JAMOMA MODULAR : UNE LIBRAIRIE C++ DEDIEE AU DEVELOPPEMENT D'APPLICATIONS MODULAIRES POUR LA CREATION

Théo De La Hogue — GMEA / Jamoma — theod@gmea.net

Julien Rabin — GMEA / Jamoma — julien.rabin@jamoma.org

Laurent Garnier — Galamus — lgarnier@galamus-software.fr

RÉSUMÉ

Parmi les différentes librairies constituant la plateforme Jamoma, Jamoma Modular présentée dans cet article, offre un ensemble de solutions pour la représentation, l'observation ainsi que l'exploration des fonctionnalités d'une application sous la forme d'une arborescence, pour l'échange de données de manière locale ou au sein d'un réseau ou encore pour la manipulation de *presets*, d'automation ou de *mappings*. Écrite dans le langage C++, Jamoma Modular est aujourd'hui utilisée pour la réalisation de différentes applications dont l'implémentation sous la forme d'*externals* et *patchs* Max/MSP est un exemple abouti.

1. INTRODUCTION

Dans le cadre des développements informatiques appliqués à la création musicale, ceux visant à la constitution d'outils pour le spectacle vivant peuvent se distinguer de façon singulière : chaque projet artistique ou musical porte en effet un ensemble de nécessités conduisant à un cahier des charges qui lui est propre. Il appartient alors à la personne responsable des développements informatiques de mettre en œuvre un environnement logiciel unique et répondant aux besoins spécifiques de ce projet.

Pour autant, l'observation des pratiques liées à ces dispositifs informatiques ainsi que l'étude des dispositifs même permettent, du point de vue des développements informatiques, de nuancer cette affirmation. Dans le cadre de cet article, on dégagera deux constats qui fondent les travaux présentés ici.

En premier lieu, la conception d'un logiciel dédié à un projet artistique ou musical peut néanmoins souvent faire appel à un ensemble de fonctionnalités générales et récurrentes : les sauvegarde et lecture de mémoires (*presets, cues*) décrivant l'état du dispositif informatique à un instant *t*, le *mapping* de valeurs issues d'un système de captation aux paramètres d'un module de synthèse par exemple sont ainsi autant de fonctionnalités génériques auxquelles le développeur peut avoir recours.

En second lieu, l'environnement informatique employé pour l'exécution de l'œuvre peut résulter de la mise en communication de plusieurs logiciels spécialisés : traitement audionumérique (échantillonneur virtuel, moteur de synthèse en temps réel, etc.), captation gestuelle (*tracking* vidéo, suiveur de geste, etc.), traitement vidéo (synthèse d'images 3D, etc.) par exemple. De plus, la mise en réseau de diverses applications est rendue complexe lorsque ce dernier est ouvert et constitué de manière dynamique. Dès lors, un ensemble d'outils génériques et répondant à des critères de modularité peut se révéler très utile pour la personne en charge du développement de ce dispositif informatique.

Le projet Jamoma offre aujourd'hui aux développeurs d'applications multimédias un ensemble de librairies C++ (*frameworks*) dédiées par exemple à la génération et au traitement de signal, à la réalisation de *graph* synchrones et asynchrones. L'une d'entre elles, Jamoma Modular dont la refonte intégrale effectuée au GMEA – Centre National de Création Musicale d'Albi-Tarn marque le passage à sa version 0.6, offre un ensemble de solutions pour la représentation des fonctionnalités d'une application sous la forme d'une arborescence et sa manipulation localement ou à distance ou encore pour la mise en œuvre de services génériques tels que la gestion de *presets* ou de *mappings*. S'il existe aujourd'hui plusieurs librairies dédiées visant à répondre à l'un de ces aspects, Jamoma Modular propose en revanche un cadre de développement intégrant ces différentes fonctionnalités au sein d'une architecture unifiée et de bas niveau.

Après une rapide présentation du projet Jamoma et de ses objectifs spécifiques, cet article présentera la librairie Jamoma Modular et des solutions qu'elle peut offrir aux développeurs. Enfin, seront donnés deux exemples de projets implémentant certaines de ses fonctionnalités.

2. UNE PLATEFORME POUR LA RECHERCHE ET LA CREATION

Depuis 2005, le projet Jamoma se définit comme une « plateforme interactive pour la recherche et la performance artistiques »[1]. Projet international et *open source* (publié sous licence BSD[2]), il est actuellement développé par Timothy Place (Cycling 74, États-Unis), Trond Lossius (BEK, Norvège), Nils Peters (CNMAT, États-Unis), Alexander Refsum Jensenius (Université d'Oslo, Norvège), Pascal Baltazar (compositeur, France), Théo De La Hogue (GMEA, France) et Julien Rabin (GMEA, France) ainsi que par les nombreux utilisateurs appor-

[1] Traduit de l'anglais par les auteurs. Définition originale : « A platform for interactive art-based research and performance ».

[2] http://opensource.org/licenses/bsd-license

tant leur contribution. Il est par ailleurs soutenu par différents organismes[3] tels que la société 74 Objects[4], BEK[5], la plateforme didascalie.net[6] ou le GMEA[7].

Notamment au travers de son implémentation dans Max/MSP, Jamoma est aujourd'hui utilisé dans de nombreux projets musicaux ou artistiques tels que ceux conduits au GMEA ainsi que dans les autres centres de création soutenant son développement et voit par ailleurs une communauté d'utilisateurs grandissante.

3. PRESENTATION DE L'ENVIRONNEMENT MODULAR

3.1. Architecture générale de Jamoma

Initialement envisagé comme un ensemble de recommandations pour le développement de modules normalisés (*patchs* Max/MSP) et leur implémentation, Jamoma recouvre aujourd'hui plusieurs librairies C++ multiplateformes utilisables notamment pour le développement d'applications multimédias[8].

L'ensemble du projet s'organise autour d'une architecture stratifiée selon ses différents objectifs [6] :

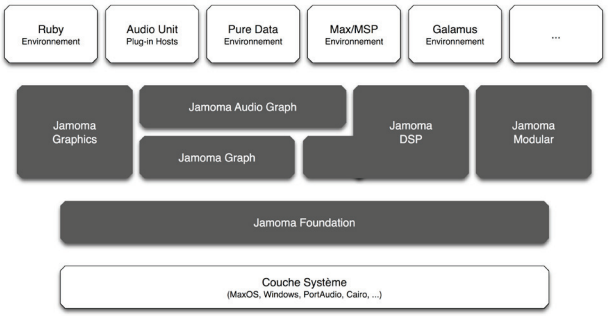

Figure 1. Architecture générale de Jamoma

- La couche Foundation constitue une librairie offrant un ensemble de classes dédiées à la construction dynamique d'objets réflexifs ;
- La couche DSP est une spécialisation de certaines de ces classes, optimisée pour le traitement de signal audionumérique ;
- Les librairies Graph et Audio Graph sont respectivement consacrées à la création de topographies permettant les communications asynchrones et synchrones entre différents objets ;
- Enfin, Graphics est une couche dédiée à la réalisation et au rendu d'interfaces graphiques.

La couche Jamoma Modular détaillée dans les sections suivantes repose sur la couche Foundation énoncée ci-dessus. Elle vise initialement à offrir une infrastructure pour le développement d'outils logiciels au sein d'environnement de programmation tels que Max/MSP, Pure Data ou plus largement d'applications autonomes.

3.2. Cahier des charges

Depuis son origine, le projet Jamoma vise à proposer, au-delà d'enjeux strictement informatiques, des solutions fondées en tout premier lieu sur les besoins issus des pratiques musicales et, à plus large échelle, aux pratiques de la création artistique en général. Les retours d'utilisateurs issus de la pratique de Jamoma dans le cadre de projets riches et variés ont ainsi pu établir le cahier des charges de la version 0.6 présentée dans cet article.

Par ailleurs, l'étude des pratiques de création [10] menée par Alessio Santini, Anne Sèdes et Benoît Simon dans le cadre du projet ANR Virage[9] dont Jamoma constituait l'un des supports d'expérimentation, a révélé de nombreux enjeux pour les logiciels de création. Dans l'article publié lors des 14e Journées d'Informatique Musicale [1], les membres du projet Virage observaient :

> *« La conclusion principale de cette étude des usages a confirmé la nécessité de travailler autour des notions d'interopérabilité et de modularité, en visant la combinaison de nos développements avec les environnements logiciels et matériels existants et correspondant aux habitudes des praticiens, plutôt que l'intégration de cette réalité complexe et multiple dans un illusoire logiciel unique et certainement trop polyvalent pour être viable. »*

Ces observations et les développements qui ont suivi ont largement participé à l'élaboration du cahier des charges de Jamoma Modular 0.6. Celui-ci comprenait notamment les objectifs d'offrir

- Une architecture orientée « Modèle – Vue – Contrôleur »[10][9] ;
- La représentation arborescente des services d'une application ;
- Des fonctions génériques de *presets*, automations, *mappings* ;
- Une exposition à tout type de réseau.

Différentes librairies proposent actuellement de manière dédiée ou partielle certaines de ces fonctionnalités

[3] Une liste des soutiens passés et actuels est consultable en ligne, à l'adresse http://jamoma.org/support.html
[4] http://74objects.com/
[5] http://bek.no
[6] http://didascalie.net
[7] http://www.gmea.net
[8] L'intégralité des codes sources de Jamoma est maintenue avec le gestionnaire de versions Git et est publiquement disponible en ligne, à l'adresse https://github.com/jamoma

[9] http://www.plateforme-virage.org
[10] L'architecture « Modèle – Vue – Contrôleur » permet de séparer le cœur opérationnel d'une application de sa présentation à l'utilisateur. http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

de *mapping* telle Libmapper[11] développée par le laboratoire IDML de l'Université de McGill ou encore les fonctionnalités de représentation arborescente telles que celles proposées au sein du projet Integra[12]. Toutefois, Jamoma Modular vise à intégrer ces solutions au sein d'un environnement unifié. Les travaux menés depuis plus d'un an ont ainsi permis d'implémenter successivement ces différents points dans la version présentée aujourd'hui.

4. JAMOMA FOUNDATION

La plupart des fonctionnalités développées dans Jamoma Modular repose sur les propriétés de la couche Foundation. Celle-ci, sur laquelle s'appuie l'ensemble du projet Jamoma, définit notamment les bases d'un environnement de programmation orienté objet ainsi qu'une librairie (NodeLib) pour la représentation sous forme arborescente.

4.1. L'environnement de programmation

Jamoma Foundation adopte un style particulier de programmation orientée objet basée sur un ensemble de classes (*class-based programming*) qui définissent les éléments de base (environnement, classe, objet, attribut, message, valeur) nécessaires au programmeur pour définir dynamiquement des objets, les instancier et les exploiter de manière générique.

L'environnement gère les objets grâce à la classe nommée `TTObject`. De nature réflexive, les instances de `TTObject` sont capables de se décrire en donnant la liste de leurs attributs et de leurs méthodes. La classe pourvoit aussi un mécanisme d'observation des attributs et des méthodes pour ainsi être notifiée d'un changement ou d'un appel. Les différents objets de Jamoma Modular héritent de cette classe et généralisent la découverte des propriétés et services d'une application.

4.2. Représentation sous forme arborescente

L'espace de nommage (*namespace*) d'une application est un moyen d'organiser et d'accéder à ses services par l'intermédiaire de chaînes de caractères (et non avec les adresses mémoires que gère l'ordinateur). Les fonctionnalités d'une application modulaire pouvant être hiérarchisées selon les différents modules qui la constituent, l'organisation des espaces de nommage est effectuée sous la forme d'une structure arborescente. Celle-ci se déploie depuis la racine (l'application elle-même) jusqu'aux feuilles (les objets qui la constituent).

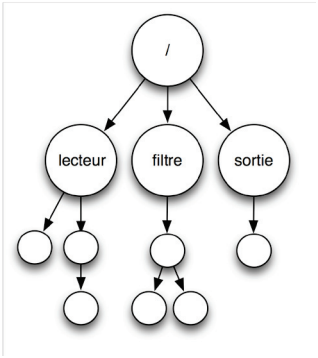

Figure 2. Arborescence d'une application modulaire

Entre la racine et les feuilles se trouvent une ou plusieurs branches représentant l'organisation des différentes fonctionnalités de l'application. Celles-ci peuvent correspondre aux différentes sous-parties de cette dernière. Les adresses sont donc des chaînes de caractères représentant le chemin depuis la racine jusqu'à l'objet désigné. Chaque niveau est séparé par un « / » (à l'exception de la racine, elle-même nommée « / »)[13].

4.2.1. Construction

La librairie NodeLib incluse dans la couche Foundation propose une classe (`TTNodeDirectory`) permettant de construire en parallèle une arborescence et une table. L'arborescence se compose de nœuds (`TTNode`) connaissant leurs enfants, leur parent et la classe `TTNodeDirectory` qui les référence. La table contient quant à elle toutes les adresses possibles de l'arborescence, permettant ainsi un accès direct et rapide aux nœuds.

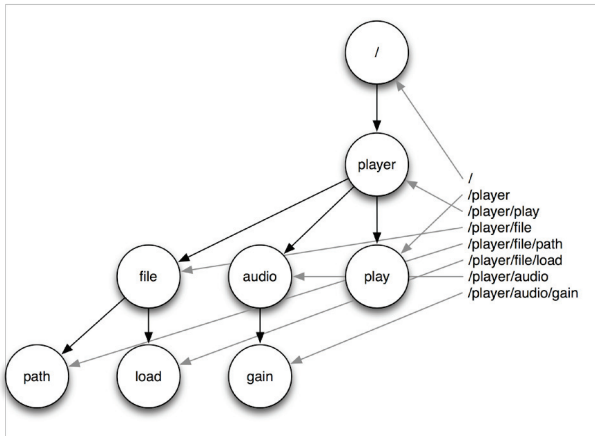

Figure 3. TTNodeDirectory

Chaque nœud contient deux informations essentielles : le nom et l'instance. La notion d'instance permet à une application de gérer des duplicatas de nœuds ayant le même nom. Les instances sont précisées lors de la création du nœud ou générées automatiquement, le cas échéant. Ainsi les nœuds référencent leurs enfants

[11] http://www.idmil.org/software/libmapper
[12] http://www.integralive.org/

[13] Le choix de la syntaxe des adresses est inspiré de la syntaxe OSC pour faciliter le routage de messages réseaux.

grâce à une table des noms dont chaque entrée référence une table des instances (dont chaque entrée référence un enfant). Un nœud peut donc avoir plusieurs enfants avec le même nom. Leur différenciation est alors effectuée grâce à leur instance. Cette notion d'instance introduit le caractère « . » comme élément de syntaxe dans les adresses pour séparer le nom du nœud de son instance.

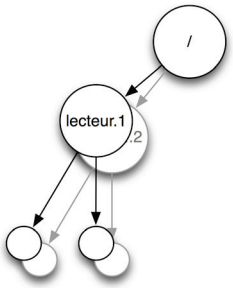

Figure 4. Instances d'un nœud dans une arborescence

Un nœud dans l'arborescence peut référencer tout type d'objet (`TTObject`). La réflexivité de ces derniers rend possible l'accès à un attribut ou l'appel d'une méthode grâce au caractère « : » dans l'adressage. Cette syntaxe ouvre la possibilité d'implémenter simplement au niveau de l'application un système de requête [8] :

```
/filtre/gain:reset
```

Figure 5. Exemple de requête (remise à l'état initial)

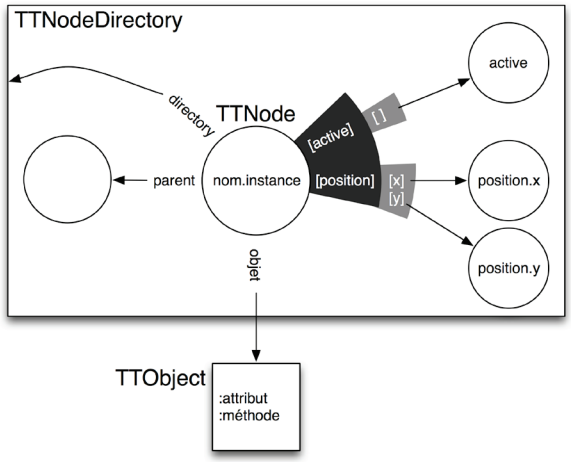

Figure 6. TTNode

4.2.2. Exploration

Le développement et l'utilisation d'une application sophistiquée invoquent un nombre important de services pouvant conduire à une arborescence vaste et complexe. Pour cela, la NodeLib pourvoit des mécanismes d'exploration et de recherche dans la base de données que constitue une telle arborescence. Il est possible de rechercher sous une adresse donnée tous les nœuds validant un test (`LookFor`) ou de savoir si au moins un nœud le valide (`IsThere`). L'exploration peut également retourner les nœuds correspondant à un type d'objet ou ceux dont les objets possèdent certains attributs égaux ou différents d'une valeur. De plus l'utilisation du caractère « * » dans une adresse permet d'obtenir tous les nœuds désignés par un nom, indépendamment de leur instance[14].

```
/piste.*/filtre.*/gain
```

Figure 7. Un exemple de requête

La requête ci-dessus cible ainsi tous les nœuds « gain » présents dans toutes les instances de l'élément « filtre », lui-même situé dans toutes les instances de l'élément « piste ».

4.2.3. Observation

La déclaration des services d'une application peut être dynamique et donc effectuée après la création de clients. Dès lors, l'observation de la création ou de la destruction de nœuds dans l'espace de nommage est nécessaire. Une observation se fait à partir d'une adresse et consiste en un mécanisme de notifications émises par le `TTNodeDirectory` après ajout ou avant suppression d'un nœud sous l'adresse observée. La notification transmet le nœud concerné à l'observateur afin qu'il puisse effectuer un filtrage sur le type ou sur les valeurs des attributs de l'objet référencé. Ici aussi, l'observation est généralisable à plusieurs adresses par l'emploi du caractère « * ».

5. JAMOMA MODULAR

Une application propose à l'utilisateur de manipuler ses fonctionnalités à travers des données qui peuvent être des commandes, des paramètres ou des retours d'information [8]. Les mécanismes d'affichage graphique, de *presets*, d'automations ou de *mappings* sont autant de clients utilisant ou souscrivant aux services des données et/ou interrogeant leurs attributs. Cette conception offre un moyen simple de mettre en œuvre une architecture « Modèle Vue Contrôleur » au sein d'une application, séparant ainsi les développements relatifs au cœur opérationnel de l'application (le modèle) de ceux relatifs à l'interface graphique (la vue), l'interface entre ses deux versants (le contrôleur) étant alors assurée par Jamoma Modular.

5.1. Déclaration des données

La déclaration de données par l'application entraîne l'instanciation d'une classe spécifique : `TTData`. Son attribut principal est en premier lieu sa valeur, mais de nombreux autres attributs spécifient également son cadre d'utilisation selon son rôle dans l'application. Ces attributs peuvent être par exemple le type de la donnée

[14] Tel que défini dans les spécifications OSC 1.0, le symbole « * » dans une adresse OSC correspond à n'importe quelle occurrence de zéro ou plusieurs caractères. (http://opensoundcontrol.org/spec-1_0)

manipulée (`integer`, `decimal`, `boolean`, `string`, `generic`, `none`) ou le service qui distingue trois rôles qu'une donnée peut supporter dans une application : `parameter`, `message` ou `return`[15]. D'autres attributs gèrent aussi :

- la valeur par défaut ;
- le filtrage des répétitions ;
- l'étiquetage (*tag*) ;
- la priorité ;
- les bornes de valeurs possibles ;
- le comportement aux bornes ;
- le pas d'incrémentation ;
- la description.

De plus, Jamoma Foundation offre pour un usage avancé un grand nombre de services également accessibles par le biais d'attributs : interpolation temporelle entre deux valeurs (librairie RampLib [8]), définition du type d'unité de la valeur (Position, Gain, Hauteur, Couleur, Température, etc.) et de l'unité dans laquelle elle s'exprime (par exemple pour la grandeur `Position` : `Cartesian3D`, `Cartesian2D`, `Spherical`, `Polar`, `OpenGL`, `Cylindrical`), conversions d'une unité à une autre (librairie DataspaceLib [8]).

5.2. *Preset*, automation et *mapping*

Fort de son implication dans les pratiques du spectacle vivant, Jamoma Modular fournit plusieurs éléments pour concevoir des mécanismes de mémorisation (*presets*, *cues*) ou de mise en relation (*mapping*) adaptés aux spécificités d'une application.

La classe `TTPreset` est dédiée à la mémorisation de l'état d'une partie de l'espace de nommage. La gestion de la mémoire est décomposée en quatre opérations que l'application doit fournir à la classe : le filtrage des objets concernés, le choix des attributs sauvegardés, le tri et le rappel des sauvegardes.

La classe `TTCue` regroupe plusieurs classes `TTPreset` pour mémoriser l'état de plusieurs parties de l'espace de nommage et en prioriser le rappel. Les classes `TTPresetManager` et `TTCueManager` gèrent quant à elles respectivement la manipulation de plusieurs classes `TTPreset` et `TTCue` pour les ordonnancer[16].

La classe `TTMapper` propose un moyen de mettre en relation la valeur d'une donnée en entrée vers la valeur d'une autre donnée en sortie. Ce *mapping* s'opère en déclarant à la classe `TTMapper` les adresses d'origine et de destination[17]. Par exemple, une mise à l'échelle par défaut est proposée en interrogeant les valeurs des bornes de chacune des données. Il est par ailleurs possible de préciser et paramétrer l'une des fonctions de transfert contenues dans la librairie FunctionLib (linéaire, puissance, cosinus, etc.) accessible depuis la couche Foundation.

5.3. Découverte et exposition sur le réseau

Pour pouvoir communiquer avec d'autres environnements (logiciels ou matériels) via divers protocoles (MIDI, OSC, DMX...) ou encore exposer son application sur le réseau pour en proposer les services, Jamoma Modular a intégré une librairie initialement développée en collaboration avec le LaBRi[18], le GMEA et Blue Yeti[19] dans le contexte de la plateforme Virage : le DeviceManager.

Ce système aujourd'hui rebaptisé ApplicationManager est conçu pour rester compatible avec de nouvelles technologies en gérant ces protocoles sous la forme de *plugins*. Ainsi chaque nouveau protocole de communication peut-être rendu compatible en implémentant un *plugin* respectant une interface. Celle-ci considère que tout protocole peut se réduire au plus à quatre types d'opération : écouter, demander, envoyer et explorer. À ce jour, trois *plugins* ont été développés lors du projet Virage : les protocoles OSC, Minuit[20] et CopperLan[21].

La classe `TTApplicationManager` gère la déclaration et la communication avec des applications distantes (représentées par la classe `TTApplication`) en leur donnant un nom et en leur associant un *plugin* de communication spécifique.

Par ailleurs, ce gestionnaire de périphériques expose l'application pour autoriser son exploration, son interrogation, son contrôle ou l'écoute des données enregistrées dans son espace de nommage. À terme, les applications s'appuyant sur la librairie Jamoma Modular et les fonctionnalités de l'ApplicationManager formeront un réseau facilement configurable et interopérable.

5.4. Sauvegarde et génération de documentation

La réflexivité des objets présentés plus haut donne la possibilité de générer automatiquement des fichiers de sauvegardes ou de la documentation standardisée dans divers formats (html, LaTeX). Le mécanisme se base sur un système de classes prenant en charge l'écriture ou la lecture de fichiers (*handler*).

En écriture, après avoir créer le fichier sur le disque, ce système de classes se charge d'appeler la méthode

[15] Ces trois rôles caractérisent respectivement des états, des commandes ou des retours d'information de l'application.

[16] Plusieurs autres fonctionnalités telles que l'interpolation entre N *presets* ou *cues* sont actuellement en cours d'implémentation.

[17] Une classe `TTMapperManager` sera prochainement implémentée pour éditer des *mappings* plus complexes et éventuellement effectuer des opérations algébriques entre les valeurs en entrées ou en sorties.

[18] http://www.labri.fr

[19] http://www.blueyeti.fr/Accueil.html

[20] Les spécifications du protocole Minuit sont disponibles en ligne à l'adresse http://www.plateforme-virage.org/?p=1889

[21] CopperLan est un protocole développé par la société belge éponyme. Une présentation générale du protocole est disponible en ligne à l'adresse http://www.copperlan.org/

`WriteAs` de l'objet qu'il décrit (et du format qu'il gère). L'objet écrit alors sa description avec les valeurs de ses attributs.

En lecture, après ouverture du fichier depuis le disque, ce même mécanisme appelle la méthode `ReadFrom` de l'objet dont il s'occupe (et du format qu'il gère). L'objet procède alors à la lecture de sa description pour changer les valeurs de ses attributs.

Un objet peut récursivement transmettre ce mécanisme de lecture/écriture à d'éventuels objets qu'il gère, permettant par exemple à la classe `TTPresetManager` de se décrire en transmettant ces opérations à l'ensemble des classes `TTPreset` qui le constitue.

Si, à ce jour, seul le `TTXmlHandler` est utilisé, il sera prochainement complété par une classe `TTTextHandler` ou d'autres formats.

6. UN EXEMPLE D'IMPLEMENTATION DANS MAX/MSP

L'environnement Jamoma au sein de Max/MSP vise à proposer aux développeurs débutants ou confirmés un ensemble de modules dédiés à la génération ou au traitement de contenu audio ou vidéo, de modules implémentant une grande variété de techniques de spatialisation, de *tracking* vidéo, etc. Il fournit également l'infrastructure nécessaire à la construction normalisée de modules personnels. Contrairement à d'autres librairies personnelles ou commerciales visant à améliorer ou faciliter le développement (TapeMovie[22], ou plus récemment Vizzie[23]) conçues depuis les fonctions standards de Max/MSP, l'implémentation de Jamoma dans Max/MSP repose sur une librairie d'*externals* spécifiques. Ces derniers exploitent les fonctionnalités des différents *framework* en vue de standardiser l'échange de données et de requêtes, systématiser la séparation des moteurs et de leur représentation graphique au sein d'une architecture « Modèle Vue Contrôleur » dans Max/MSP.

6.1. *Externals* Max/MSP hérités de classes Jamoma

La génération des objets Max/MSP (*externals*) à partir des `TTObject` des différentes librairies de Jamoma se fait automatiquement grâce à un mécanisme d'emballage (*wrapping*). Celui-ci consiste à interroger la classe pour lui demander ses attributs et messages afin de créer les attributs et messages tels que définis par l'API Max. L'*external* ainsi généré bénéficie des mêmes fonctionnalités que la classe Jamoma et permet aux programmeurs Max/MSP d'utiliser les outils décrits précédemment.

Par exemple, les objets `jcom.parameter`, `jcom.message` et `jcom.return` héritent de la classe `TTData`, `jcom.namespace` emballe la classe `TTExplorer` tandis que les objets `jcom.send` et `jcom.receive` exposent directement les fonctionnalités des classes `TTSender` et `TTReceiver`.

```
int TTCLASSWRAPPERMAX_EXPORT main(void)
{
return wrapTTModularClassAsMaxClass(
       TT("Mapper"), "jcom.map", NULL);
}
```

Figure 8. Génération de l'*external* `jcom.map`

Ce mécanisme accepte cependant d'intégrer des spécificités liées à Max/MSP. Ainsi, plusieurs *externals* ne sont pas des emballages directs de classes mais augmentent ces dernières avec un ensemble de fonctionnalités propres à cet environnement. De plus, certaines classes ne font pas l'objet d'un emballage mais sont utilisées dans le code pour offrir des commodités (`TTXmlHandler` par exemple).

L'ensemble des fonctionnalités abordées avec la description de Jamoma Modular est ainsi disponible à travers divers *externals*.

6.2. Construction d'un *patch* « modèle »

Un *patch* modèle correspond au cœur opérationnel du programme. Il peut abriter une technologie pour générer, traiter ou diffuser des médias qu'un programmeur Max/MSP souhaiterait rendre accessible en proposant une logique d'utilisation.

Cette opération est réalisée en insérant dans un *patch* les *externals* propres à la déclaration de paramètres (caractérisant l'état du patch), de commandes ou de retours d'information. Les objets `jcom.parameter`, `jcom.message`, et `jcom.return` sont trois emballages spécifiques de la classe `TTData` (dont l'attribut « service » est respectivement égal à `parameter`, `message` ou `return`).

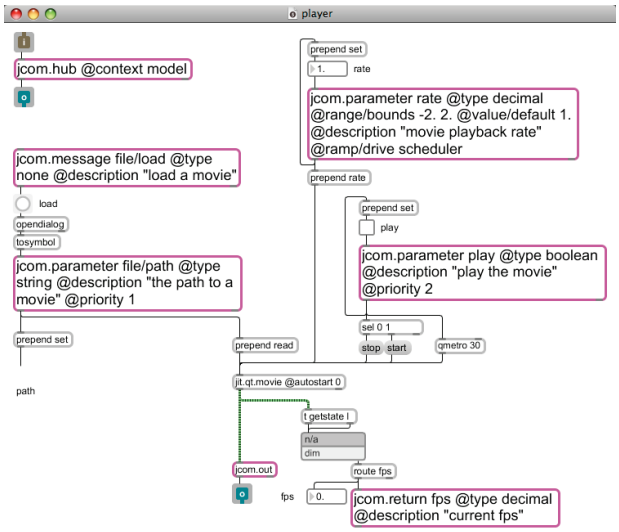

Figure 9. Modèle de lecteur vidéo

Lorsqu'un programme Max/MSP instancie un *patch* modèle, l'espace de nommage est automatiquement augmenté des noms des paramètres, messages et retours

[22] http://tapemovie.org/
[23] http://cycling74.com/2010/11/19/introducing-vizzie/

que contient ce dernier. Ceux-ci sont enregistrés sous la racine pour constituer un namespace « plat », c'est-à-dire sans branche.

L'aspect arborescent d'un programme Max/MSP (comprenant plusieurs niveaux de sous-*patches*) est obtenu en insérant l'objet `jcom.hub` et en spécifiant son attribut `@context model`. Sa présence concrétise l'existence d'une branche dans l'arborescence. Les adresses des objets `jcom.parameter`, `jcom.message`, `jcom.return` sont dès lors relatives au *hub* et sont enregistrées sous un nœud dont le nom est celui du *patch*[24].

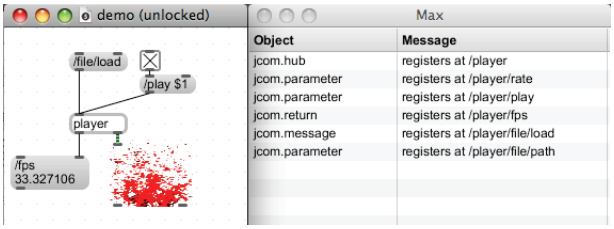

Figure 10. Espace de nommage du modèle

L'arborescence devient alors représentative de l'organisation du programme rendant par exemple possible la création de modèles incluant eux-mêmes d'autres modèles de pistes en insérant des modèles de gains, de filtrages ou d'équalizations.

6.3. Construction d'un *patch* de « vue »

Un *patch* de vue permet de créer une interface totale ou partielle vers le *patch* modèle, autorisant la réalisation d'une logique d'utilisation spécifique et adaptée à un contexte. La construction consiste à déclarer par des objets `jcom.view`[25] l'accès et/ou l'observation de paramètres, de messages ou de retours d'information.

Figure 11. Interface pour le lecteur vidéo

Il est donc possible de sélectionner les données d'un ou plusieurs *patches* modèles qu'un utilisateur souhaiterait voir regrouper. De plus, il est possible de changer dynamiquement le choix des paramètres, messages et retours observés. Ainsi un seul *patch* de vue peut rendre compte tour à tour de l'état de différentes instances d'un modèle.

L'instanciation d'un *patch* de vue enregistre aussi les noms des objets `jcom.view` dans l'arborescence selon la même logique décrite pour les patches modèles.

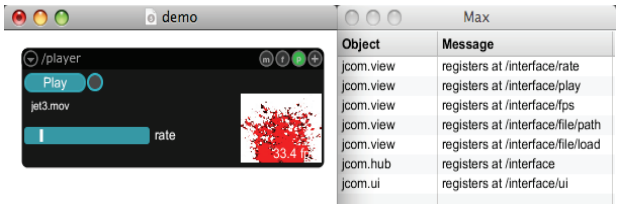

Figure 12. Espace de nommage de l'interface

7. UN AUTRE EXEMPLE D'UTILISATION PAR GALAMUS

Au-delà du strict cadre de la création musicale, la société Galamus-Software[26] illustre un autre cas de travaux exploitant certaines fonctionnalités de Jamoma Modular. Nouvelle société de recherche et développe-

[24] Ces fonctionnalités spécifiques à Max/MSP ont inspiré le développement des classes `TTContainer` et `TTsubscriber`. Ces classes sont donc à la disposition de n'importe quel projet C++ mais il reste à évaluer dans quelle mesure leur spécification s'avère suffisamment générique pour intéresser d'autres environnements.

[25] L'objet `jcom.view` hérite de la classe `TTViewer` qui regroupe un `TTSender` et un `TTReceiver`.

[26] http://galamus-software.fr

ments, conception et édition de logiciels spécialisés, Galamus-Software développe[27] des solutions interactives innovantes et créatives notamment pour l'affichage dynamique, le *digital media*, la communication interactive. La société partage à ce titre la nécessité de mise en communication d'environnements logiciels différents, devant pouvoir communiquer entre eux et être facilement paramétrables.

7.1. Galalib

Dans le cadre de ses activités, Galamus travaille actuellement à la réalisation d'une API C++ multiplateforme et *Open Source* développée et testée parallèlement sous Mac OSX et sous Windows7, permettant d'étendre aisément certaines fonctionnalités de Jamoma Modular à d'autres environnements logiciels.

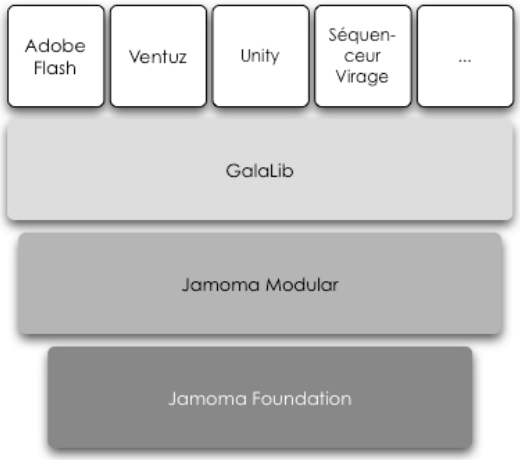

Figure 13. Organisation générale des librairies Jamoma et Galamus

Galalib fournit une interface « Namespace.h », qui met à disposition un ensemble de méthodes permettant de créer et de gérer le plus simplement possible une arborescence de paramètres, à savoir créer et supprimer des paramètres, mettre à jour et récupérer des attributs, créer des écoutes sur des paramètres. Ces opérations peuvent avoir lieu à l'intérieur de l'application mais également entre des applications distantes.

Galalib utilise les fonctionnalités de *preset* et de *mapping* de Jamoma Modular. Par ailleurs, elle possède des méthodes de chargement et sauvegarde des espaces de nommage dans des fichiers au format xml : par exemple il est possible de créer une arborescence de paramètres en l'écrivant dans un fichier xml dédié. Celle-ci sera ensuite automatiquement générée au chargement du fichier.

Cette bibliothèque vise ainsi à faciliter la prise en main de ces environnements aux développeurs de Galamus et à toute personne ayant des besoins similaires.

[27] Les activités de recherche et développement sont coordonnées par Jean-Michel Couturier (également le co-fondateur de Blue Yeti, structure partenaire du projet VIRAGE)

7.2. Exemples de fonctionnalités

Outre son intégration dans les logiciels mentionnés dans l'illustration précédente (Adobe Flash, Ventuz, Unity, séquenceur Virage) l'emploi de Galalib est également prévu au sein de

- Drivers Galalib pour des périphériques gestuels ;
- Drivers Galalib pour des applications de rendu et de diffusion de contenus ;
- Outils de gestion Galamus.

La figure suivante donne un exemple d'utilisation de trois applications communicant par la Galalib :

- une application de captation (par exemple renvoyant la position de la main de la personne située devant l'écran) ;
- une application de diffusion (par exemple affichant un contenu 3D à l'écran) ;
- un séquenceur (comme celui du projet Virage) permettant de modifier le mapping et la valeur de certains paramètres dans le temps.

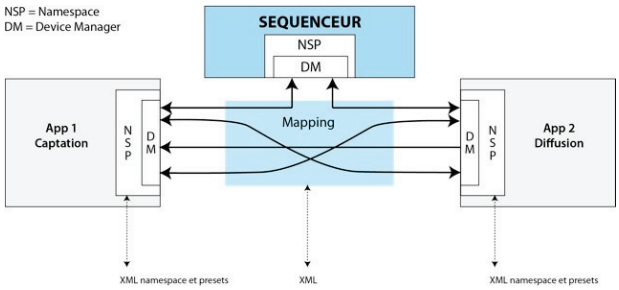

Figure 14. Un exemple d'applications communicant par la Galalib

Chaque application a accès à l'espace de nommage des autres applications par le réseau, via la classe `TTApplicationManager`. Une application peut également charger le fichier (au format xml) préalablement sauvegardé de l'arborescence d'une seconde application. Ceci autorise notamment le travail hors réseau.

Ainsi les différentes briques logicielles pourront communiquer pour modifier des paramètres, créer des *mappings* et charger des *presets*.

8. CONCLUSION

Jamoma Modular 0.6 résulte des réflexions et développements engagés depuis plus d'une année au sein du GMEA et plus largement au sein du projet Jamoma. Au terme de ces efforts, les premiers retours de tierces développeurs semblent confirmer l'intérêt de la librairie pour le développement de logiciels liés à la création. Par ailleurs, notamment au travers de l'implémentation de ses fonctionnalités sous la forme d'*externals* Max/MSP dont la version *alpha* a été publiée en février dernier, Jamoma Modular semble satisfaire les besoins initialement établis. L'architecture orientée « Modèle – Vue –

Contrôleur » autorise la séparation des différents centres d'attention et la déportation de l'interface de contrôle. L'emploi d'une structure arborescente pour la représentation de la structure d'un logiciel semble un moyen efficace et flexible pour le développeur ou l'utilisateur final d'explorer les fonctionnalités d'une application. De plus, l'ajout de fonctions génériques semble répondre aux besoins de logiciels appliqués à la création ou, à plus large échelle, aux logiciels convoquant des médias de différente nature. Enfin, depuis les développements initiés dans le cadre du projet Virage et à travers les efforts combinés de Jamoma et de Galamus, une passerelle générique pour la communication au sein d'un réseau est aujourd'hui disponible.

Dans les mois à venir, plusieurs aspects devront recevoir une attention particulière. Tout d'abord, il convient de multiplier les phases d'expérimentation et d'éprouver la solidité et l'efficience de Jamoma Modular. Des mesures de performances dans le cadre de projets de création complexes pourront alors guider les optimisations éventuelles. Ensuite, au-delà du projet Jamoma et des objectifs qui sont les siens, son degré de généricité devra continuer d'être évalué avec soin. Il importe pour cela d'envisager les projets ayant recours aux fonctionnalités de Jamoma Modular et ceux à venir comme autant de moyens de soutenir et poursuivre les échanges engagés entre les membres des communautés informatique et musicale.

9. REFERENCES

[1] Baltazar, P., Allombert, A., Marczak, R., Couturier, J-M., Roy, M., Sèdes, A., Desainte-Catherine, M. « Virage : Une réflexion pluridisciplinaire autour du temps dans la création numérique », Actes des 14e Journées d'Informatique Musicale, Grenoble, 2009.

[2] Baltazar, P., Gagneré, G. « Outils et pratiques du sonore dans le spectacle vivant », Actes des 12e Journées d'Informatique Musicale, Lyon, avril 2007.

[3] Battier, M., Schnell, N. « Introducing composed instruments, technical and musicological implications », Proceedings of the 2002 conference on New Instruments for Musical Expression, Dublin, 2002.

[4] Bossis, B. « Écriture instrumentale, écriture de l'instrument », Actes du colloque international : Composer au XXIe siècle : processus et philosophies, Montréal, 2007.

[5] Malloch, J., Sinclair, S., Wanderley, M. M. « A network-based framework for collaborative development and performance of digital musical instruments », Proceedings of the 2007 Computer Music Modelling and Retrieval, Berlin, 2008.

[6] Place, T., Lossius, T., Peters, N. « A flexible and dynamic C++ framework and library for digital audio signal processing », Proceedings of the International Computer Music Conference, New York, États-Unis, 2010.

[7] Place, T., Lossius, T., Refsum Jensenius, A., Peters, N. « Flexible control of composite parameters in Max/MSP », Proceedings of the International Computer Music Conference, Belfast, 2008.

[8] Place, T., Lossius, T., Refsum Jensenius, A., Peters, N., Baltazar, P. « Addressing classes by differentiating values and properties in OSC », Proceedings of the 2008 conference on New Instruments for Musical Expression, Genova, 2008.

[9] Reenskaug, T. « Models – Views – Controllers », Notes de synthèse, Xerox PARC, décembre 1979. Une version électronique au format pdf est disponible en ligne à l'adresse http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf (*Adresse électronique vérifiée le 03/03/2011*).

[10] Santini, A., Sèdes, A., Simon, B. « Virage : Analyse des usages/État de l'art », rapport interne, version n° 3, Paris, 2009. Une version électronique est disponible en ligne à l'adresse http://www.plateforme-virage.org/?p=1550 (*Adresse électronique vérifiée le 23/02/2011*).