paramètres d'intialisation **p4**, **p5** et **p6** sont passés via le paramètre «argslist». Comme pour les fonctions standards, on assigne la sortie audio par le biais du paramètre «out».

Cette technique devient particulièrement utile lorsqu'utilisée à l'intérieur d'un instrument Python. Par exemple, dans le script ci-dessous, un instrument Csound est définit à l'intérieur d'un instrument Python et la fonction *note* servira à effectuer les appels d'événements en modifiant les valeurs **p3** et **p5** à chaque nouvel événement :

[9]En Csound, une variable de contrôle débute obligatoirement par la lettre [k]. Dans l'exemple suivant, la variable «kpit», qui multiplie la fréquence d'un oscillateur, sera référencée dans le dictionnaire par la clé «pit».

```
beginPythonInst(voice=1)
myfile = os.getcwd() + '/Resources/examples/myOrch.txt'
myInstrument(myfile, duration=1, argslist=[...], vardict={...})
endPythonInst()

def note():
    dict = {'myInstrument': {}}
    dict['myInstrument']['p3'] = random.randint(2,8)
    dict['myInstrument']['p5'] = random.randint(50,200)
    sendEvent(voice=1, dict=dict, process=proc)
```

3.4.6 Interface graphique

Un groupe de fonctions servent à construire une interface graphique, permettant la manipulation de certains paramètres du script. La fonction *beginGUI* commence par construire un cadre, c'est à dire une fenêtre où l'on pourra placer des objets d'interface. Des fonctions tels *makeSlider*, *makeButton* et autres seront utilisées pour construire l'interface. En plus des paramètres de configuration, ces fonctions prennent un autre paramètre qui spécifiera quelle fonction sera appelée chaque fois que l'objet sera manipulé. Ce dernier point est essentiel, car c'est la fonction donnée en argument qui recevra, par exemple, la valeur du potentiomètre ou le signal du bouton. De cette fonction, les valeurs peuvent être utilisées directement dans Python, ou passées à Csound, via le protocole OSC.

```
def onPlay():
    sendOscTrigger(value=1, adress='/play', port=8000)

def handleSlider(val):
    sendOscControl(value=val, adress='/s1', port=8000)

def handleSlider2(val):
    sendOscControl(value=val, adress='/s2', port=8000)

frame = beginGUI(size=(260,300))
p1 = makeButton(frame, label='Play', pos=(90,20), function=onPlay)
s1(frame, mini=-24, maxi=24, pos=(50,50), function=handleSlider)
s2(frame, mini=-24, maxi=24, pos=(50,120), function=handleSlider2)
endGUI(frame)
```

Ce script construit une interface graphique contenant trois objets, un bouton et deux potentiomètres. Chaque fois qu'un objet change d'état, il appelle sa fonction dédiée et lui passe, s'il y a lieu, sa valeur en paramètre. Cette valeur est ensuite acheminée vers Csound par le biais des fonctions *sendOscControl* et *sendOscTrigger*.

CHAPITRE 4

LE CYCLE DES VOIX : TRAVAUX ET MÉTHODES

4.1 Contexte

Ce cycle musical a été entièrement composé à l'aide d'outils développés au cours des cinq dernières années. Les modèles de synthèse de la voix chantée et du didjeridu ont été utilisés pour générer la quasi-totalité des matériaux sonores utilisés, et la composition des pièces a été réalisée dans le logiciel de programmation musicale Ounk, présenté au chapitre précédent. Chaque composition, à l'exception d'une seule, existe sous la forme d'un script Python, générant un processus musical exécuté par Csound, qui peut être joué en temps réel ou en temps différé. Quelques scripts sont donnés en annexes. Ce cycle a été conçu pour pouvoir être diffusé aussi bien en version stéréophonique qu'en version multiphonique.

Le choix de n'utiliser que les modèles de synthèse pour la composition de tout le cycle fut motivé par trois raisons. Premièrement, l'utilisation limitée de matériaux a forcé l'exploration en profondeur des possibilités qu'ils offraient sur le plan du timbre et de la qualité sonore. Deuxièmement, ce choix m'a permis de me concentrer entièrement sur l'écriture de la musique, en fonction d'une instrumentation fixe, donc avec une pleine connaissance des textures et des modes de jeu disponibles. Enfin, la nature des instruments proposait une certaine cohérence timbrale de l'oeuvre.

4.2 Musique

Le *Cycle des voix* est une suite musicale constituée de sept compositions consistant chacune en un seul geste musical. *Choeurs*, la dernière pièce du cycle, est une synthèse des gestes musicaux explorés explorés durant la composition du cycle et regroupe presque tous les matériaux sonores utilisés dans les pièces précédentes. Bien que les pièces aient été composées pour être diffusées dans un ordre précis en fonction de l'évolution dramatique de l'oeuvre, chaque composition peut exister indépendamment du cycle. Les sections suivantes expliquent la structure des pièces constituant le *Cycle des voix*.

Drone

Drone est la pièce d'ouverture du cycle. Utilisant les deux modèles de synthèse décrits dans le présent ouvrage (synthèse de la voix et du didjeridu), l'élément principal de cette pièce est composé pour un didjeridu solo dont la source d'excitation est une voix de synthèse. Explorant un univers très sobre et introspectif, *Drone* ouvre le cycle avec une longue trame exploitant le côté organique des sons graves générés par le didjeridu. La pièce évolue très lentement, s'attardant sur les différentes textures résonantes générées par le chant dans le didjeridu.

La chaîne de traitements dans ce processus est complexe et riche en possibilités. La synthèse vocale est d'abord générée sur différentes hauteurs et différents timbres, puis elle est modulée par l'action des lèvres à l'entrée du didjeridu. Le résultat de ce traitement est ensuite transformé par les qualités spectrales du didjeridu, composé de 45 modes de résonances. La fréquence centrale de chaque mode est en constante évolution, simulant un didjeridu dont la longueur et la forme changent au cours du temps. Une distorsion est également ajoutée à certains événements de la voix de synthèse, ainsi qu'une modulation en anneau, permettant de simuler des cris dans l'instrument. Ces événements servent d'une part à modifier la richesse du spectre de façon marquée et, d'autre part, à moduler la dynamique générale de la pièce en

créant de grandes vagues entre les sections douces et les sections fortes.

La pièce est jouée en temps réel dans l'environnement Ounk car le jeu de la synthèse vocale est produit par des algorithmes écrits en Python et transmis à Csound sous la forme d'événements indépendants. En parallèle, des changements continus sont appliqués aux paramètres du didjeridu. Le déplacement du formant principal de la bouche crée une résonance qui balaie le spectre, la modification de la longueur du tuyau déplace les résonances de l'instrument et une variation de la pression des lèvres modifie la richesse harmonique du timbre.

Paysages

Paysages explore le paradoxe entre les sons de la nature et la synthèse sonore. Dans cette pièce, des sons à connotation naturelle très forte, tels des grillons et des rafales de vent, ont été simulés avec le modèle de synthèse de la voix. Ces textures évoquant la nature cohabitent avec des sons purement synthétiques, tels que la note de basse constituée d'une forme d'onde simple. Ce paysage sonore sert de toile de fond à une exploration libre des possibilités timbrales offertes par le modèle de synthèse de la voix. Pour la génération des voix principales de cette pièce, les paramètres de contrôle sont explorés dans les registres extrêmes afin de générer des textures sonores complètement différentes de la voix chantée.

Une version particulière du modèle de synthèse vocale, offrant la liberté de modifier en temps réel certains paramètres de contrôle, a été créée spécialement pour générer les voix solos de cette pièce. Par exemple, dans le modèle de synthèse de la voix généralement utilisé, les fréquences centrales des formants sont générées automatiquement en fonction de la voyelle à synthétiser. Dans cette version du modèle, une variation aléatoire appliquée à chacun des formants permet de déplacer les filtres selon des trajectoires complètement différentes de celles présentes dans l'articulation de phonèmes. À certains moments de la pièce, la vitesse et l'amplitude du vibrato sont modifiées de façon à obtenir des sons se situant entre la synthèse

vocale et la modulation de fréquence.

Ce modèle est incorporé dans l'environnement Ounk avec la fonction *myInstrument*, permettant ainsi d'interpréter directement du code écrit en Csound (voir chapitre 3.4.5). Les trajectoires de contrôle sont générées en Python et sont ensuite dirigées vers les paramètres de l'instrument Csound via le protocole *Open Sound Control*.

Déphasages

Première de deux pièces influencées par la musique de Steve Reich, la pièce *Déphasages* explore les effets des déphasages rythmiques dans un contexte polyphonique. Sept voix de polyphonie sont introduites à tour de rôle tout au long de la pièce, augmentant progressivement la densité de l'harmonie. Chacune des voix chante de courts motifs musicaux, en boucle, sur un tempo commun. Les rythmes de la pièces sont constitués de cellules ternaires de 2, 4 ou 8 temps, avec la croche comme plus petite valeur temporelle. De façon périodique, certaines voix décalent leur temps d'entrée d'une double-croche par rapport à la pulsation de base. Cela a pour effet de créer une sensation de tempo deux fois plus rapide et de générer des roulements de notes lorsque deux voix décalées l'une par rapport à l'autre jouent dans un même registre. Après une certaine durée, le décalage temporel est annulé et la rythmique de base est rétablie. Les voix recommencent à chanter en phase les unes avec les autres, créant une sensation de repos après un moment de grande activité rythmique. Pour marquer le moment où les voix se synchronisent à nouveau, les cellules rythmiques et l'harmonie changent, et une voix est ajoutée au groupe.

Une mélodie ajoutée au tiers de la pièce et chantée par des voix de basse vient soutenir le tempo en marquant les temps. Cette voix n'est jamais décalée dans le temps, assurant une présence sur les temps forts de la mesure afin de garder une pulsation bien sentie. Cette ligne mélodique ajoute de la couleur à l'harmonie des voix en imposant une note fondamentale très présente.

Cette pièce exploite un élément particulier du logiciel Ounk : l'environnement *step sequencer*. Cet environnement permet de placer n'importe quel processus ou chaîne de traitements dans un instrument répondant à la pulsation d'un métronome. Une ou plusieurs tables de rythmes peuvent être définies pour chaque séquenceur et tous les paramètres des fonctions, à l'intérieur de la séquence, peuvent aussi être modifiés par la lecture cyclique de tables désignées. L'environnement *step sequencer* permet de construire des cellules musicales hautement évolutives, respectant une pulsation très précise, tout en exécutant un processus sonore arbitraire.

Les rythmes et les mélodies sont générés par des algorithmes élaborés avec le langage Python dans le cadre du développement de la suite de logiciels *TamTam*[1], et adaptés à l'environnement Ounk.

Modulo

La pièce *Modulo* est construite sur une harmonie constante de douze notes, chantées par des voix de synthèse. Afin d'éliminer le caractère vocal des sons, chaque voix est modulée par un oscillateur à basse fréquence, faisant graduellement apparaître et disparaître la voix. Comme chaque oscillateur modulant possède une fréquence différente, l'harmonie entendue est en mouvement constant selon les croisements provoqués par les apparitions et les disparitions des différentes voix. Cette toile de fond donne un caractère très aérien à la pièce. La dynamique de mouvement est construite à partir de modules de distorsion, associés chacun à une voix, et possédant chacun leur propre oscillateur à basse fréquence affectant leur amplitude respective. Les douze voix et distorsions, réparties de façon fixe dans l'espace, créent des effets de déplacement par l'apparition et la disparition des éléments. Afin d'accentuer ce phénomène, les distorsions, dont les emplacements dans l'espace sont associés à chacune des voix, reçoivent en entrée audio le total des douze

[1]La suite de logiciels *TamTam* a été développée par une équipe de l'Université de Montréal, sous la direction de Jean Piché, dans le cadre du projet *One Laptop Per Child*.
http ://tamtam4olpc.wordpress.com/

voix de synthèse. Ceci crée l'effet d'une voix entendue en un point de l'espace alors que cette même voix, traitée, apparaît en un autre point de l'espace, créant des jeux de mouvement intéressants.

Chorus morph

La pièce *Chorus morph* explore le fondu progressif d'un timbre sonore vers un autre. La musique commence avec une de très nombreuses voix, des sopranos chantant des notes extrêmement aiguës et des basses chantant à des fréquences fondamentales inférieures à 20 Hz, créant une masse sonore très dense, difficile à associer à la voix chantée. Au fur et à mesure que la pièce évolue, les sons s'identifient de plus en plus à la voix chantée, alors que de longs glissandos appliqués aux fréquences fondamentales font converger les notes vers une fréquence médiane où toutes les voix se rejoignent à la fin, créant un chorus sur une seule note. *Chorus morph* est donc une masse sonore en constante transformation, partant d'un spectre complexe vers un spectre simple.

Dans cette pièce, seuls les registres sont importants, les notes chantées doivent être le plus éloignées possible d'une quelconque tonalité ou harmonie. Les fondamentales sont obtenues à l'aide de générateurs de nombres aléatoires contrôlés de type *Weibull*, illustré par l'équation 4.1, dont la distribution est fonction de deux paramètres, λ, le centre de gravité et K, la forme de la distribution.

$$x = \lambda \left(\log \left(\frac{1}{1 - random(0,1)} \right) \right)^{(1/K)} \tag{4.1}$$

Les figures 4.1 et 4.2 illustrent les densités de distribution aux quatre endroits pivots de la pièce.

Comme cette pièce nécessite une grande quantité de voix pour créer la masse sonore, plusieurs processeurs sont requis pour générer la musique en temps réel. Comme les algorithmes sont gérés dans l'environnement Python, qui envoie en

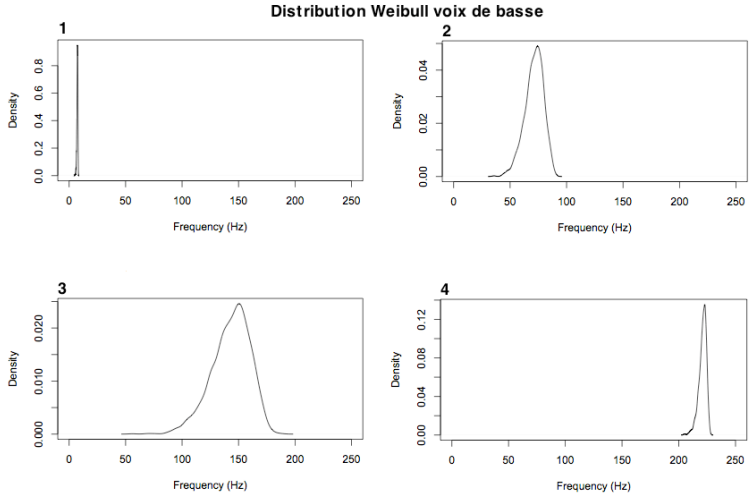


FIG. 4.1 – Densités des distributions *Weibull* pour les voix de basse.

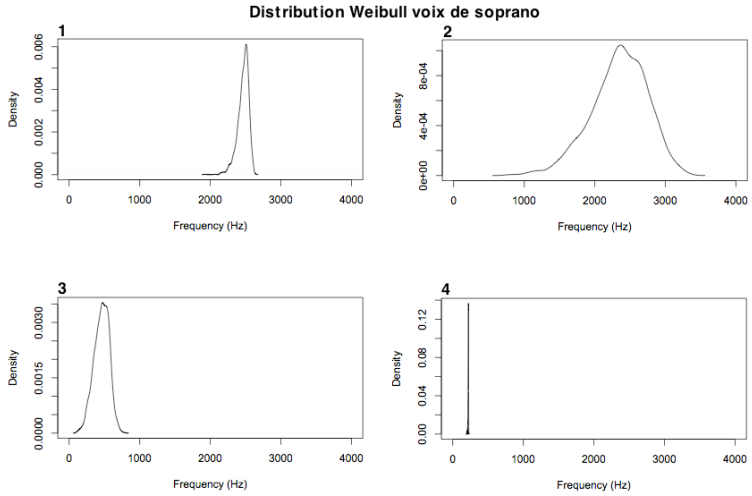


FIG. 4.2 – Densités des distributions *Weibull* pour les voix de soprano.

temps réel des événements à jouer aux instruments Csound, le rendu en temps différé ne peut être utilisé pour cette pièce. Les capacités multi-processeurs de Ounk sont utilisées pour lancer, au même moment, plusieurs scripts semblables mais possédant leurs propres générateurs de nombres aléatoires propres, générant des masses légèrement différentes. Chaque script appelle une nouvelle instance de Csound, qui prendra le premier processeur disponible. Les flux sonores de chacune des instances sont enregistrées dans des fichiers séparés et sont ensuite mixées en un seul fichier son.

Reich

Cette deuxième pièce influencée par l'oeuvre de Steve Reich est construite en deux parties. La première partie, plus lente, installe un style musical constitué de courtes cellules binaires chantées en boucle dans une harmonie à sept voix. Sept voix sont utilisées afin d'obtenir une octophonie où chaque voix n'est pas exactement située dans un haut-parleur, mais plutôt dans un espace situé entre deux haut-parleurs. Une table de probabilité, qui gère la longueur des cellules rythmiques, appelle une longueur métrique différente à chaque nouvelle mesure. Des cellules de quatre notes ont la plus forte probabilité d'être pigées, tandis que des cellules de huit ou seize notes viendront parfois étendre les lignes mélodiques, contrebalançant l'effet de répétition. La deuxième section de la pièce applique le même principe mais sur un tempo beaucoup plus rapide avec des changements d'harmonie plus fréquents, créant une section plus dynamique. Une mélodie très présente à la basse vient accompagner les mélodies aux voix supérieures.

Voici comment l'environnement *Step sequencer* de Ounk a été utilisé pour créer les mélodies de la pièce *Reich* :

```
s1 = setRhythm(1, TAP, 60,85,70)

metro(bus='metro', tempo=132, tap=TAP, tempoVar='tempVar',
      tapVar='tapVar', starttime=SEQSTART, duration=SEQDUR)
```

```
beginSequencer(input='metro', table=s1)
seqParameterTable(['pitch', 'duration', 'fadeout'],
                  [sop1, notedur, fadedur])
setVoice(vowel=0, amplitude=.4, voicetype=3, vibratoamp=0, out='v1')
endSequencer(starttime=SEQSTART, duration=SEQDUR)
```

Deux fonctions (*setRhythm* et *setVoice*) sont définies au début du script et permettent respectivement de générer des cellules rythmiques et des timbres de voix. La fonction *seqParameterTable* permet d'utiliser des tables préalablement créées à l'aide d'algorithmes de génération aléatoire, pour modifier certains paramètres de la synthèse vocale à chaque temps de la mesure. Les tables, qui évoluent dans le temps en exécutant un fondu sur plusieurs listes de valeurs, viennent modifier respectivement la hauteur, la durée ainsi que le temps de chute de chaque note. Ce système permet de créer des boucles introduisant des changements de comportement sur toute l'étendue de la pièce.

Choeurs

Cette pièce incorpore presque tous les types de timbres possibles avec le modèle de synthèse vocale. *Choeurs* a été composée en parallèle à la création du logiciel Ounk. C'est la seule pièce du cycle qui a été mixée de façon traditionnelle dans un séquenceur multipistes. Les matériaux utilisés proviennent principalement de générateurs algorithmiques, que ce soit pour les motifs rythmiques, pour les motifs mélodiques ou pour les contrôles des paramètres de sons vocalement impossibles. Bien que la première version de cette pièce ait été composée avec Max/MSP, une grande partie des matériaux ont été refaits avec Ounk.

Choeurs est construite en trois parties. La première section présente les matériaux développés au cours de la pièce : des accords denses, des sons de vent créés avec le modèle de synthèse de la voix et des événements très brefs dans le registre aigu. La section centrale installe une trame rythmée comme fond sonore pour un choeur de sopranos et fait le lien avec la section finale. Celle-ci est constituée d'un choeur d'environ quarante voix, créant un fond harmonique dense, au dessus duquel se

démarque une voix de soprano solo. Cette section utilise plusieurs matériaux déjà utilisés aux deux premières sections de la pièce afin de faire de brefs rappels. Pour mener au point culminant du cycle, la finale propose une atmosphère très planante composée d'une harmonie riche et complexe.

Choeurs profite de l'élimination de la contrainte de la génération en temps réel pour exploiter des matériaux d'une grande densité. Par exemple, la section rythmique a été générée dix fois, chaque génération incluant plusieurs voix aux timbres et aux comportements différents, et mixée afin de produire une pulsation en constante évolution.

Le but de cette pièce était d'utiliser au maximum les qualités expressives du modèle de synthèse de la voix chantée sans aucune contrainte de programmation. Les seuls effets audio-numériques utilisés dans cette composition consiste en une réverbération par convolution ainsi que quelques filtres d'égalisation.

CONCLUSION

Dans le domaine de la synthèse sonore, la voix chantée constitue probablement le dernier défi qui n'a pas encore trouvé de solution élégante. Le modèle développé au cours de ce projet vise à offrir aux compositeurs une synthèse de bonne qualité sonore, simple à manipuler et peu gourmande en ressources, que ce soit en puissance de calcul, en espace disque ou en mémoire vive. Une attention particulière a été portée aux micro-modulations des paramètres, éléments essentiels pour conférer un caractère le plus naturel possible aux sons de synthèse. Aussi, ce modèle s'avère efficace pour produire des phonèmes incorporant des consonnes, particulièrement les consonnes plosives. Le modèle génère une synthèse vocale naturelle, riche et versatile. Il a été spécifiquement développé à l'intention des compositeurs.

Le modèle de synthèse du didjeridu décrit dans cette thèse, bien que relativement simple dans son implémentation, constitue un module de traitement aux sonorités intéressantes. Le didjeridu, par sa forme conique, génère des modes de résonance dans un rapport non-harmonique, créant ainsi une structure modale riche et complexe. Les paramètres de contrôle, simples à manipuler, en font un instrument particulièrement utile pour la génération ou le traitement de signaux audio. Utilisé de pair avec le modèle de synthèse vocale, il permet de simuler un réel joueur de didjeridu ainsi que les différents modes de jeu de l'instrument.

Afin de fournir un environnement de contrôle agréable et versatile aux modèles décrit ci-dessus, un logiciel de programmation musicale, *Ounk*, a été développé. Ce logiciel allie la puissance du langage de programmation Python à la qualité du moteur audio Csound. Il permet l'écriture de scripts effectuant des traitements sur des signaux audio ou la composition de pièces musicales complètes, de nature générative ou non. Il offre le choix entre le rendu en temps réel ou le rendu en temps différé, ce qui permet de composer à basse résolution, sans délai d'attente à l'écoute, et de générer une version haute définition de la pièce lorsque celle-ci est au point. Les

modèles de synthèse de la voix et du didjeridu sont complètement intégrés à Ounk, notamment par l'ajout de valeurs par défauts à tous les paramètres. Ceci simplifie le contrôle des instruments, puisque l'utilisateur n'a qu'à spécifier les variables qu'il désire contrôler.

Le *Cycle des voix* est une oeuvre pour synthèse vocale et didjeridu entièrement composée dans le logiciel *Ounk*. La contrainte de n'utiliser que ces sources de matériaux pour la construction de la musique avait pour but de forcer une exploration exhaustive des multiples textures sonores possibles avec ces modèles de synthèse. La provenance restreinte des sources confère aussi une certaine homogénéité à l'ensemble des pièces constituant le *Cycle des voix*.

Cette recherche ouvre la voie à la composition générative à l'aide du langage de programmation Python. Ce langage est complet, simple à maîtriser et lorsqu'allié au moteur audio Csound, il devient un puissant outil pour la gestion d'algorithmes musicaux. La création du *Cycle des voix* démontre que le développement du logiciel *Ounk* en est à un état suffisamment avancé pour être utilisé dans la gestion de projets musicaux complexes. La prochaine étape dans le développement du logiciel consistera à permettre à l'utilisateur de jouer sur un nombre illimité de canaux de sortie. Ce développement sera appuyé par un projet de composition d'une oeuvre générative à 32 voix pour un dôme de haut-parleurs.

SOURCES DOCUMENTAIRES

Bilmes, J. (2003). Lecture 1. EE516 Computer Speech Processing. University of Washington, Dept. of Electrical Engineering.

Blumstein, S. and Stevens, K. (1979). Acoustic invariance in speech production : evidence from measurements of the spectral characteristics of stop consonants. *Journal of the Acoustical Society of America*, 66 :1001–1017.

Bélanger, O. (2008). Ounk - an audio scripting environment for signal processing and music composition. In *Proc. International Computer Music Conference*, pages 399 – 402, Belfast, Irlande du Nord.

Bélanger, O. and Traube, C. (2005). Modélisation du didjéridou et de ses modes de jeu. In *Proc. Conference on Interdisciplinary Musicology*, pages 21–23, Montréal, Canada.

Bélanger, O., Traube, C., and Piché, J. (2007). Designing and controlling a source-filter model for naturalistic and expressive singing voice synthesis. In *Proc. International Computer Music Conference*, pages 328 – 331, Copenhagen, Danemark.

Chafcouloff, M. (2004). Voir la parole. *Travaux Interdisciplinaires du Laboratoire Parole et Langage d'Aix-en-Provence*, 23 :23–65.

Coleman, R. (1971). Male and female voice quality and its relationship to vowel formant frequencies. *Journal of speech and hearing research*, 14 :565–577.

Cook, P. (1991). *Identification of control parameters in an articulatory vocal tract model with applications to the synthesis of singing*. Phd thesis, Stanford University.

D'Alessandro, N., d'Alessandro, C., Le Beux, S., and Doval, B. (2006). Real-time calm synthesizer, new approaches in hands-controlled voice synthesis. In *Proc. International Conference on New Interfaces for Musical Expression*, pages 266 – 271, Paris, France.

Delattre, P. (1970). Des indices acoustiques aux traits pertinents. In *Proceedings of the 6th International Congress of Phonetic Sciences*, pages 35–47, Prague, Tchécoslovaquie.

Delattre, P., Liberman, A., and Cooper, F. (1955). Acoustic loci and transitional cues for consonants. *Journal of the Acoustical Society of America*, 27 :769–773.

Delattre, P., Liberman, A., Cooper, F., Harris, K., and Hoffman, H. (1958). Effect of third-formant transitions on the perception of the voiced stop consonants. *Journal of the Acoustical Society of America*, 30 :122–126.

Delvaux, V., Demolin, D., Soquet, A., and Kingston, J. (2004). La perception des voyelles nasales du français. In *Proc. Journée d'Étude sur la Parole*, pages 157–160, Fès, Maroc.

Everest, A. (1989). *The Master Handbook of Acoustics*. TAB BOOK Inc., second edition.

Fletcher, N. (1996). The didjeridu (didgeridoo). *Acoustics Australia*, 24 :11–15.

Fletcher, N., Hollenberg, L., Smith, J., and Wolfe, J. (2001a). Didjeridu acoustics. Technical report, University of New South Wales, Sydney, Australia.

Fletcher, N., Hollenberg, L., Smith, J., and Wolfe, J. (2001b). The didjeridu and the vocal tract. In *Proc. International Symposium on Musical Acoustics*, pages 87–90, Perugia, Italie.

Henrich, N., Doval, B., and D'Alessandro, C. (2003). The voice source as a causal/anticausal linear filter. In *ISCA ITRW VOQUAL*, pages 15–19, Geneva, Suisse.

Iturbide, M. R. (nil). La synthèse granulaire et la synthèse par formants. http ://www.artesonoro.net/tesisgran/cap4/cap4.html.

Jackson, P. (2001). Acoustic cues of voiced and voiceless plosives for determining place of articulation. In *Proc. Workshop on Consistent and Reliable Acoustic Cues for Sound Analysis*, pages 19–22, Aalborg, Denmark.

Joliveau, E., Smith, J., and Wolfe, J. (2004). Vocal tract resonances in singing : The soprano voice. *Journal of the Acoustical Society of America*, 116 :2434–2439.

Lee, E. (nil). Parametric speech synthesis. http ://ptolemy.eecs.berkeley.edu/ eal/audio/voder.html.

Lisker, L. (1975). Is it vot or a first-formant transition detector ? *Journal of the Acoustical Society of America*, 57 :1547–1551.

Lu, H.-L. (2002). *Toward a high-quality singing synthesizer with vocal texture control*. Phd thesis, Stanford University.

Martin, P. (2000). identification des sons du français. http ://www.lli.ulaval.ca/labo2256/illust.htmlvoyaco.

McCartney, J. (1998). Continued evolution of the super-collider real time environment. In *Proc. International Computer Music Conference*, pages 133 – 136, Ann Arbor, USA.

Niyorgi, P. and Ramesh, P. (1998). Incorporating voice onset time to improve letter recognition accuracies. In *Proc. IEEE International Conference on Acoustics Speech and Signal Processing*, pages 13 – 16.

Rodet, X. (1984). Time-domain formant wave-function synthesis. *Computer Music Journal*, 8 :9–14.

Rodet, X., Potard, Y., and Barrière, J.-B. (1984). The chant project : From synthesis of the singing voice to synthesis in general. *Computer Music Journal*, 8 :15–31.

Smith, J. O. (2005). Virtual acoustic musical instruments : Review of models and selected research. In *Proc. IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, Mohonk Mountain House, New Paltz, New York.

Sundberg, J. (1977). The acoustics of the singing voice. *Scientific American*, Mars :82–91.

Verfaille, V., Guastavino, C., and Depalle, P. (2005). Perceptual evaluation of vibrato models. In *Proc. Conference on Interdisciplinary Musicology*, pages 149–151, Montréal, QC, Canada.

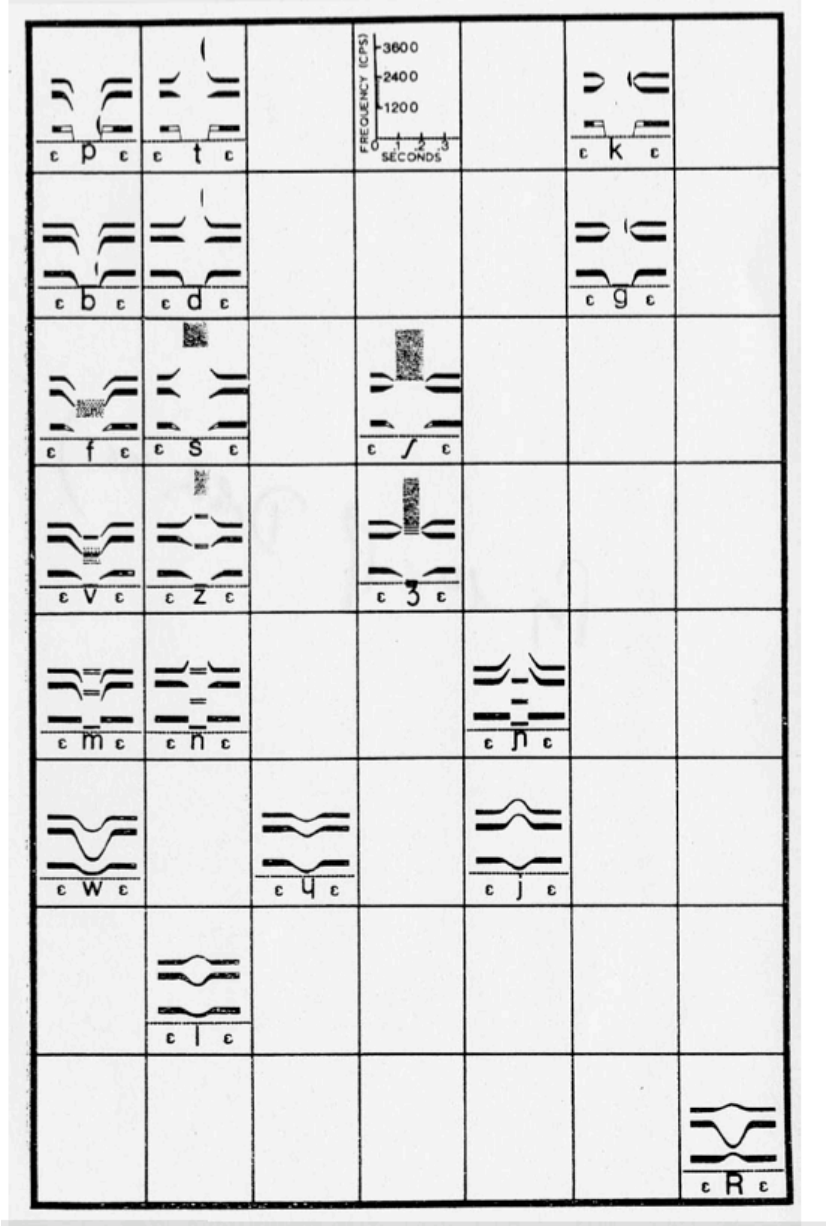
Wang, G. (2008). A comment on many-core computing and real-time audio software systems. In *Proc. International Computer Music Conference*, Belfast, Irlande du Nord.

Wessel, D. (2008). Reinventing audio and music computation for many-core processors. In *Proc. International Computer Music Conference*, Belfast, Irlande du Nord.

Wright, M. and Freed, A. (1997). Opensound control : A new protocol for communicating with sound synthesizers. In *Proc. International Computer Music Conference*, pages 101 – 104, San Francisco.

Annexe I

Trajectoires de formants (Delattre, 1970)



Annexe II

Libraire de fonctions Ounk

Fonctions servant à écrire le fichier Csound

Catégorie	**Functions**
Analysis	attackDetector, centroid, pitchAmp, rms
Controls	busMix, busScale, expsegr, keyPressed, lfo, lfo2, linsegr, metro, midiCtl, midiTrigger, midiTriggerInc, mouse, oscReceive, oscSend, oscTriggerSend, printMidiCtl, rando, randomChoice, randomh, randomi, readTable, spline, trigEnvelope, trigRandom, vibrato, weightedRandomChoice
Custom Inst	myInstrument
Effects	compressor, delay, didjeridu, distortion, flanger, fold, harmonizer, phaser, resonator, ringMod, vdelay, vocoder
Filters	bandpass, bandreject, dcblock, eqFilter, highpass, lowpass, resFilter
GenTables	genAdsr, genDataTable, genExpseg, genLineseg, genRhythmTable, genSoundTable, genWaveform, genWindow, reGenAdsr, reGenDataTable, reGenExpseg, reGenLineseg, reGenSoundTable, reGenWaveform, reGenWindow, recordBuffer, tableMorphing
General	clear, getAudioAttributes, getImpulsePath, getMidiPath, getSoundInfo, getSoundList, getSoundPath, getSoundfontPath, onStop, setAudioAttributes, setAudioDevice, setChannels, setGlobalDuration, setImpulsePath, setMidiDevice, setMidiPath, setSoundPath, setSoundfontPath, speakersConfig
Loops	beginLoop, endLoop
Midi Synth	beginMidiSynth, endMidiSynth, midiSynthCtl, midiSynthGetBus, readMidiFile, splitKeyboard, splitVelocity
Output	createLadspaPlugin, directOut, getPid, recordPerf, route, startCsound, stopCsound, toDac, udpAudioReceive, udpAudioSend
Python Inst	beginPythonInst, endPythonInst
Reverbs	convolveReverb, infiniteReverb, reverb, waveguideReverb
Sampling	recordAndLoop
Sequencer	beginSequencer, endSequencer, seqParameterTable, sequencerPitchTable
Sources	bell, freqMod, inputMic, pinkNoise, playSound, pluckedString, sawtooth, sine, soundfont, soundfontMidi, square, train, voiceSynth, waveform, whiteNoise
Spatialisation	pan1to2, pan1to4, pan1to8, pan2to4, pan2to8, pan4to8, panner, spat1to4
Spectral	arpegiator, blur, crossSynth, fftBandpass, fftBandreject, freeze, maskFilter, reSynth, smooth, transpose
Table process	granulator, granulator2, granulator3, looper, soundTableRead, warper
Trigged Inst	beginTrigInst, endTrigInst

Fonctions servant à écrire des algorithmes en Python

Catégorie	**Functions**
Algorithmic	chord, droneAndJump, drunk, getValue, line, loopseg, mapper, markov, midiToHertz, midiToTranspo, mkChangeOrder, mkRecord, mkSetList, mkStartPlayback, mkStartRecord, next, oneCall, repeater, scale
GUI	beginGUI, endGUI, makeButton, makeCircleSlider, makeMenu, makeMultiSlider, makeSlider, makeSpin, makeToggle, makeXYSlider
Patterns	changePattern, changeTime, getBar, getBeat, getTime, pattern, regenerate, rhythmPattern, sendCsoundMsg, sendEvent, sendOscControl, sendOscTrigger

Annexe III

Reich

```
################################################################################
################################################################################
###                            R E I C H                                     ###
###                      Voice synthesis music                               ###
###                            Ounk script                                   ###
###                      2008, Olivier Belanger                              ###
################################################################################
################################################################################

import random

#setAudioDevice(onumber=3)
octoSetup(1,2,6,5,8,3,7,4)

RES = 'high'
CHANNELS = 8
DUR = 330
TAP = 16

setGlobalDuration(DUR)

if RES == 'low':
    setAudioAttributes(samplingrate=22050, controlrate=2205, sampleformat=16)
else:
    setAudioAttributes(samplingrate=48000, controlrate=12000, sampleformat=24)

setChannels(1)

def setRhythm(v, tap, p1, p2, p3):
    rlist = [genRhythmTable(tap, p1, p2, p3) for i in range(2)]
    randomh(bus=v, rate=random.uniform(.05,.2), portamento=1)
    table = tableMorphing(rlist, v, tap)
    return table

def setVoice(vowel, amplitude, voicetype, vibratoamp, out):
    consonant = random.randint(1,3)
    vowelarti = .75 #random.random() * .3 + .8
    formantres = random.random() * .15 + .5
```

```
    brilli = random.random() * .1 + .8
    if voicetype == 2:
        glotalres = 2
        rough = .05
    else:
        glotalres = .3 #random.random() * .5 + .3
        rough = random.random() * .05 + .1
    voiceSynth(amplitude = amplitude, vowel = vowel, consonant = consonant,
               fadeout = 0.05, duration = .25, voicetype = voicetype,
               vowelarticulation = vowelarti, formantresonance = formantres,
               brilliance = brilli, glotalresonance = glotalres,
               vibratoamp = vibratoamp, roughness = rough, out = out)

scl = [scale('Cm', format='hertz'), scale('C', format='hertz'),
       scale('Dm', format='hertz')]

# global amplitude
linsegr('globalamp', .85, DUR-41.570, .85, 10, 1.1, .01, 0, 32.429, 0)

# TIMES #
SEQSTART = 0
SEQDUR = DUR
SEQ1DUR = 182.1
SEQ2DUR = SEQDUR - SEQ1DUR

# Rhythm tables
s1 = setRhythm(1, TAP, 60,85,70)
s2 = setRhythm(2, TAP, 70,80,75)
a1 = setRhythm(3, TAP, 70,70,70)
a2 = setRhythm(4, TAP, 65,80,80)
t1 = setRhythm(5, TAP, 85,70,75)
t2 = setRhythm(6, TAP, 90,80,60)
b1 = setRhythm(7, TAP, 95,70,40)

# pitch generators
seq = loopseg(37, 43) # soprano
seq2 = loopseg(35, 40) # alto
dro = loopseg(28,36) # tenor
rep = loopseg(40, 45) # bass

# lists of pitch tables
pitsop1 = []
pitsop2 = []
pitalto1 = []
```

```
pitalto2 = []
pittenor1 = []
pittenor2 = []
pitbass1 = []
for i in range(10):
    scnum = random.randint(0,2)
    pitsop1.append(genDataTable([scl[scnum][seq.next()] for i in range(TAP)]))
    pitsop2.append(genDataTable([scl[scnum][seq.next()] for i in range(TAP)]))
    pitalto1.append(genDataTable([scl[scnum][seq2.next()] for i in range(TAP)]))
    pitalto2.append(genDataTable([scl[scnum][seq2.next()] for i in range(TAP)]))
    pittenor1.append(genDataTable([scl[scnum][dro.next()] for i in range(TAP)]))
    pittenor2.append(genDataTable([scl[scnum][dro.next()] for i in range(TAP)]))
    pitbass1.append(genDataTable([scl[scnum][rep.next()] for i in range(TAP)]))

randomChoice(bus=['pitvar', 'pitvar'],
             choice=[0,.111,.222,.333,.444,.555,.666,.777,.888,.999],
             rate=[0,.05], starttime=[SEQSTART, SEQSTART+SEQ1DUR],
             duration=[SEQ1DUR,SEQ2DUR])

sop1 = tableMorphing(pitsop1, 'pitvar', TAP)
sop2 = tableMorphing(pitsop2, 'pitvar', TAP)
alto1 = tableMorphing(pitalto1, 'pitvar', TAP)
alto2 = tableMorphing(pitalto2, 'pitvar', TAP)
tenor1 = tableMorphing(pittenor1, 'pitvar', TAP)
tenor2 = tableMorphing(pittenor2, 'pitvar', TAP)
bass1 = tableMorphing(pitbass1, 'pitvar', TAP)

# sequencer note duration
linsegr(bus='notedur', i1=1, dur1=SEQ1DUR, i2=0, dur2=0,
        i3=1, starttime=SEQSTART, duration=SEQDUR)

notedur1 = genDataTable([.5]*TAP)
notedur2 = genDataTable([.2]*TAP)
notedur = tableMorphing(tablelist=[notedur1, notedur2], indexVar='notedur',
                        length=TAP, starttime=SEQSTART, duration=SEQDUR)

linsegr(bus='fadedur', i1=1, dur1=SEQ1DUR, i2=0, dur2=0,
        i3=2, starttime=SEQSTART, duration=SEQDUR)

fadedur1 = genDataTable([.075]*TAP)
fadedur2 = genDataTable([.075]*TAP)
fadedur = tableMorphing(tablelist=[fadedur1, fadedur2], indexVar='notedur',
                        length=TAP, starttime=SEQSTART, duration=SEQDUR)
```

```
# metronome
linsegr(bus='tempVar', i1=.5, dur1=SEQ1DUR, i2=.5, dur2=0, i3=1,
        dur3=SEQ2DUR, i4=1, starttime=SEQSTART, duration=SEQDUR)
randomChoice(bus='tapVar', choice=[.125,.25,.25,.25,.25, 1], rate=.2)

metro(bus='metro', tempo=132, tap=TAP, tempoVar='tempVar',
      tapVar='tapVar', starttime=SEQSTART, duration=SEQDUR)

# SEQUENCES
# soprano voices
beginSequencer(input='metro', table=s1)
seqParameterTable(['pitch', 'duration', 'fadeout'], [sop1, notedur, fadedur])
setVoice(vowel=0, amplitude=.5, voicetype=3, vibratoamp=0, out='v1')
endSequencer(starttime=SEQSTART, duration=SEQDUR)

beginSequencer(input='metro', table=s2)
seqParameterTable(['pitch', 'duration', 'fadeout'], [sop2, notedur, fadedur])
setVoice(vowel=0, amplitude=.5, voicetype=3, vibratoamp=0, out='v2')
endSequencer(starttime=SEQSTART, duration=SEQDUR)

# alto voices
beginSequencer(input='metro', table=a1)
seqParameterTable(['pitch', 'duration', 'fadeout'], [alto1, notedur, fadedur])
setVoice(vowel=0, amplitude=.7, voicetype=1, vibratoamp=0, out='v3')
endSequencer(starttime=SEQSTART, duration=SEQDUR)

beginSequencer(input='metro', table=a2)
seqParameterTable(['pitch', 'duration', 'fadeout'], [alto2, notedur, fadedur])
setVoice(vowel=0, amplitude=.7, voicetype=1, vibratoamp=0, out='v4')
endSequencer(starttime=SEQSTART, duration=SEQDUR)

# tenor voices
beginSequencer(input='metro', table=t1)
seqParameterTable(['pitch', 'duration', 'fadeout'], [tenor1, notedur, fadedur])
setVoice(vowel=4, amplitude=1.5, voicetype=0, vibratoamp=0, out='v5')
endSequencer(starttime=SEQSTART, duration=SEQDUR)

beginSequencer(input='metro', table=t2)
seqParameterTable(['pitch', 'duration', 'fadeout'], [tenor2, notedur, fadedur])
setVoice(vowel=4, amplitude=1.5, voicetype=0, vibratoamp=.01, out='v6')
endSequencer(starttime=SEQSTART, duration=SEQDUR)

# bass voices (now soprano colorature)
beginSequencer(input='metro', table=b1)
```

```
seqParameterTable(['pitch', 'duration', 'fadeout'], [bass1, notedur, fadedur])
setVoice(vowel=0, amplitude=.4, voicetype=3, vibratoamp=.001, out='v7')
endSequencer(starttime=SEQSTART, duration=SEQDUR)

# EFFECTS #
# harmonic pad
PADSTART = SEQSTART
PADDUR = SEQ1DUR - .1
pit = [scl[0][i] for i in range(20, 56, 5)]
amp = [.2,.15,.12,.09,.05,.03,.02,.01]
typ = [2,2,0,0,1,1,3,3]
vib = [.01,.01,.005,.005,.003,.003,.001]
linsegr(bus='sinusbass', i1=0, dur1=.01, i2=1, dur2=SEQ1DUR-.02,
        i3=1, dur3=.01, i4=0, starttime=PADSTART, duration=PADDUR)
for i in range(7):
    voiceSynth(pitch=pit[i], amplitude=amp[i]*.85, duration=PADDUR,
               starttime=PADSTART, fadein=0.0001, fadeout=.1, brilliance=.8,
               voicetype=typ[i], vibratoamp=vib[i], out=str(i+1))
    sine(pitch=[32.7033, 48.9997], amplitude=[.03,.009],
         amplitudeVar='sinusbass', out=str(i+1))

# bass drone
DRONESTART = SEQSTART + SEQ1DUR - .02
DRONEDUR = SEQ2DUR

bassmult = [1, 1.12246, 1.33484, 0.89089889, 1, 1.12246, 1.49831, 1.33484,
            1.12246, 1.33484, 0.89089889, 1, 1.12246, 1, 0.89089889, 1.33484,
            1.12246, 1, 1.49831, 1.12246, 1, 0.89089889, 1.12246, 1]
bassstarts = [0, 7.2727, 10.90905, 14.5454, 21.8181, 25.45455, 32.72715, 36.3635,
              43.6362, 47.27255, 50.9089, 58.1816, 61.81795, 69.09065, 72.727,
              79.9997, 83.63605, 87.2724, 90.90875, 98.18145, 101.8178, 109.0905,
              112.72685, 116.3632]
bassdurs = [7.2727, 3.63635, 3.63635, 7.2727, 3.63635, 7.2727, 3.63635, 7.2727,
            3.63635, 3.63635, 7.2727, 3.63635, 7.2727, 3.63635, 7.2727, 3.63635,
            3.63635, 3.63635, 7.2727, 3.63635, 7.2727, 3.63635, 3.63635]
for j in range(24):
    for i in range(7):
        bstart = DRONESTART + bassstarts[j]
        if j < 23:
            bdur = bassdurs[j]
            fade = .075
        else:
            bdur = 30
            fade = 25
```

```
        basspit = [32.7033,32.7087,65.4068,65.4013,32.7011,32.6999,65.41]
        bassmu = bassmult[j]
        voiceSynth(pitch=basspit[i]*bassmu, amplitude=2.5, voicetype=2, fadein=.04,
                   fadeout=fade, brilliance=.8, starttime=bstart, glotalresonance=1,
                   duration=bdur, out='bass%d' % (i+1))
        expsegr('sinamp', 0.001, .04, 1, bdur - .23, 1, .2, 0.0001,
                starttime=bstart, duration=bdur)
        pit = random.uniform(.997, 1.003) * 32.7033
        sine(pitch = pit*bassmu, amplitude = .07, starttime=bstart, duration=bdur,
             amplitudeVar='sinamp', out='bass%d' % (i+1))
        lowpass(input='bass%d' % (i+1), amplitude=1, cutoff=1000,
                starttime=bstart, duration=bdur, out=str(i+1))


# high pitch delta
if RES == 'low':
    DELTASTART = SEQSTART + SEQ1DUR - 30 # 30.5
else:
    DELTASTART = SEQSTART + SEQ1DUR - 30.07
DELTADUR = 48
delta = genExpseg([0.001, 29.25, .15, .75, 1, .005, .1, 17.995, .001])
readTable(bus='delta', table=delta, starttime=DELTASTART, duration=DELTADUR)
for i in range(7):
    n = [1,2,3,4,5,6,7][i]
    hpit = scl[0][52:59][i]
    voiceSynth(pitch=hpit, starttime=DELTASTART, duration=DELTADUR,
               amplitude=.4, voicetype=3, vowel=4, fadeout=0,
               vibratoamp=0, brilliance=.8, out='vnoise%d' % n)
    route(input='vnoise%d' % n, starttime=DELTASTART,
          duration=DELTADUR, amplitudeVar='delta', out=str(n))

# high pitch delta 2
if RES == 'low':
    DELTA2START = SEQ1DUR + 116.3432 - 30 # 30.8
else:
    DELTA2START = SEQ1DUR + 116.3432 - 30.34
DELTA2DUR = 60
delta2 = genExpseg([0.001, 29, .15, 1, 1, .005, .05, 29.995, .001])
readTable(bus='delta2', table=delta2, starttime=DELTA2START, duration=DELTA2DUR)
for i in range(7):
    n = [1,2,3,4,5,6,7][i]
    hpit = scl[0][54:61][i]
    voiceSynth(pitch=hpit, starttime=DELTA2START, duration=DELTA2DUR,
               amplitude=.3, voicetype=3, vowel=4, fadeout=0,
```

```
                vibratoamp=0, brilliance=.7, out='vnoise2%d' % n)
    route(input='vnoise2%d' % n, starttime=DELTA2START,
          duration=DELTA2DUR, amplitudeVar='delta2', out=str(n))

# Noisy
NOISEDUR = SEQ1DUR
noisetable = genLineseg([0,1,1,5,1,20,.2,1000,.1, 1000, 0])
randomi(bus='lfo', mini=.5, maxi=1.4, rate=.2, duration=NOISEDUR)
lfo2(bus='noise', table=noisetable, frequency=.1136363636,
     offset=0, frequencyVar='lfo', duration=NOISEDUR)
for i in range(7):
    voiceSynth(amplitude=1.25, voicetype=random.randint(0,3),
                vowel=random.randint(0,6), fadein=.001, fadeout=10,
                roughness=1, formantresonance=.15, vibratoamp=0.0005,
                duration=NOISEDUR, out='noisy%d' % (i+1))
    route(input='noisy%d' % (i+1), amplitudeVar='noise',
          duration=NOISEDUR, out=str(i+1))

# Amplitudes
randomh(bus=['var1','var2','var3','var4','var5','var6','var7'],
        mini=0.5, maxi=1.3, portamento=0.25, rate=[.2,.28,.23,.43,.47,.52,.58])

busMix('amp1', 'globalamp', 'var1', 'times')
busMix('amp2', 'globalamp', 'var2', 'times')
busMix('amp3', 'globalamp', 'var3', 'times')
busMix('amp4', 'globalamp', 'var4', 'times')
busMix('amp5', 'globalamp', 'var5', 'times')
busMix('amp6', 'globalamp', 'var6', 'times')
busMix('amp7', 'globalamp', 'var7', 'times')

route(input='v1', amplitudeVar='amp1', out='7')
route(input='v2', amplitudeVar='amp2', out='3')
route(input='v3', amplitudeVar='amp3', out='5')
route(input='v4', amplitudeVar='amp4', out='1')
route(input='v5', amplitudeVar='amp5', out='6')
route(input='v6', amplitudeVar='amp6', out='2')
route(input='v7', amplitudeVar='amp7', out='4')

# output, reverb and panoramisation
if RES == 'low':
    for i in range(7):
        route(input=str(i+1), amplitude=.75, out='rev%d' % (i+1))
else:
    if CHANNELS == 2:
```

```
        for i in range(7):
            route(input=str(i+1), amplitude=.4, out='rev%d' % (i+1))
    elif CHANNELS == 8:
        for i in range(7):
            convolveReverb(input=str(i+1), impulse='ORTFFloorR10_m48.wav',
                           amplitude=.8, mix=.025, out='rev%d' % (i+1))

if CHANNELS == 2:
    pan1to2(input='rev1', pan=0)
    pan1to2(input='rev2', pan=0.166)
    pan1to2(input='rev3', pan=0.333)
    pan1to2(input='rev4', pan=0.5)
    pan1to2(input='rev5', pan=0.666)
    pan1to2(input='rev6', pan=0.833)
    pan1to2(input='rev7', pan=1)
elif CHANNELS == 8:
    pan1to8(input='rev1', pan=0)
    pan1to8(input='rev2', pan=0.166)
    pan1to8(input='rev3', pan=0.333)
    pan1to8(input='rev4', pan=0.5)
    pan1to8(input='rev5', pan=0.666)
    pan1to8(input='rev6', pan=0.833)
    pan1to8(input='rev7', pan=1)

if RES == 'low':
    startCsound()
else:
    startCsound(file='Reich')
```

Annexe IV

Chorus Morph

```
##############################################################################
##############################################################################
###                     C H O R U S  M O R P H                             ###
###                      Voice synthesis music                             ###
###                      Ounk script (part 1)                              ###
###                     2008, Olivier Belanger                             ###
##############################################################################
##############################################################################

import random, math
import chorus_setup
reload(chorus_setup)
#setAudioDevice(onumber=3)
octoSetup(1,2,6,5,8,3,7,4)

if chorus_setup.RES == 'low':
    setAudioAttributes(samplingrate=44100, controlrate=4410, sampleformat=16)
elif chorus_setup.RES == 'high':
    setAudioAttributes(samplingrate=48000, controlrate=12000, sampleformat=24)
setGlobalDuration(-1)

beginPythonInst()
voiceSynth(pitch=220, duration=45, vowel=4, fadeout=5, pitchglissando=20,
           vowelglissando=20, brilliance=.96, glotalresonance=1,
           vibratoamp=0.001, roughness=0, out='voice')
endPythonInst()

if chorus_setup.RES == 'low':
    reverb(input='voice', revtime=1.5, mix=.3, amplitude=.5, out='rev')
elif chorus_setup.RES == 'high':
    convolveReverb(input='voice', mix=.03, out='rev')

if chorus_setup.CHANNELS == 2:
    toDac(input='rev', amplitude=1.7)
elif chorus_setup.CHANNELS == 8:
    pan2to8(input='rev', pan=0)
    setChannels(8)
```

```
if chorus_setup.RECORD:
    recordPerf('Chorus_Morph-1')

proc = startCsound()

def sc(i, mi=1, ma=6):
    return (i - mi) / ((ma - 1.) - (mi - 1))

# soprano voices
numv1 = chorus_setup.NUMHIGH
v1 = 1
def voice1():
    global v1
    proc
    curtime = pat1.getTime()
    pat1.changeTime(random.randint(5000,10000)*0.001)
    if curtime <= 60:
        p1 = 10
        p2 = mapper(curtime,0,60,40,10)
    elif curtime <= 200:
        p1 = mapper(curtime,60,200,10,1)
        p2 = mapper(curtime,60,200,10,20)
    elif curtime <= 220:
        p1, p2 = 1, 40
    elif curtime <= 260:
        p1 = mapper(curtime,220,260,1,.89)
        p2 = mapper(curtime,220,260,20,80)
    elif curtime <= 280:
        p1, p2 = .89, 80
    else:
        p1, p2 = .89, 80
        pat1.play(False)
        pat2.play(False)
        pat3.play()
        return
    pitch = random.weibullvariate(p1,p2) * 250
    if pitch < 500: amp = .3
    else: amp = .03
    dict = {}
    dict['voiceSynth'] = {}
    dict['voiceSynth']['duration'] = -1
    dict['voiceSynth']['voicetype'] = 1
    dict['voiceSynth']['amplitude'] = amp
    dict['voiceSynth']['pitch'] = pitch
```

```
    dict['voiceSynth']['vowel'] = random.randint(0,11)
    dict['voiceSynth']['voicenumber'] = v1
    dict['voiceSynth']['vowelglissando'] = random.uniform(2,5)
    dict['voiceSynth']['pitchglissando'] = random.uniform(2,5)
    dict['voiceSynth']['formantresonance'] = random.uniform(.75,1)
    dict['voiceSynth']['brilliance'] = random.uniform(.9,.96)
    dict['voiceSynth']['vibratospeed'] = random.uniform(2.,6.)
    dict['voiceSynth']['vibratoamp'] = random.random() * 0.015
    dict['voiceSynth']['pan'] = sc(v1,1,numv1)
    sendEvent(1, dict, proc)
    v1 += 1
    if v1 > numv1: v1 = 1

# bass voices
numv2 = chorus_setup.NUMBASS
v2 = 1
def voice2():
    global v2
    curtime = pat2.getTime()
    pat2.changeTime(random.randint(5000,10000)*0.001)
    if curtime <= 80:
        p1 = 0.03
        p2 = 20
    elif curtime <= 140:
        p1 = mapper(curtime,80,140,.03,.3)
        p2 = 20
    elif curtime <= 220:
        p1 = .3
        p2 = 20
    elif curtime <= 260:
        p1 = mapper(curtime,220,260,.3,.89)
        p2 = mapper(curtime,220,260,20,80)
    elif curtime <= 300:
        p1 = .89
        p2 = 80
    else:
        pass
    pitch = random.weibullvariate(p1,p2) * 250
    if pitch < 30:
        amp = 1.5
        rough = .99
        gliss = .05
    elif pitch < 60:
        amp = 1.2
```

```
        rough = .8
        gliss = 1
    elif pitch < 100:
        amp = .9
        rough = .3
        gliss = 2
    else:
        amp = .7
        rough = .05
        gliss = 3
    dict = {}
    dict['voiceSynth'] = {}
    dict['voiceSynth']['duration'] = -1
    dict['voiceSynth']['voicetype'] = 2
    dict['voiceSynth']['amplitude'] = amp
    dict['voiceSynth']['pitch'] = pitch
    dict['voiceSynth']['vowel'] = random.randint(0,6)
    dict['voiceSynth']['voicenumber'] = v2 + 100
    dict['voiceSynth']['fadein'] = 10
    dict['voiceSynth']['vowelglissando'] = gliss
    dict['voiceSynth']['pitchglissando'] = 5
    dict['voiceSynth']['roughness'] = rough
    dict['voiceSynth']['formantresonance'] = random.uniform(.75,1)
    dict['voiceSynth']['glotalresonance'] = 1
    dict['voiceSynth']['brilliance'] = random.uniform(.9,.96)
    dict['voiceSynth']['vibratoamp'] = 0.015
    dict['voiceSynth']['pan'] = 1 - sc(v2,1,numv2)
    sendEvent(1, dict, proc)
    v2 += 1
    if v2 > numv2: v2 = 1

# final
v3 = 1
def final():
    global v3
    if v3 <= numv1:
        voix = v3
        vtype = 1
        amp = .3
        pan = sc(voix,1,numv1)
    else:
        voix = v3 - numv1 + 100
        vtype = 2
        amp = .7
```

```
        pan = 1 - sc(v3-numv1,1,numv2)
    dict = {}
    dict['voiceSynth'] = {}
    dict['voiceSynth']['voicenumber'] = voix
    dict['voiceSynth']['voicetype'] = vtype
    dict['voiceSynth']['amplitude'] = amp
    dict['voiceSynth']['pan'] = pan
    sendEvent(1, dict, proc)

    v3 += 1
    if v3 == numv1 + numv2 + 1:
        pat3.play(False)

pat1 = pattern(time=5, function=voice1)
pat2 = pattern(time=5, function=voice2)
pat3 = pattern(time=.25, function=final)
pat1.start()
pat2.start()
pat3.start()
pat1.play()
pat2.play()

voice1()
voice2()
```