INTEGRATION DU COMPILATEUR FAUST DANS L'APPLICATION DE PATCHING COLLABORATIF KIWI

Pierre Guillot
CICM - EA 1572,
Université Paris 8 et Ircam
guillotpierre6@gmail.com

Eliott Paris
CICM - EA 1572,
Université Paris 8
eliottparis@gmail.com

Alain Bonardi
CICM - EA 1572,
Université Paris 8 et Ircam
alain.bonardi@univ-paris8.fr

RÉSUMÉ

Cet article revient sur l'intégration du compilateur à la volée Faust au sein de l'application de patching collaboratif Kiwi. Nous évoquons dans un premier temps le contexte et les enjeux de ce développement. Puis, nous détaillons les choix de mise en œuvre nous ayant amené à l'élaboration de l'objet faust~, permettant d'écrire et d'exécuter du code Faust de manière collaborative au sein de l'application Kiwi, en nous intéressant particulièrement à l'interface et aux possibilités offertes aux utilisateurs. Enfin, nous ouvrons sur les perspectives envisagées, notamment en matière de création musicale et de pédagogie.

1. INTRODUCTION

L'application Kiwi[1] est environnement de programmation graphique modulaire dédié au traitement et à la synthèse audio dans la lignée des logiciels tels que Max[2] ou Pure Data . Sa principale originalité est de permettre à plusieurs personnes d'éditer et de contrôler un même patch à distance à travers un réseau auquel ils sont connectés [1], [2]. Développé par le CICM dans le cadre du projet ANR MUSICOLL, cet environnement a été utilisé à la fois sur le terrain de la création musicale et de la pédagogie liée à l'enseignement du temps réel au collège et à l'université [3], [4]. De l'autre côté, Faust est un langage de programmation fonctionnel dédié à la synthèse et au traitement temps réel du signal audionumérique développé par le laboratoire de recherche du GRAME [5]. Au-delà du langage, Faust offre aussi un ensemble d'outils permettant de générer du code C++ mais aussi du code machine via le compilateur à la volée (*just-in-time compiler*) LLVM permettant d'exécuter du code Faust au sein d'applications tierces [6].

L'intégration du compilateur Faust au sein de l'application Kiwi présente alors un double intérêt. L'enjeu principal à l'origine de ce développement est de pourvoir le logiciel Kiwi des nombreuses possibilités offertes par ce langage. En effet, les développements actuels de Kiwi n'ont pas encore pu aboutir à une suite suffisante d'objets pour des pratiques avancées du patching et cela pour des raisons à la fois de temps, qui sont aussi intrinsèques au modèle même de l'application et de son approche du collaboratif. De plus, l'application Kiwi ne comporte pas pour l'instant d'interface de programmation applicative (API) permettant aux utilisateurs d'ajouter dynamiquement de nouveaux objets au logiciel, et donc de nouvelles fonctionnalités ou traitements sonores au langage comme on pourrait le faire avec les logiciels Max ou Pure Data. Aussi, intégrer le compilateur Faust permet donc de tirer parti des différents modules de traitement du signal et notamment de ses filtres, déjà présents au sein de ses bibliothèques standards. Mais au-delà du cas spécifique de l'application Kiwi, l'intégration du compilateur Faust au sein d'un logiciel de patching tel que Max ou Pure Data offre un bénéfice certain "*car les opérations nécessitant de travailler au niveau de l'échantillon, par exemple les filtres IIR, ne peuvent pas directement être mises en œuvre de manière efficace dans ces langages, et doivent être fournies en tant que primitives ou en tant que plugins externes. L'avantage d'un langage entièrement compilé comme Faust est qu'il peut être utilisé pour implémenter des algorithmes efficaces DSP au niveau de l'échantillon*" [7].

Réciproquement, il y a fort à penser que, par l'aspect collaboratif de l'application de Kiwi, l'intégration du compilateur Faust permette d'ouvrir de nouvelles perspectives quant à la manière de coder en ce langage et cela particulièrement dans un cadre pédagogique.

Avant de présenter l'outil final et les possibilités offertes à l'utilisateur, nous revenons sur les différentes perspectives de développement qui ont été envisagées et sur les principaux projets antérieurs qui ont inspiré l'approche.

2. PREMIERES APPROCHES

Originellement, un code en langage de programmation Faust nécessite d'être compilé afin de générer entre autres des bibliothèques de traitement du signal, des *plugins* ou des applications autonomes [5]. Aussi le langage s'accompagne d'un ensemble d'outils, tels qu'un compilateur mais aussi l'environnement de développement intégré FaustWork ou la version hébergée en ligne Faust Online Compiler [8], permettant de déployer du code directement pour un ensemble de cibles logicielles.

[1]L'application Kiwi est disponible gratuitement pour les systèmes d'exploitation Linux, MacOS et Windows sur le site : kiwi.mshparisnord.fr et son code source est sous licence GPLv3 et disponible sur le répertoire Github : github.com/Musicoll/Kiwi (sites consultés en janvier 2019).

[2]Le logiciel Max, originellement développé à l'IRCAM par Miller Puckette est aujourd'hui développé et distribué par la société Cycling'74 : cycling74.com (site consulté en janvier 2019).

Il est par exemple possible de transformer du code Faust en *plugins* VST[3] ou encore en bibliothèque dynamique à destination des environnements de patching Max ou Pure Data [9].

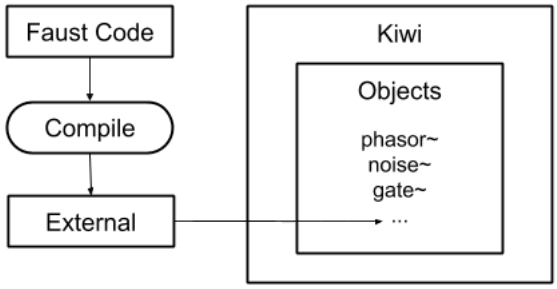

Figure 1. Schéma représentant la première approche envisagée pour utiliser du code Faust au sein de l'application Kiwi. Le code Faust est compilé en un objet externe Kiwi qui est intégré aux objets natifs de l'application.

Une première possibilité d'intégration de Faust au sein de Kiwi aurait pu être de créer une nouvelle cible spécifique. L'objectif aurait été de mettre en œuvre une communication entre la classe *dsp* Faust en C++ et l'application Kiwi afin qu'un code Faust puisse générer une bibliothèque destinée à être chargée en tant qu'objet à part entière au sein de celle-ci (Figure 1), [10]. Cependant, cette première solution n'a pas été retenue dans la mesure où le développement de Kiwi n'est pas encore assez mature pour que l'on puisse définir une interface de programmation qui soit assez stable pour rendre ce développement pérenne[4].

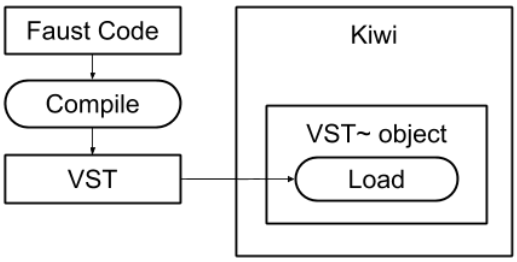

Figure 2. Schéma représentant la deuxième approche envisagée pour utiliser du code Faust au sein de l'application Kiwi. Le code Faust est compilé en un *plugin* VST qui est chargé dynamiquement dans Kiwi par un objet natif *VST~*.

Il a alors été envisagé d'utiliser une architecture logicielle extérieure standardisée, en l'occurrence le format VST, afin de contourner le problème. Le développement a consisté à créer un objet intégré dans l'application permettant de charger dynamiquement des *plugins* au format VST (Figure 2). Cette approche est fonctionnelle, il suffit alors d'utiliser les outils logiciels Faust pour générer un *plugin* à partir du code puis de charger ce *plugin* via l'objet, mais elle révèle un problème plus générique lié à l'approche collaborative. En effet, l'ajout de fonctionnalités via des objets externes, notamment sous forme de *plugins*, est complexe à gérer dans un système collaboratif car cela implique que le rendu sonore d'un patch chargé au sein de l'application par plusieurs utilisateurs puisse différer d'un utilisateur à un autre. Il est, en effet, possible des utilisateurs ne possèdent pas un *plugin* chargé par un autre utilisateur, ou alors qu'il soit dans une version différente. Aucune solution ne nous semblait réellement viable.

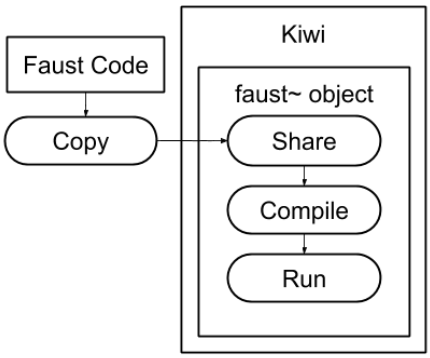

Figure 3. Schéma représentant l'approche finale pour utiliser du code Faust au sein de l'application Kiwi. Le code Faust est chargé et compilé dynamiquement dans Kiwi par un objet natif *faust~*.

Comme évoqués précédemment, les outils de compilation Faust sont aussi disponibles sous la forme d'une bibliothèque *libfaust*[5] qui peut être directement embarquée au sein d'autres applications. Construite autour du compilateur à la volée LLVM, cette bibliothèque permet, au sein des applications, de transformer du code textuel en code machine exécutable sans faire appel à des dépendances externes. Plusieurs projets existants s'appuient aujourd'hui sur cette technologie. On pourra citer l'objet *faustgen~* pour Max, développé par le GRAME [9], l'intégration au langage *Pure* d'Albert Gräf et son interface pour Pure Data [11], la bibliothèque de *plugins* audio iPlug 2 [12], ou encore l'application de compilation à la volée FaustLive [6], elle aussi développée par le GRAME. Comme cela est présenté par la suite, cette approche offre de nombreux avantages et s'est révélée adaptée à un contexte collaboratif, c'est pourquoi elle a été retenue (Figure 3). À titre indicatif, il est intéressant de citer l'objet *faustgen~*[6],

[3]Les formats de *plugins* audionumériques VST (Virtual Studio Technology) 2 et 3 sont développés par la société Steinberg www.steinberg.net (page consultée en janvier 2018).

[4]Il semblait encore préférable de limiter autant que possible l'exposition et l'utilisation de certaines parties maîtresses du code de Kiwi dans des applications tierces. Ceci afin d'éviter des complications ou des pertes de compatibilité liées à des modifications souvent fréquentes dans notre approche prospective.

[5]www.grame.fr/logiciels/libfaust (site consulté en janvier 2019)

[6]L'objet *faustgen~* pour Pure Data est libre et gratuit et disponible pour les systèmes d'exploitation Linux, MacOS et Windows sur le répertoire Github github.com/CICM/pd-faustgen (site consulté en janvier 2019) et via le système de gestion de bibliothèques externes Deken intégré à Pure Data.

équivalent Pure Data de l'objet homonyme pour Max, qui a servi de terrain propice aux premiers développements de cette approche.

3. PRESENTATION GENERALE DE L'OBJET

Inspiré par les précédents travaux analogues et suite aux premières maquettes et expérimentations, l'intégration définitive du compilateur Faust au sein de l'application Kiwi se trouve sous la forme de l'objet standard de traitement du signal : *faust~*.

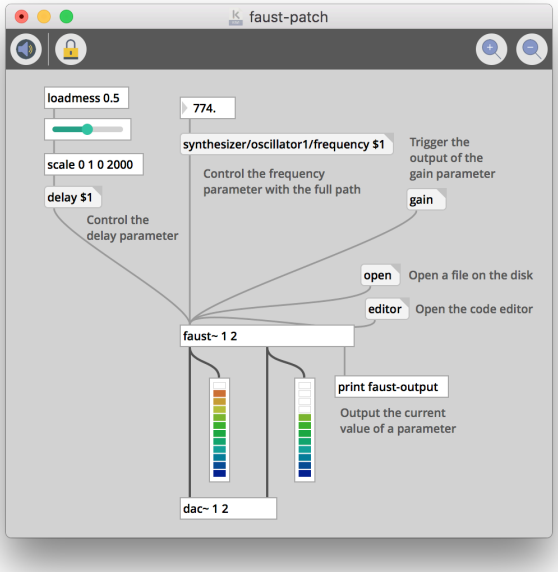

Figure 4. Patch Kiwi utilisant l'objet *faust~*. Les différents objets contenus dans le patch servent à contrôler l'objet ou les paramètres du traitement.

L'objet requiert deux arguments définissant les nombres maximaux d'entrées et de sorties de type signal supportés. Lorsque l'audio est allumé, l'objet envoie directement les signaux audio associés à l'instance Faust de traitement du signal pour que celle-ci lise les échantillons reçus en entrée et écrive les résultats du traitement défini par le code dans les signaux de sortie[7]. Notons qu'une sortie de type contrôle supplémentaire est toujours présente afin de récupérer les valeurs courantes, notamment des paramètres passifs.

Le langage Faust offre deux interfaces principales pour contrôler et/ou représenter certaines variables au sein du moteur audio. Ces interfaces, ou paramètres, sont de deux types : actifs et passifs. Les paramètres actifs sont les paramètres qui viennent modifier l'état du moteur audio. Il est possible de les contrôler avec des messages de type « *clé - valeur* », comme « *frequency 220* » par exemple, en utilisant pour la clé soit leurs noms simples soit leurs noms complets, ou chemin ; c'est-à-dire prenant en compte les groupes et sous-groupes auxquels ils appartiennent, tel que « *synthesizer/oscillator1/frequency* » (Figure 4). Une attention particulière a été portée à la gestion des paramètres lorsque le code Faust est recompilé. Si la définition d'un paramètre dans le code Faust n'a strictement pas changé[8] alors, l'objet préserve la valeur courante du paramètre, diminuant ainsi les variations normalement dues à la réinitialisation du moteur. Ce comportement semble particulièrement utile et apprécié durant la phase d'édition du code. Les paramètres passifs exposent des valeurs du moteur Faust, comme la valeur courante du signal. Lorsque l'objet reçoit le nom d'un paramètre passif, sa valeur courante est alors envoyée par la dernière sortie assurant ainsi la synchronisation des messages sur le bon fil d'exécution et à la fréquence définie par l'utilisateur.

Enfin, toujours dans l'optique de diminuer les variations sonores liées à la recompilation du traitement Faust, la mise en œuvre de la compilation du code est sans verrous (*lock-free*) et le changement effectif de l'ancien code vers le nouveau est quasi-atomique. En pratique, lorsque le code est modifié, le moteur audio du précédent code continue de tourner sans interruption et ce n'est que lorsque le nouveau moteur audio du nouveau code est prêt que l'objet remplace l'ancien moteur audio. Aussi, la transaction est très rapide et ne génère presque pas d'artefact - toutes proportions gardées au regard de la similitude des traitements mis en œuvre entre les deux codes.

Les caractéristiques génériques du fonctionnement de l'objet qui viennent d'être présentées montrent qu'une attention toute particulière a été portée pour offrir un outil limitant au maximum les artefacts liés à la compilation du code Faust. Cette propriété prend tout son sens lors des pratiques collaboratives.

4. EDITION COLLABORATIVE DU CODE

Dans un contexte où plusieurs personnes sont connectées à un même patch dans Kiwi, les utilisateurs bénéficient tous du même code mais aussi des mêmes options de compilation, telles que les optimisations utilisées par le compilateur Faust. La seule option qui n'est pas partagée, et qui n'est pas non plus modifiable par l'utilisateur, car elle dépend du système d'exploitation, est le chemin vers les bibliothèques standard de Faust qui sont embarquées avec l'application Kiwi. Cela permet d'assurer que, pour une version donnée de Kiwi, tous les utilisateurs possèdent les mêmes bibliothèques et qu'un code fonctionnant sur une machine fonctionne de la même manière sur toutes les autres.

Le code Faust du moteur audio est partagé entre les utilisateurs d'un même patch. Il est intégré et sauvegardé au sein du document et ne nécessite donc pas d'un

[7]Le modèle de données utilisé par la mise en œuvre des fonctionnalités collaboratives au sein du logiciel empêche la modification a posteriori du nombre d'entrées et de sorties. Aussi, si le moteur audio Faust nécessite un plus grand nombre de signaux, que ce soit en sortie ou en entrée, alors l'objet détourne le traitement et un message d'avertissement est envoyé à l'utilisateur via la console de Kiwi. Ce fonctionnement assure que l'instance Faust n'essaye pas de lire ou d'écrire dans des signaux non-existant.

[8]Si le nom, le chemin, les valeurs minimales et maximales d'un paramètre ne diffèrent pas entre deux compilations.

référencement vers un fichier externe pour fonctionner (Figure 3). Pour initialiser le code Faust de l'objet, une première possibilité est de le faire en chargeant un code préexistant grâce au message « open » de l'objet qui permet de sélectionner un fichier portant l'extension *.dsp* (Figure 4). Dans ce cas, le contenu textuel du fichier est sauvegardé au sein du modèle de document du patch et synchronisé avec les autres utilisateurs connectés. Ce nouveau code Faust est alors compilé sur la machine de chaque utilisateur. Si, pour une raison ou une autre, le code génère une erreur, il est quand même compilé mais ne produit pas de son. Ce choix a été réalisé afin d'assurer une cohérence du rendu sonore chez les différents utilisateurs. La seconde manière de modifier le code Faust est de le faire via l'éditeur de texte intégré à l'objet.

Cet éditeur permet de modifier le rendu sonore de l'objet *faust~* par l'édition du code Faust à partir de l'interface de Kiwi. Le code édité est aussi partagé et mis à jour au sein du modèle de données permettant l'édition collaborative de son contenu à travers cette interface. L'éditeur de code de l'objet *faust~* dans Kiwi, par rapport à celui présent au sein de l'objet *faustgen~* de Max, comporte plusieurs améliorations fonctionnelles. Il gère notamment la coloration syntaxique du code spécifique au langage Faust[9], et permet l'affichage des erreurs directement sur l'interface (Figure 5). À chaque fois que le texte est modifié au sein de l'éditeur, en tâche de fond, l'objet recompile le code Faust à la volée de façon à pouvoir présenter les erreurs d'édition éventuelles à l'utilisateur. Lorsque l'utilisateur qui édite le texte clique sur le bouton « *Synchronize Audio* », le code visualisé est transmis au moteur audio qui actualise alors le rendu sonore de l'objet. Lors d'une session connectée, tous les participants peuvent donc mettre à jour le code d'un objet *faust~* et synchroniser ainsi le rendu sonore de l'objet sur toutes les machines connectées. Si un utilisateur ouvre l'éditeur alors qu'un autre est en train d'éditer son contenu, il peut voir s'afficher en temps réel les modifications apportées au texte dans la mesure où le code est transmis par l'intermédiaire du modèle de données à tous les participants. Cependant, l'édition collaborative de texte en temps réel à proprement parler, à savoir le fait que plusieurs personnes puissent simultanément écrire du code au sein de l'éditeur, n'est pas possible car la bibliothèque *flip*[10] qui soutient le modèle de donnée de l'application n'a pas été conçue initialement pour gérer ce cas. Même si un développement allant dans ce sens avait été évoqué et envisagé dans le cadre du projet MUSICOLL, il n'a pas encore été effectué à l'heure actuelle. De plus, l'écriture collaborative du texte en temps réel pose aussi un certain nombre de problématiques d'ordre plus ergonomiques, liées à l'interface graphique, qui sont gérées dans certaines bibliothèques spécialisées [13], mais qui ne le sont pas dans la plupart des bibliothèques applicatives orientées mono-utilisateur, telles que Juce. Permettre l'édition collaborative de texte aurait donc nécessité soit d'intégrer une bibliothèque tierce qui puisse le gérer, soit de recréer tout un système, ce qui n'était alors pas envisageable au vu de la complexité de la tâche et du temps imparti. Comme nous ne disposions plus de ressources suffisantes à dédier à cette tâche vers la fin du projet, il nous a donc fallu trouver une approche alternative.

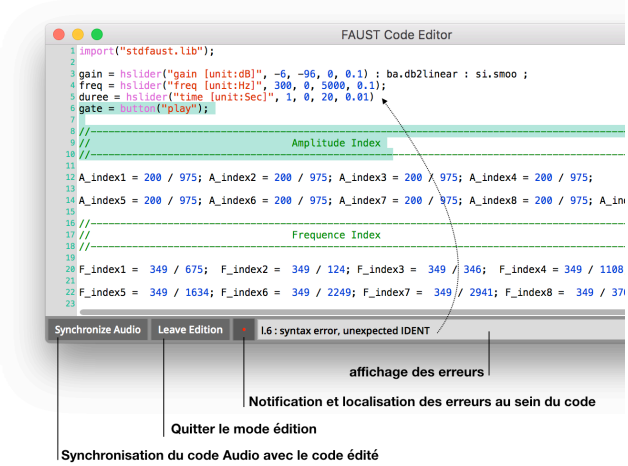

Figure 5. Interface graphique de l'éditeur de code Faust embarquée au sein de l'objet faust~ de Kiwi. La partie de code exposée dans cette capture provient d'une mise en œuvre en langage Faust du modèle de cloches de Risset, réalisée par le compositeur João Svidzinski.

Il existe de nombreuses manières de gérer l'édition concurrente d'un document [14]. Celle que nous avons actuellement mise en place peut être qualifiée de *semi-pessimiste* : le premier utilisateur qui démarre l'édition du document et lui-seul, dispose des droits en écriture, et ce jusqu'à ce qu'il laisse la main[11]. Cette approche nous a permis de mettre en œuvre rapidement un système qui fonctionne. Néanmoins, celle-ci n'est généralement pas idéale dans la mesure où elle pose de nombreux problèmes pratiques. Une personne qui a la main sur le contenu peut empêcher les autres d'y accéder. Si la personne garde la main trop longtemps, s'absente ou encore oublie de libérer l'accès, les autres utilisateurs sont bloqués et ne peuvent pas modifier librement son contenu. Pour contourner ces problèmes nous avons donc décidé de mettre en place une approche plus flexible permettant à la fois aux utilisateurs d'observer les modifications apportées à un document par un autre, mais aussi de pouvoir prendre la main s'ils le souhaitent sur l'édition. Ce mode ne lui permet pas d'éditer le texte, en revanche, il peut suivre son édition en temps réel dans la mesure où le texte tapé est transmis à tous les utilisateurs connectés à la session.

[9]La mise en place de la coloration syntaxique repose sur un code d'Oliver Larkin github.com/olilarkin/juce_faustllvm (site consulté en janvier 2019).

[10]*Flip* est une bibliothèque de modèle de données permettant de concevoir des applications collaboratives. Cette bibliothèque a été développée par la société Irisate, partenaire du projet MUSICOLL, irisate.com/flip-overview (site consulté en janvier 2019).

[11]L'approche pessimiste s'oppose à une approche qualifiée d'optimiste dans laquelle le système n'a pas besoin d'être verrouillé pour laisser les utilisateurs lire ou éditer les données.

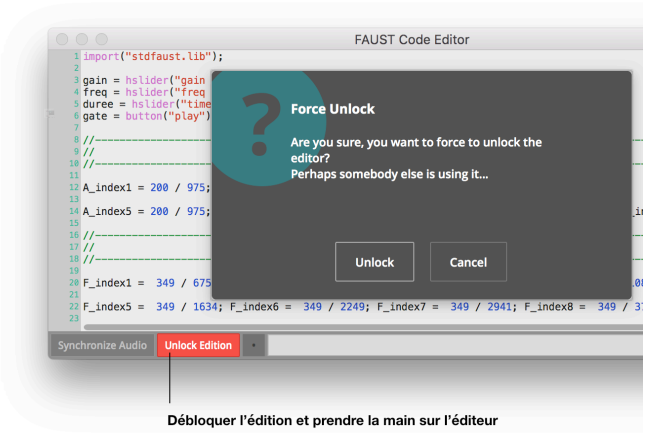

Figure 6. Interface graphique de l'éditeur de code de l'objet *faust~*, tel qu'il apparaît à un utilisateur lorsque quelqu'un d'autre est en train d'éditer le texte. Le bouton « *Unlock Edition* » permet de prendre la main sur l'édition.

La manière dont a été conçue l'interface permet aussi à l'utilisateur de prendre la main de manière forcée sur le code en appuyant sur le bouton « *Unlock Edition* » (Figure 6). Une fenêtre contextuelle s'affiche alors pour l'informer que cette action va avoir pour conséquence de faire perdre la main à l'éditeur courant à son profit. Si l'utilisateur accepte, l'éditeur est déverrouillé de son côté et se retrouve dans la configuration présentée plus haut (Figure 5), laissant alors à l'utilisateur la possibilité d'apporter les modifications qu'il souhaite au code.

5. PERSPECTIVES D'UTILISATION

Nous envisageons dans l'immédiat deux types d'utilisation du compilateur Faust dans des situations collaboratives, d'une part pédagogiques, d'autre part dans le domaine de la documentation des œuvres mixtes.

L'objet faust~ est désormais disponible au sein de la dernière version de l'application Kiwi permettant de tirer parti des nombreuses fonctions offertes par ce langage fonctionnel dédié au traitement du signal audionumérique. Ce développement offre réciproquement un nouvel outil flexible pour exécuter du code Faust, car possédant les avantages des logiciels de patching, mais l'objet *faust~* est surtout un nouvel outil pour créer et éditer du code Faust et cela de manière collaborative. Il est, grâce à cela, certain que Kiwi peut devenir un outil pédagogique essentiel pour l'apprentissage de Faust. À ce propos, Julius Smith propose dans son enseignement de Master au CCRMA[12] sur le traitement de signal deux approches utilisant le langage Faust et dans lesquelles Kiwi serait fort apprécié [15]. Lors des démonstrations en temps réel sur le traitement du signal audio, l'approche collaborative offre la possibilité aux étudiants de dépasser le rôle du spectateur-auditeur et de prendre part au jeu en contrôlant les paramètres en temps-réel, action qui était, pour des raisons techniques, jusqu'à présent réservées à l'enseignant seul. De même, l'engagement de l'étudiant lors de sessions de programmation en direct peut être renforcé par l'aspect collaboratif car il a la possibilité de prendre la main sur l'édition de code afin de soumettre des propositions ou de mettre en avant des parties qui lui semblent confuses ; cela en temps réel et de façon partagée sur l'ensemble des machines connectées. En somme, un dialogue plus direct peut donc se construire entre l'enseignant et les étudiants. Dans ce cadre, la coloration syntaxique et la notification des erreurs de l'éditeur de code de Kiwi sont des fonctionnalités essentielles auxquelles il serait par ailleurs intéressant de rajouter une représentation en blocs-diagramme afin de faciliter le débogage et la compréhension du programme par les étudiants.

Concrètement, l'application Kiwi sera donc utilisée sur six séances dans le cadre du cours intitulé *Langages de programmation en informatique musicale 2* à l'Université Paris 8, dispensé par Alain Bonardi au second semestre 2018-2019, pour renouveler l'enseignement de Faust. Nous espérons donc que l'objet pourra ainsi bénéficier aux étudiants et plus généralement à tous les utilisateurs de Kiwi en leur permettant de se former au langage et de l'exploiter à des fins musicales, dans la même optique d'apprentissage par le faire-ensemble qui est propre à l'ensemble de l'application. Enfin, ce développement spécifique permet encore d'ouvrir de nouvelles pistes de recherche à explorer, notamment pour permettre d'offrir une collaboration synchrone du code au sein de l'éditeur.[13]

Pendant l'année 2019, le CICM va mener un travail de recherche sur *Inharmonique* pour voix et bande de Jean-Claude Risset, dans le cadre du projet « Créer, (re)créer, valoriser la musique faisant appel à l'informatique au XXIe siècle » porté à la Maison des Sciences de l'Homme Paris Nord par João Svidzinski en collaboration avec Antonio de Sousa Dias de la Faculté des Beaux-Arts de l'Université de Lisbonne. Partant d'un ensemble d'éléments matériels et de diverses sources dont l'analyse de Denis Lorrain [16], il s'agit d'aboutir à une version interactive temps réel de l'œuvre. Ce projet contribuera également aux travaux du groupe travail AFIM « Archivage collaboratif et préservation créative » (2018-2019).

Le choix du langage Faust dans la documentation des œuvres avec électronique a déjà fait ses preuves par exemple dans le cadre du projet ANR ASTREE au cours duquel il a permis de proposer de nouvelles versions de *Turenas* de John Chowning [17] ou de *En Echo* de Philippe Manoury [18]. Dans le cadre du portail Analyses de l'Ircam, le langage Faust va également être utilisé pour permettre la manipulation des modules utilisés par les compositeurs en WebAudio [19].

[12]Center for Computer Research in Music and Acoustics, département de Musique de l'Université de Stanford en Californie aux États-Unis.

[13]Notons que cet enjeu se retrouve plus généralement dans le fait de permettre à plusieurs utilisateurs d'éditer un texte de manière collaborative dans Kiwi, comme dans un objet commentaire au sein d'un patch.

Il est donc logique de penser à Faust pour écrire la nouvelle version temps réel de l'électronique de *Inharmonique* de Risset. Comme le travail associera plusieurs chercheurs, l'utilisation de Kiwi en édition collaborative des codes Faust intervenant dans l'œuvre est une piste prometteuse.

6. REMERCIEMENTS

La majeure partie de la recherche exposée dans cet article est financée dans le cadre de la convention attributive d'aide de l'Agence Nationale de la Recherche n°ANR-15-CE38-0006-01.

7. REFERENCES

[1] Paris E., Millot J., Guillot P., Bonardi A. et Sèdes A., "Kiwi : vers un environnement de création musicale temps réel collaboratif premiers livrables du projet Musicoll", Actes des Journées d'Informatique Musicale 2017 (JIM 2017), Paris, France, 2017. <hal-01550190>

[2] Paris E., "Une approche du patching audio collaboratif : enjeux et développement du collecticiel Kiwi", Thèse de doctorat en Esthétique, Sciences et Technologie des Arts, sous la direction d'Anne Sèdes et Alain Bonardi, Université Paris 8, Saint-Denis, France, 2018. <hal-01550190>

[3] Sèdes A., Bonardi A., Paris E., Millot J. et Guillot P., "Teaching, investigating, creating: MUSICOLL", *Innovative Tools and Methods for Teaching Music and Signal Processing*, 2017, 978-2- 35671-444-2. <hal-01581698>

[4] Galleron P., Maestri E., Millot J., Bonardi A. et Paris E., "Enseigner le patching de manière collective avec le logiciel collaboratif Kiwi", Actes des Journées d'Informatique Musicale 2018 (JIM 2018), Amiens, France, 2018. <hal-01791492>

[5] Orlarey Y., Fober D., et Letz S., "FAUST - an Efficient Functional Approach to DSP Programming", *New Computational Paradigms for Computer Music*, Editions Ircam/Delatour, Paris, France, 2009.

[6] Denoux S., Letz S., Orlarey Y. et Fober D., "FAUSTLIVE Just-In-Time Faust Compiler... and much more", In Proceedings of the Linux Audio Conference (LAC 2013), Graz, Autriche, 2013. <hal-00965266>

[7] Orlarey Y., Letz S. et Fober D., "Des outils pour enseigner Faust", RFIM - Revue Francophone d'Informatique et Musique [En ligne], n° 6 - Techniques et méthodes innovantes pour l'enseignement de la musique et du traitement de signal, 2018, http://revues.mshparisnord.org/rfim/index.php?id=566 (site consulté en décembre 2018).

[8] Michon R. Et Orlarey Y., "The Faust Online Compiler: a Web-Based IDE for the Faust Programming Language", In Proceedings of the Linux Audio Conference (LAC 2012), Stanford University, Californie, États-Unis, 2012.

[9] Letz S., Fober D. et Orlarey Y., "Comment embarquer le compilateur Faust dans vos applications ?", Actes des Journées d'informatique Musicale 2013 (JIM 2013), Paris, France, 2013. <hal-00832224>

[10] Fober D., Orlarey Y. et Letz S., "Faust architecture design and OSC support." In Proceedings of the Conference on Digital Audio Effects (DAFx 2011), Paris, France, 2011.

[11] Gräf A., "Functional Signal Processing with Pure and Faust using the LLVM Toolkit", In Proceedings of the Sound and Music Computing Conference, Padova, Italie, 2011.

[12] Larkin O., "Faust in iPlug 2 : Creative coding audio plug-ins", In Proceedings of the International Faust Conference (IFC 2018), Mainz, Allemagne, 2018.

[13] Rosemann, M. et Greenberg, S. "Building Real Time Groupware with GroupKit, A Groupware Toolkit", ACM Transactions on Computer Human Interaction - ACM TOCHI, ACM Press, New York, États-Unis, pp. 66-106, 1996.

[14] Ellis C. A. et Gibbs S. J. "Concurrency Control in Groupware Systems", In Proceedings of the ACM SIGMOD Conference on the Management of Data (SIGMOD 1889), Portland, Oregon, États-Unis, 1989.

[15] Smith J., "Faust dans une salle de classe : démonstration, live, coding (programmation en direct) et implémentations de référence", RFIM - Revue Francophone d'Informatique et Musique [En ligne], Numéros, n° 6 - Techniques et méthodes innovantes pour l'enseignement de la musique et du traitement de signal, 2018, http://revues.mshparisnord.org/rfim/index.php?id=550 (site consulté en décembre 2018).

[16] Lorrain D. "Analyse de la bande magnétique de l'œuvre de Jean-Claude Risset, Inharmonique", Rapport Ircam 26/80, 1980, http://articles.ircam.fr/textes/Lorrain80a (site consulté le 25 janvier 2019)

[17] Pottier L., "La "régénération" des sons de *Turenas* de John Chowning", *Préserver - Archiver - Reproduire : musique et technologie, jeux vidéo,* direction Evelyne Gayou, Portraits polychromes, Hors-série thématique n°21, INA-GRM, Paris, 2013.

[18] Bonardi A., "Approches pratiques de la préservation/virtualisation des œuvres interactives mixtes : *En Echo* de Manoury," Actes des Journées d'Informatique Musicale (JIM 2011), Saint-Etienne, France, 2011. <hal-00656887>

[19] Bonardi A., Pellerin G. et Zawadzki E., "Un exemple de convergence de l'analyse et de l'informatique musicales. Enrichir le projet Analyses par le Web Audio.", Actes des Journées d'Informatique Musicale (JIM 2017), Paris, France, 2017. <hal-01582653>