libTuile : un moteur d'exécution multi-échelle de processus musicaux hiérarchisés

Florent Berthaut, David Janin, Myriam Desainte-Catherine

LaBRI, CNRS UMR 5800
Laboratoire Bordelais de Recherche en Informatique

Rapport de recherche RR-1468-13

libTuile : un moteur d'exécution multi-échelle de processus musicaux hiérarchisés

21 février 2013

Florent Berthaut, David Janin et Myriam DeSainte-Catherine

Université de Bordeaux
LaBRI UMR 5800
351, cours de la libération
F-33405 Talence, FRANCE

Table des matières

libTuile : un moteur d'exécution multi-échelle de processus musicaux hiérarchisés

Florent Berthaut, David Janin et Myriam DeSainte-Catherine *
Université de Bordeaux, LaBRI UMR 5800,
351, cours de la libération,
F-33405 Talence
{berthaut|janin|myriam}@labri.fr

21 février 2013

Résumé

La *libTuile* est une librairie expérimentale réalisée afin d'évaluer le potentiel multi-échelle et interactif de la modélisation par *tuilage* de processus musicaux hiérarchisés.

1 Introduction

1.1 Problématique

Il existe aujourd'hui de nombreux langages d'écriture et de traitements sonores, qu'ils soient textuels comme *Supercollider/chuck* [14] ou *Faust*[7], ou visuels comme *Max/Msp* et *PureData* [5]. L'écriture de pièces musicales reste cependant une tâche délicate car ces langages ne proposent que bien peu de métaphores ou de paradigmes de programmation permettant de décrire facilement (et de façon compositionnelle) le placement temporel des objets sonores à produire. Plus encore, si l'on souhaite intégrer une gestion d'évènements externes - déclencheurs de processus audio - le maintien de la cohérence temporelle et/ou rythmique de la pièce écrite peut devenir inextricable.

C'est pour remédier à cela que le séquenceur interactif *i-score* [2] propose explicitement un outil de spécification du placement des objets sonores les uns par rapport au autre. En intégrant des points de contrôle explicites et un mécanisme de résolution de contraintes de placement, il permet une *écriture du temps* plus abstraite. Néanmoins, faute de mécanismes de contrôles dynamiques puissants - conditionnelle, boucle - l'applicabilité d'*i-score* reste encore limitée.

*ce travail est partiellement soutenu par le projet CONTINT 2012 - ANR 12 CORD 009 02 - INEDIT

Le travail présenté ici, qui s'appuie sur une extension d'*i-score*, vise à expérimenter les possibilités d'intégration des outils de description programmatique de pièce musicale qui sont offerts par les langages évoqués ci-dessus, avec cette capacité d'*écriture (dynamique) du temps* offerte par *i-score*.

1.2 Caractéristiques fonctionnelles

Implémentation d'une proposition récente d'algèbre de synchronisation de signaux audio ou musicaux tuilés [4], la *libTuile* apparait avant tout comme un *outil de mixage* multi-échelle et hiérarchique qui, en intégrant un module de lecture granulaire de flux audio, se révèle d'un usage particulièrement souple.

Couplée avec d'autres outils existants de l'informatique musicale tels que la librairie audio *libAudioStream* [6], le langage synchrone *Faust*[7] ou le séquenceur interactif *i-score*[2], la *libTuile* vise à devenir le premier prototype de *moteur d'exécution* pour le T-calcul, une proposition de langage de programmation intégrant la programmation par tuilage (ou *tiled programing*) [11].

Dans le prolongement du logiciel interactif *Drile* [3], un mécanisme d'execution cyclique et hiérarchique associé à la *libTuile* offre aussi, via un couplage avec le moteur temps-réel *JackAudio* et une interface, le *simpleTuilesLooper*, un outil de *performance live* qui nous permet d'expérimenter les métaphores et les concepts sous-jacents.

2 Modéles de processus musicaux

Dans cette section, nous passons en revue quelques un des concepts clés qui sous-tendent la *libTuile*.

2.1 Modélisation à quatre dimensions

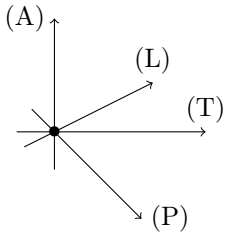
Il existe aujourd'hui de nombreux modèles de représentation de la musique qui répondent à des contraintes d'usages variés. Par exemple, la *partition musicale classique* est largement dédiée à une lecture rapide par les musiciens de phrases mélodiques, qui peuvent être multiples, et qu'ils devront *interpréter* et *synchroniser* entre elles.

Plus informatique, les *piano rolls* des séquenceurs audio et midi visent en particulier à expliciter un placement temporel plus fin qui précise le positionnement relatif des attaques et des fins les unes par rapport aux autres ou par rapport à une pulsation de référence.

Les *grilles d'accords* ou *basses chiffrées* qui peuvent étiqueter les partitions classiques offrent une vue plus abstraite du contexte harmonique sous-jacent conduisant les musiciens à improviser tout en respectant l'esprit sinon le texte de la partition. Des *regroupements* en accords, mesures, motifs, phrases, mouvements, etc., visent aussi à décrire des structures de plus en plus abstraites [13] pouvant rendre compte, parmi d'autres aspects, de la conformité à un style musical ou de l'intention du compositeur...

Si on ajoute à cela des règles d'interaction, telles qu'elles peuvent apparaître, de façon plus ou moins formalisée, dans les musiques improvisées, on comprend qu'il s'agit là, au-delà des exemples évoqués ci-dessus, d'un vaste sujet de recherche aux développements potentiellement infinis.

On peut cependant mettre un petit peu d'ordre dans ces multiples formalismes en les classifiant selon les dimensions musicales qu'ils permettent de représenter. En effet, la structure d'une pièce musicale interactive peut commodément être décrite selon quatre dimensions : le temps T, l'alternative L, le parallélisme P et l'abstraction A.



Par exemple, la partition d'un thème de jazz annotée par des accords se positionne dans le plan de l'abstraction : les notes, les lignes mélodiques, les accords, et du temps : l'évolution (et les interdépendances) de chacune des caractéristiques de ces niveaux d'abstractions.

Un *piano roll* se place typiquement dans le plan formé par le parallélisme et le temps. Une *représentation arborescente* des réactions possibles d'un système musical interactif au jeu d'un musicien, representation classique en modélisation des systèmes réactifs, est décrite dans le plan formé par l'alternative et le temps.

2.2 Une dimension temporelle multi-échelle

Dans cet espace de modélisation des processus musicaux, chaque niveau d'abstraction induit des métaphores ou des concepts spécifiques propres à la description des caractéristiques liées à chaque dimension.

Dans la dimension temporelle, on voit apparaître au moins quatre types d'échelles de temps qui semblent de natures différentes.

Le temps logique (causal). Les évènements musicaux sont unitaires, positionnés les uns par rapport aux autres sur une échelle de temps logique, e.g. *avant*, *après*, *en même temps*.

Les intervalles de temps réels entre deux évènements logiques peuvent varier de quelques secondes, par exemple pour des progressions harmoniques, à quelques minutes, pour des successions de thèmes, ou bien plus encore pour, par exemple, des séquences de mouvements voire de pièces musicales entières.

Le temps symbolique (quantifié). Les évènements musicaux ont maintenant une durée et une position temporelle calculées sur une (ou des) échelle(s) de temps symbolique définie par référence à une (ou plusieurs) unité(s) de référence telle que la battue ou la mesure de l'ordre de la seconde.

Les concepts de positionnements temporels logiques évoqués ci-dessus échouent à décrire seuls une structuration temporelle plus complexe faite de superpositions partielles [8].

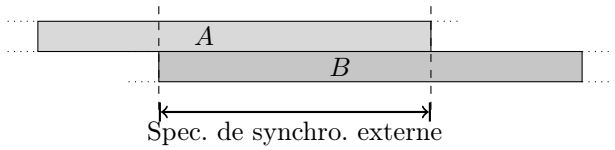
Le temps réel asynchrone (placé). Les évènements musicaux sont maintenant positionnés, par exemple lors d'une performance, sur une échelle de temps réel. Ce placement temporel, perceptible par l'oreille humaine, est réalisé avec une précision de 10^{-1}s à 10^{-3}s.

A cette échelle de temps, l'apparition des évènements reste irrégulière. De nombreux phénomènes de tension et de résolution rythmique peuvent être obtenu, en musique savante comme en musique pop ou traditionnelle, par un positionnement temporel choisi *autour* de la pulsation abstraite. On est donc dans un flux évènementiel asynchrone.

Le temps synchrone («continu»). Découpés en grains ou échantillons réguliers (dans le cas numérique) ou résultant de processus continus (dans le cas acoustique) les sons musicaux obtenus sont maintenant réalisés (ou approximés) sur une horloge régulière de période de 10^{-1}s à 10^{-5}s.

2.3 De la synchronisation aux tuilages

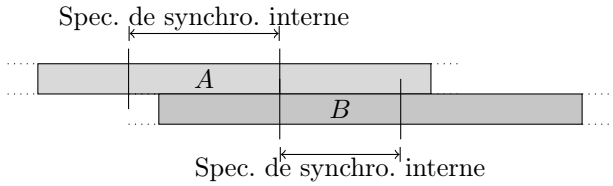
De pratique courante en traitement audio, le mixage consiste a positionner (ou synchroniser) dans le temps, l'un par rapport à l'autre, deux signaux, pour ensuite appliquer une opération de transformation (ou fusion), par exemple un *cross-fade*, sur les parties de signaux ainsi superposées.



La synchronisation de deux signaux requiert en général une analyse de ces signaux afin de déterminer le début et la fin de la zone de recouvrement. Par exemple, le séquenceur interactif *i-score* [2] permet, à l'aide de contraintes exprimées en logique de Allen[1], de spécifier les recouvrements autorisés.

Cette approche par *spécification externe*, qui s'appuie sur les signaux à mixer et l'analyse croisée de leurs superpositions possibles, n'est pas compositionnel. Chaque mixage necessite une analyse des signaux à mixer.

Le *tuilage* apparait dès lors que ces informations de synchronisation sont intégrées aux signaux eux-mêmes [8] produisant ainsi des *signaux tuilés* [4]. Plus précisément, tout comme une tuile de toit, ou une séquence musicale avec barres de mesures, chaque signal peut être associé à un *intervalle de synchronisation*, qui spécifie comment le signal sera positionné lors d'une opération de synchronisation vis à vis des signaux (tuilés) voisins. On passe ainsi d'une spécification externe de la synchronisation à une *spécification interne* de la synchronisation.



Par opposition à cet intervalle de synchronisation, l'intervalle couvert par la totalité du signal est appelé *intervalle de réalisation*.

La synchronisation de deux signaux tuilés revient à positionner les fenêtres de synchronisation de ces signaux en séquence, sans autre analyse ou calcul. L'opération de fusion (paramétrable) est alors appliquée. Le signal obtenu reste tuilé. L'opération de mixage ainsi défini, appelé SEQ, est compositionnelle. Plus encore, combinée à des fonctions de fusions variables, elle se révèle particulièrement expressive.

La structure mathématique sous-jacente qui permet de décrire ces paramètres de tuilages se révèle particulièrement robuste. C'est un monoïde inversif [12]. Une étude théorique récente de la théorie des langages induite par ces structures démontre par ailleurs tout à la fois sa simplicité et sa puissance [10, 9].

On peut aussi dériver de la structure inversive du produit de mixage SEQ de nombreux autres opérateurs [4]. On dispose par exemple d'opérateurs de synchronisation à gauche FORK ou à droite JOIN. L'ajout de deux opérateurs additionnels de resynchronisation RESYNC, agissant sur le position relatif de la fenêtre de synchronisation, et d'expansion/contraction STRETCH, conduit à une algèbre de mixage de signaux tuilés particulièrement souple d'utilisation [4].

Son intégration récente à un langage de programmation : le T-calcul [11] semble particulièrement prometteuse. En effet, dans le produit de synchronisation $A;B$ de deux signaux tuilés A et B on décrit tout à la foi une dépendance abstraite de type causal : A *avant* B, et une opération de synchronisation concrète : le positionnement temporel relatif, à l'échantillon près, du signal associé à A par rapport au signal associé à B.

Autrement, via le tuilage, le T-calcul [11] intègre dans un formalisme unique les deux échelles de temps extrêmes du temps logique (causal) et du temps réel synchrone («continu»). On se convainc sans difficulté qu'il permet aussi de décrire les échelles de temps intermédiaires du temps symbolique (quantifié) et du temps réel asynchrone (placé).

3 Implémentation

Dans cette section, nous décrivons les composants logiciels de la libTuiles, notamment le mécanisme de commandes pour la communication entre threads asynchrones et synchrones.

3.1 LibTuiles : construction et exécution d'arbres de tuiles

LibTuiles est une bibliothèque C++ permettant de construire et d'exécuter des arbres de tuiles. Dans ces arbres, chaque tuile possède un identifiant unique sous forme d'un nombre entier non signé. La construction et la manipulation des arbres de tuiles s'effectuent à l'aide des méthodes suivantes :

addLeaf*(const float& d, unsigned int& id) :* créé une nouvelle tuile feuille de durée initiale d et assigne son identifiant à la variable id.

addLoop*(const unsigned int& id1, unsigned int& id) :* créé une nouvelle tuile résultant de l'application de l'opérateur LOOP sur la tuile id1 et assigne son identifiant à la variable id.

addSeq*(const unsigned int& id1, const unsigned int& id2, unsigned int& id),* **addFork***(...) et* **addJoin***(...) :* créént tous trois une nouvelle tuile résultant de l'application respective des opérateurs SEQ, FORK et JOIN sur les tuiles id1 et id2 et assigne son identifiant à la variable id.

setTuileLength*(const unsigned int& id, const float& d) :* applique l'opérateur STRETCH sur la tuile id afin de redimensionner sa fenêtre de réalisation à la durée d.

setTuileLeftOffset*(const unsigned int& id, const float& lo) :* applique l'opérateur RESYNC sur la tuile id afin de modifier l'offset gauche de sa fenêtre de synchronisation.

setTuileRightOffset*(const unsigned int& id, const float& ro) :* applique l'opérateur RESYNC sur la tuile id afin de modifier l'offset droit de sa fenêtre de synchronisation.

setBpm*(const float& bpm) :* définit le tempo de lecture de l'arbre.

setRoot*(const unsigned int& id) :* définit la tuile id comme racine de l'arbre.

play*() et* **stop***() :* lancent et arrêtent respectivement la lecture de l'arbre.

removeTuile*(const unsigned int& id) :* retire la tuile id de l'arbre.

clear*() :* supprime toutes les tuiles de l'arbre.

De manière interne, la construction et l'exécution des arbres s'effectuent dans un thread séparé, afin d'éviter des ralentissements dus à des calculs effectués dans le thread principal de l'application, e.g. le thread de l'interface graphique. Le mécanisme de communication entre les threads est décrit dans la section 3.3.

Lors de la lecture de l'arbre, la progression temporelle s'effectue dans la racine et est répercutée en descendant dans l'arborescence. Chaque opérateur fait en effet avancer la position dans ses enfants en fonction des paramètres de leurs intervalles de synchronisation et de réalisation. A chaque temps t est donc calculée la position dans chacune des tuiles. Il est de la même façon possible de connaitre la position absolue de chacune des tuiles dans l'arbre. Puisque l'avancement temporel s'effectue pour chacun des nœuds de l'arbre de manière relative à son nœud parent, il est possible de modifier dynamiquement l'arbre pendant la lecture.

Des commandes d'activation et de désactivation sont envoyées depuis le thread de lecture en fonction de la position de la lecture dans la tuile, la tuile étant active entre 0 et la longueur de son intervalle de réalisation. Des commandes de durée sont aussi envoyées lorsque la taille de l'intervalle de réalisation est mise à jour ou lors des changements de tempo, ainsi que des commandes de position absolue quand l'arbre est modifié. Ainsi, un moteur de synthèse/traitement synchrone, tel que celui décrit dans la section 3.2 peut recevoir toutes les informations nécessaires au positionnement temporel des différents processus associés aux tuiles.

Les propriétés des tuiles sont accessibles grâce à la fonction *getTuileProps(const unsigned int& id)* qui renvoie une structure associée à la tuile d'identifiant id (si elle existe) et composée des différentes propriétés de la tuile : taille d'intervalle de réalisation, offsets droit et gauche de l'intervalle de synchronisation, position absolue dans l'arbre. Ces propriétés étant susceptibles d'être modifiées lors de manipulations de l'arbre, ce mécanisme permet donc par exemple de mettre à jour leur représentations dans une interface graphique.

3.2 Moteur synchrone connecté à la libTuiles pour l'organisation temporelle de processus sonores

Le moteur d'exécution des tuiles asynchrone est couplé à un moteur de traitement/synthèse audio synchrone basé sur le serveur de son Jack. Ce moteur reçoit les commandes, décrites dans la section précédente, de configuration temporelle des processus associés aux tuiles par identifiant.

Les processus peuvent être de deux types. Les processus de lecture audio permettent de lire des fichiers son. Ils gèrent également l'étirement temporel, dû aux changements de tempo de l'arbre et à l'opérateur STRETCH, grâce à l'utilisation de la synthèse granulaire. A la vitesse de lecture initiale, les grains joués se chevauchent à moitié et le pas de déplacement dans le fichier son entre deux grains est égal à une moitié de grain. Lorsque la vitesse de lecture est ralentie ou que la durée des tuiles est allongée, le pas de déplacement est réduit et un offset aléatoire est ajouté afin de prévenir un effet métallique dû à la proximité des grains successifs. De plus le chevauchement entre grains est accentué afin de lisser le son. Lorsque la vitesse de lecture augmente, le pas de déplacement est augmenté et le chevauchement est également accentué afin de réduire les variations d'amplitude entre les grains successifs. Cette méthode de synthèse, malgré les différents artefacts et distorsions qu'elle génère comparativement à d'autres techniques d'étirement temporel, permet d'étirer les sons en temps-réel pour un coût en calcul très faible, ainsi que de se repositionner dans le son sans saut dans le signal.

Il est également possible d'associer des processus de traitement/synthèse sonore aux tuiles, par le biais d'effets FAUST. Il est ensuite possible de rediriger la sortie de n'importe quel processus vers ce traitement FAUST. A chaque fenêtre de rendu audio, le processus FAUST lit les échantillons sur la sortie du processus auquel il est connecté. Les traitements ne sont ainsi effectués que lorsque les

processus se chevauchent temporellement, ce qui est contrôlé en manipulant les fenêtres de synchronisation des tuiles associées.

3.3 Architecture logicielle orientée objet pour la communication entre échelles temporelles

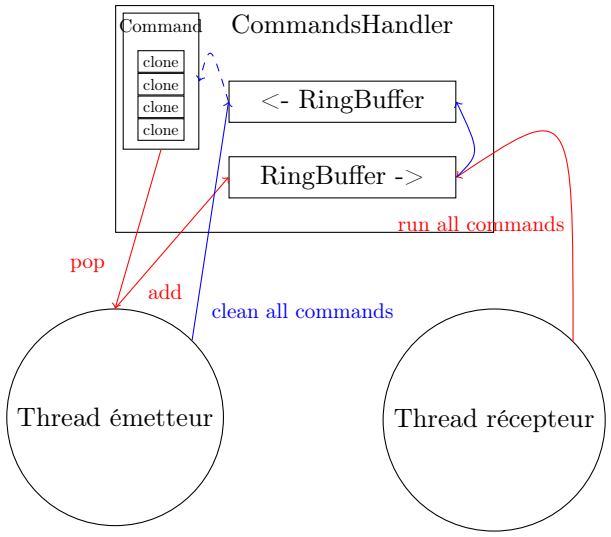


FIGURE 1 – Architecture logicielle objet pour l'échange entre deux threads associés à différentes échelles temporelles

Un aspect important de l'architecture de la libTuiles est l'utilisation de Commandes, représentée sur la Figure 1. Ces composants logiciels permettent de respecter les contraintes dues à la connexion entre les échelles évènementielles, temps-réel asynchrone et temps-réel synchrones, toutes prises en charge par des threads différents. En particulier, le thread temps-réel synchrone, donc géré par le serveur de son JACK, ne doit pas comporter d'allocations/désallocations mémoire ni de mécanismes de verrouillage. L'architecture proposée s'appuie sur plusieurs patrons de conception objet connus parmi lesquels le patron Prototype, le patron Abstract Factory et le patron Commande.

Une classe *CommandsHandler* gère la création et la manipulation d'instances de la classe *Command* ainsi que leur envoi d'un thread émetteur à un thread récepteur. Une instance de cette classe est donc partagée entre les classes gérant les threads. Des associations "chaîne de caractère - Commande" sont tout d'abord ajoutées à cette classe. Par exemple, le *CommandsHandler* du moteur synchrone possède les commandes ActivateProcess et DeactivateProcess, identifiées par leurs noms. Lors de l'ajout, une instance de chaque commande est créée comme prototype avec un certain nombre de clones possédant un pointeur

vers leur prototype. Ainsi des classes associées à chaque message à faire passer d'un thread à l'autre peuvent être définies simplement en les faisant hériter de la classe *Command* et en redéfinissant leur fonction *run()* afin de manipuler une structure de données destinataire dans le thread récepteur, par exemple activer/désactiver un processus.

Lors de l'exécution, le thread émetteur récupère un pointeur vers une instance de la commande désirée en appelant une méthode *popCommand* du *CommandsHandler* avec le nom de la *Commande*. Cette instance est alors retirée des clones disponibles du prototype de cette *Commande* et peut être ajustée avec différents paramètres tels que l'identifiant de la tuile, la nouvelle durée/position ... Aucune allocation mémoire n'est donc effectuée pendant l'exécution. Le pointeur vers cette commande est ensuite redonné au *CommandsHandler* et transite d'un thread à l'autre grâce à un *RingBuffer*, permettant ainsi d'éviter les verrouillages des threads.

Le thread récepteur appelle de son côté la méthode *runCommands* du *CommandsHandler* qui va lire les commandes depuis le *RingBuffer*, appeler leur fonction *run* et les renvoyer au thread émetteur par un deuxième *RingBuffer* en sens inverse. Finalement, le thread émetteur appelle la méthode *cleanCommands* du *CommandsHandler*. Lors de cette méthode, chaque pointeur revenant du thread récepteur est remis dans la liste des clones disponibles du prototype associé.

Cette architecture logicielle , tout en respectant les principes de programmation objet, permet de faire transiter des commandes entre plusieurs threads, sans allocations mémoire ni mécanismes de verrouillage. Elle est donc particulièrement adaptée aux systèmes mêlant différentes échelles temporelles dont certaines très sensibles aux retards comme les threads audio temps-réel.

4 Expérimentation interactive

4.1 Le *SimpleTuilesLooper*

SimpleTuilesLooper est une application permettant de tester la composition temporelle avancée de processus notamment pour la performance live, en s'appuyant sur la *libTuiles* et le moteur synchrone décrits ci-dessus.

Cette application définit comme racine de l'arbre de tuiles une tuile *Loop* dont le premier enfant est une tuile feuille. Toutes les autres tuiles ajoutées à l'arbre sont synchronisées à cette première tuile feuille, dont la fenêtre de synchronisation, dynamiquement modifiable, définit par conséquent la fenêtre de la tuile *Loop* et ainsi la zone de lecture en boucle dans l'arbre.

SimpleTuilesLooper permet de créer des tuiles à partir de fichiers audio et de fichiers dsp FAUST et de combiner ces tuiles pour construire un arbre à l'aide de la métaphore graphique du drag and drop. Les fichiers sont prélevés depuis un explorateur de fichiers et déposés sur la partition. Soit ils sont déposés sans contact avec d'autres tuiles et donc insérés par l'opérateur FORK avec la tuile racine, soit ils sont placés en composition séquentielle ou parallèle avec une tuile

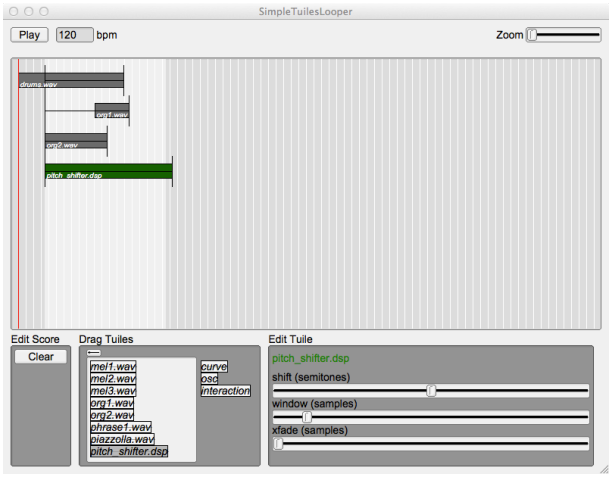


FIGURE 2 – SimpleTuilesLooper permet d'expérimenter la composition temporelle de processus en s'appuyant sur la libTuiles

existante et insérés dans l'arbre avec les opérateurs associés.

Une boîte de dialogue permet de régler les paramètres des effets FAUST ainsi que la connexion des processus entre eux. L'arbre peut ensuite être lu et le tempo de lecture modifié. Un aperçu de l'interface du *SimpleTuilesLooper* est donné sur la figure 2.

4.2 Interaction et placement temporel dynamique

Le *simpleTuilesLooper* permet donc d'exécuter en boucle des expressions de tuiles dont les définitions peuvent être éditées et modifiées de façon interactive, grâce au moteur d'exécution *libTuile*.

De plus, via l'utilisation de buffers d'entrées comme paramètres de transformation de tuiles, on peut facilement enrichir l'interaction offerte par le *simpleTuilesLooper*. Ce faisant, on retrouve la puissance d'expression des systèmes synchrones, tel que Pure data ou Max/Msp, qui sont naturellement paramétrés par de tels flux d'entrée.

Un mécanisme additionnel d'écriture interactive du temps est proposé. Une tuile d'interaction, *ad hoc*, permet d'associer à des évènements *midi* le placement, en séquence ou en parallèle, de signaux tuilés.

5 Conclusion

La *libTuile* et le *simpleTuilesLooper* présentés ici permettent donc d'expérimenter l'écriture dynamique du temps que permet la programmation par tuilage.

Les concepts et les développements associés sont aujourd'hui assez avancés pour une telle expérimentation.

Bien entendu, dans la perspective plus long terme de fournir un moteur d'exécution au *T-calcul* [11] et de produire une version 2.0 du logiciel *i-score* [2] qui intégrera des structures de contrôles plus puissantes, cette présentation n'est qu'un rapport d'étape. Il reste encore à détailler la sémantique opérationnelle du T-calcul [11] pour conduire à une implémentation complète de tout les mécanismes d'écriture du temps qu'il induit.

Références

[1] J. Allen and G. Ferguson. Actions and events in interval temporal logic. In Oliviero Stock, editor, *Spatial and Temporal Reasoning*, pages 205–245. Springer Netherlands, 1997.

[2] A. Allombert, M. Desainte-Catherine, and G. Assayag. Iscore : a system for writing interaction. In *Third International Conference on Digital Interactive Media in Entertainment and Arts (DIMEA 2008)*, pages 360–367. ACM, 2008.

[3] F. Berthaut, M. Desainte-Catherine, and M. Hachet. Drile : an immersive environment for hierarchical live-looping. In *Proceedings of New Interfaces for Musical Expression (NIME10)*, pages 192–197, Sydney, Australia, 2010.

[4] F. Berthaut, D. Janin, and B. Martin. Advanced synchronization of audio or symbolic musical patterns : an algebraic approach. *International Journal of Semantic Computing*, 6(4) :1–19, 2012.

[5] Alessandro Cipriani and Maurizio Giri. *Electronic Music and Sound Design - Theory and Practice with Max/Msp.* Contemponet, 2010.

[6] S. Letz et al. The LibAudioStream library, 2012. `http://libaudiostream.sourceforge.net/`.

[7] D. Fober, Y. Orlarey, and S. Letz. FAUST architectures design and OSC support. In *14th Int. Conference on Digital Audio Effects (DAFx-11)*, pages 231–216. IRCAM, 2011.

[8] D. Janin. Vers une modélisation combinatoire des structures rythmiques simples de la musique. *Revue Francophone d'Informatique Musicale (RFIM)*, 2, 2012.

[9] D. Janin. Algebras, automata and logic for languages of labeled birooted trees. Technical Report RR-1467-13, LaBRI, Université de Bordeaux, 2013.

[10] D. Janin. Overlaping tile automata. In *8th International Computer Science Symposium in Russia (CSR)*, LNCS (to appear). Springer-Verlag, 2013.

[11] D. Janin, F. Berthaut, M. DeSainte-Catherine, Y. Orlarey, and S. Salvati. The T-calculus : towards a structured programming of (musical) time and space. Technical Report RR-1466-13, LaBRI, Université de Bordeaux, 2013.

[12] M. V. Lawson. *Inverse Semigroups : The theory of partial symmetries.* World Scientific, 1998.

[13] F. Lerdahl and R. Jackendoff. *A generative theory of tonal music*. MIT Press series on cognitive theory and mental representation. MIT Press, 1983.

[14] S. Wilson. *The SuperCollider Book*. Cambridge : The MIT Press, 2011.