Numéros / n° 7-8 - Culture du code

Â«Â Architecture logicielle du projet ossia Â Â»

Jean-Michaël Celerier

Résumé

Le développement du séquenceur ossia score de manière continue depuis 2014 a été l?occasion d?explorer certains patrons de conception traditionnels dans le cadre d?un logiciel écrit en suivant les normes récentes du langage C++ : 2011, 2014, 2017. Ce document présente une vue d?ensemble de l?architecture du projet ossia et détaille certains points d?implémentation dans l?optique d?en offrir une perspective contextualisée par les idiomes de développement C++ en 2020.

1. Historique et usages du projet ossia

L?environnement ossia est une suite d?outils logiciels pour la création artistique numérique (Celerier, 2018). Il se compose principalement d?ossia score, un séquenceur pour l?écriture de l?interactivité, et de libossia qui assure l?interopérabilité des outils multimédia. Développé dans la continuité des projets Boxes, Virage et i-score, il résout plusieurs problèmes auxquels se trouvent souvent confrontés les créateurs et réalisateurs en informatique musicale, théâtrale ou plus généralement créative.
Le développement d?ossia, et score en particulier, a été l?occasion d?expérimenter diverses techniques de développement logiciel, de multiples patrons de conceptions et des choix d?architecture possibles pour des logiciels auteurs multimédia. Cet article présente un compte-rendu des choix réalisés durant l?écriture du logiciel et des spécificités qui en découlent, dans le but d?en permettre le partage et la discussion avec la communauté de l?informatique musicale.

On présentera d?abord les grandes lignes de l?architecture logicielle de libossia, qui est une bibliothèque en C++ ayant pour objectif de s?intégrer dans un maximum d?environnements logiciels existants afin de permettre la communication entre chacun, *via* un modèle de données fixé dans libossia. La partie suivante décrira brièvement le fonctionnement de score, qui est le logiciel de création et d?exécution de partitions interactives basé sur libossia. Un détail des patrons de conception utilisés dans score y succédera, avec les remarques que l?on peut y apporter en tenant compte de l?expérience acquise lors du développement. Pour conclure, les axes de recherche actuellement en phase exploratoire pour le projet ossia seront discutés.

1.1. Qu?est-ce qu?ossia ?

Une brève introduction aux concepts principaux de libossia et ossia score est nécessaire. Cette suite de logiciels a été créée dans l?objectif d?écrire et d?exécuter des partitions interactives, pouvant impliquer différents logiciels et matériels. On veut, par exemple, pouvoir réaliser la partition suivante, constituée d?un jeu de lumière piloté par Max/MSP, d?une bande son et d?une manette de contrôle :

- La partition commence par la lecture d?un premier son.

- À partir de 10 secondes, un deuxième son est joué. Quand l?interprète bouge la manette dans l?axe *X*, l?intensité lumineuse est égale au volume de la bande son multiplié par l?amplitude du mouvement. Si, au bout de 25 secondes, la manette va au maximum de l?axe *X* (1), la partition reboucle à 10 secondes. Il en va de même au bout de 35 secondes.
- À partir de 48 secondes, si la manette va au minimum de l?axe *X* (- 1), la partition se termine : un fondu met progressivement fin à la lumière et à un troisième son ; le joystick n?a plus aucun effet.

La figure 1 en présente l?écriture dans ossia score. Cette écriture présuppose l?existence de paramètres à contrôler, ici : **joystick:/x** et **max_osc:/lumiere**. Ces paramètres sont accessibles dans le logiciel *via* un arbre basé sur la spécification OSC, qui offre une abstraction sur les protocoles couramment utilisés en art numérique : OSC, MIDI, Artnet, etc. La bibliothèque libossia permet d?utiliser et de partager aisément ces structures d?arbres sur le réseau dans différents logiciels.

Figure 1. Exemple de partition ossia score

Note : L?écriture suit principalement une logique de *time-line*, avec pour particularités des éléments permettant de briser la linéarité du temps : les *triggers* (signes *T* en jaune) et la boucle, présentée au centre. La courbe violette est une fonction de transfert qui est appliquée régulièrement à chaque message en entrée. La courbe rouge est une automation. La courbe rouge en pointillés est une automation spéciale : elle part de la valeur courante du paramètre automatisé plutôt que d?une valeur fixe, ce qui permet des transitions douces. Les lignes bleues horizontales sont appelées intervalles et dénotent un écoulement du temps pendant lequel des processus peuvent s?exécuter. Certains processus sont de simples générateurs ou fonctions de transfert, tandis que d?autres sont des conteneurs : toute cette partition est par exemple contenue dans un processus « Scenario » qui pourrait être dupliqué ou lui-même encapsulé.

Source : Jean-Michaël Celerier

1.2. Contexte dans l?univers de la création multimédia interactive

Ossia s?inscrit dans la famille des logiciels pour la création d?art numérique interactif, un domaine riche. L?un d?eux est IanniX (Coduys, 2004), un descendant du système Upic de Xenakis, qui permet de créer des partitions graphiques interactives : plusieurs lignes de temps sont possibles, représentées par des curseurs pouvant suivre des trajectoires en plusieurs dimensions et exécuter des fonctions données par l?utilisateur à chaque instant, pour réaliser des *mappings* par exemple. Le logiciel est écrit en C++ avec la bibliothèque Qt.

Ascograph (Burloiu, 2015) propose une interface graphique réalisée en C++ avec OpenFrameworks pour le langage réactif utilisé par le système Antescofo. Plus récemment, on notera Kiwi (Paris, 2017), un système proche des *patchers* traditionnels tels que Max/MSP ou PureData, qui propose à plusieurs utilisateurs de modifier et d?exécuter le même patch en réseau. Kiwi est écrit en C++ avec Juce ? on notera notamment l?effort fait par ses auteurs pour adhérer aux pratiques actuelles de développement en C++.

2. L?écosystème ossia et libossia

Le c?ur de fonctionnement d?ossia est la bibliothèque libossia [(1)], qui assure l?interopérabilité entre les différents logiciels et matériels. Cette bibliothèque contient aussi les fonctions et types de données utilitaires et ceux qui n?ont pas de dépendance à une interface graphique. Le moteur d?exécution de score en fait partie.

Le projet Jamoma (La Hogue, 2011), sur lequel les premières versions d?ossia score (alors appelé i-score) était basé, était séparé en plusieurs bibliothèques logicielles (Foundation, Modular?) pouvant, de plus, chacune posséder des greffons (*plugins*). À la place, nous avons fait le choix pour libossia d?une unique bibliothèque, configurable à la compilation pour les utilisateurs ayant des besoins spécifiques. En effet, la séparation de Jamoma en multiples bibliothèques a été à la source de difficultés d?intégration avec des environnements tiers tels que Max/MSP et tendait à complexifier la compilation sur différentes plateformes.

2.1. Organisation logicielle

La bibliothèque libossia contient le code source de la bibliothèque principale, ainsi que les codes sources des adaptations à des environnements de programmation créative : Max/MSP, PureData, Unity3D? La bibliothèque principale est elle-même séparée en sous-dossiers :

- *audio* : interfaçage avec les cartes son ;
- *dataflow* et *editor* : moteur d?exécution de score, *editor* contient les structures temporelles, *dataflow* le graphe de données ;
- *network* : protocoles de communication et modèle d?arbre basé sur OSC ;
- *preset* : système de préréglages pouvant être chargés et sauvegardés, pour l?arbre défini par *network* .

Pour chaque adaptation, l?objectif est d?adhérer le plus possible aux contraintes de programmation des environnements existants [(2)], en particulier si l?environnement possède une structure hiérarchique native : libossia se base alors sur cette dernière dans la mesure du possible.

2.2. Communication avec libossia

La partie *network* définit un arbre basé sur les principes du protocole OSC (Wright, 1997), de manière à pouvoir être adapté facilement à d?autres protocoles. L?architecture repose sur quatre types de base dont on détaille la terminologie (en gras, le nom des types de données dans le code source) :

- Les protocoles (**protocol_base**) : implémentent un protocole de communication donné ? OSC, ArtNet, OSCQuery, MIDI?
- Les n?uds (**node_base**) sont la structure de base qui forme un arbre OSC.
- Les périphériques (**device_base**) sont la racine de l?arbre, un n?ud spécial utilisé pour l?identifier. Chaque *device* est associé à un protocole. Un protocole spécial de multiplexage existe.
- Les paramètres (**parameter_base**) : peuvent être associés à un n?ud et lui confèrent des attributs tels qu?un type, une valeur, un domaine de définition, une unité?

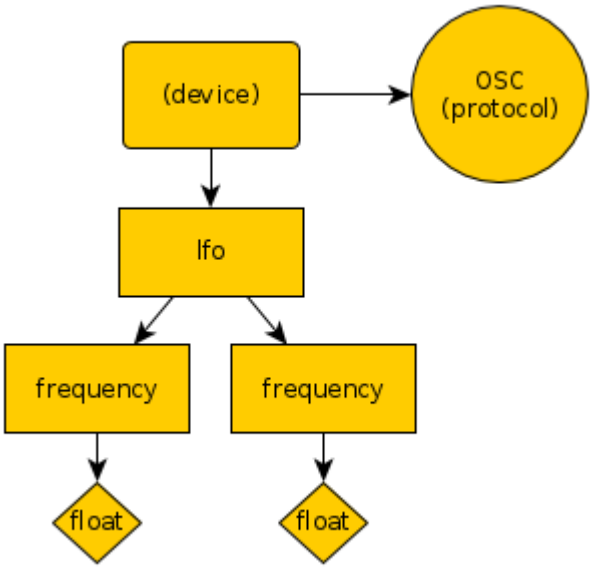
Prenons par exemple les messages OSC suivants :

/lfo/frequency 1.0

/lfo/amplitude 0.5

Pour qu?un programme utilisant libossia puisse recevoir ce message, les structures données en figure 2 sont nécessaires.

Figure 2. Diagramme d?objets pour la représentation d?un arbre OSC



Note : Le protocole est dans un cercle, le périphérique dans un rectangle arrondi, les n?uds dans des rectangles et les paramètres dans des losanges. La flèche signifie une relation de composition.

Source : Jean-Michaël Celerier

Le périphérique possède un nom, afin que l?on puisse adresser plusieurs périphériques depuis le même logiciel utilisant libossia. Par exemple, si le périphérique se nomme **synth**, on pourra écrire **synth:/lfo/frequency** pour se référer au premier n?ud dans les objets Max/MSP de libossia.

Une autre architecture ayant pour but de résoudre un problème similaire a été essayée dans la bibliothèque coppa [(3)]. Plutôt que d?utiliser des objets individuels, on utilise une seule table de hachage, associant à chaque paramètre un tableau de fonctions. Cependant, cela a entraîné en pratique un coût en mémoire et une complexité de programmation plus importante pour assurer un fonctionnement dans le cadre *multi-thread*.

L?expérience dont nous disposons désormais nous permet d?affirmer que l?on ne choisirait pas à nouveau d?implémenter un fonctionnement *multi-thread* tel qu?il est actuellement supporté dans libossia (la garantie offerte est la possibilité pour un fil d?exécution de changer la valeur d?un paramètre pendant qu?un autre lit cette valeur). En effet, la plupart des environnements intégrant libossia (Max/MSP, Unity3D?) ne supportent les modifications de leur propre modèle de données que depuis un unique fil d?exécution en dehors de cas bien précis. Les fonctionnalités de synchronisation n?ont donc pas d?intérêt et consomment inutilement des ressources dans ce cas d?usage très courant.

3. Le modèle logiciel de score

Score est un séquenceur pour l?intermédia et l?interactivité. « Séquenceur » signifie que la nature du logiciel est de séquencer des évènements et processus dans le temps. L?intermédialité implique que le logiciel permet de contrôler et mettre en scène diverses formes de média (son, vidéo, MIDI?) par le biais

de commandes communiquées à d?autres logiciels. Grâce à l?interactivité, les partitions créées avec score pourront varier selon les performances, *via* des directives apposées par les compositeurs dans les partitions et prises en compte par le moteur d?exécution du logiciel.

Figure 3. Capture d?écran d?ossia score dans sa version 2.5.2

Note : L?interface graphique complète de ossia score est affichée, avec différents processus dans un scénario : automations, dégradés de couleurs, scripts JavaScript, notes MIDI, greffons VST et Faust, fichiers sons. Un arbre ossia est visible sur la gauche.

Source : Jean-Michaël Celerier

3.1. Le modèle des partitions dans ossia score

Les éléments de base du modèle, réifiés par des objets C++, sont :

- Processus (**ProcessModel**) : est une fonction dépendante du temps, exécutée à intervalle régulier. C?est la principale manière d?étendre score avec de nouvelles fonctionnalités. Par exemple, les greffons VST, les courbes d?automation et les programmes Faust sont intégrés en tant que processus. Ces derniers ont des entrées et sorties définies par des ports pouvant soit être connectés les uns aux autres, soit envoyer ou recevoir des messages aux adresses définies par les n?uds ossia.
- Intervalle (**IntervalModel**) : est un intervalle temporel qui peut délimiter le début et la fin d?un groupe de processus.
- Point de synchronisation (**TimeSyncModel**) : marque le début et la fin d?un groupe d?intervalles. Il permet notamment d?attendre que les intervalles antécédents aient fini de s?exécuter avant de passer à la suite et se déclenche lorsqu?une condition externe est vérifiée. Il est assimilable à certaines constructions de type *when (x) then (y)* dans les langages réactifs.
- État (**StateModel**) : est un ensemble de données et de fonctions à exécuter à un instant précis. Chaque intervalle débute et se termine par un état.
- Condition (**EventModel**) : permet d?exécuter sélectivement des intervalles et des états selon une condition externe. Cela représente une forme de *if (x) then (y)*. Un point de synchronisation peut posséder plusieurs conditions et une condition peut posséder plusieurs états.

Certains processus permettent d?organiser ces évènements selon des règles données. Les trois principaux sont **Scenario**, **Loop** et **Nodal**. **Scenario** est le processus principal. Il permet de créer et séquencer les éléments du modèle sous forme de graphe dirigé acyclique dans une *timeline*. **Loop** est un processus qui possède un unique intervalle, débuté et terminé par deux points de synchronisation. Lorsque le second point est exécuté, l?exécution reprend au premier, ce qui permet la répétition. **Nodal** permet d?agencer les

processus sous forme de graphe de flots données, exactement à la manière de PureData ou Max/MSP.

3.2. Un éditeur de document générique

L?un des besoins principaux pour le développement de score était l?extensibilité : il fallait pouvoir adapter le logiciel à de nouveaux cas d?utilisation, créer de nouveaux processus, etc. Les cahiers des charges réalisés pendant les projets ANR Virage et Ossia ont mis en évidence ce besoin d?étendre les capacités du logiciel pour faire face aux besoins artistiques sans cesse changeants, ainsi qu?aux futurs périphériques et logiciels avec lesquels une compatibilité devra être possible.

Le logiciel repose donc sur une bibliothèque de base, indépendante du domaine, qui offre de nombreux points d?extensibilité et permet à des greffons logiciels de rajouter leurs propres fonctionnalités, ainsi que de créer de nouveaux points d?extension. Par exemple, une extension à score permet la création de filtres vidéo. Cette extension publie aussi une interface qui permet à d?autres extensions d?ajouter de nouveaux filtres par la suite.

L?emphase a été mise très tôt sur la séparation en greffons pour chaque fonctionnalité majeure du logiciel, plutôt qu?une base de code monolithique. Cela force le développeur à concevoir un arbre de dépendances correct, et non circulaire, ce que l?éditeur de liens du système refuserait. Lorsqu?un cycle apparaît, il est extrait dans un nouveau greffon.

Cela n?est pas sans coût, ajoute des indirections et complexifie le développement, si l?on compare avec une structuration monolithique du code. Le coût en performance peut être atténué *via* les options d?optimisation des éditeurs de liens modernes, qui permettent d?optimiser les appels au travers des bibliothèques. En revanche, celle de la complexité du code est sans réponse : bien qu?étant un bénéfice de manière globale pour score, des logiciels avec un champ d?application plus réduit et correctement défini dès le début du projet pourraient finalement se retrouver perdants, en comparaison avec une architecture monolithique.

Le modèle de document repose sur des objets organisés en un arbre. Les processus et intervalles, entre autres, sont tous des n?uds de cet arbre. Les objets possèdent tous un identifiant qui permet de rechercher et référencer un objet donné *via* une chaîne de caractère, ce qui est utile pour la sauvegarde ainsi que pour le mécanisme d?*undo-redo*. On peut noter que cela ne concerne pas les structures d?exécution fournies par libossia, qui sont plus simples.

4. Patrons de conception dans score

On considère dans cette section les patrons de conception principalement utilisés dans score, en notant avec l?expérience acquise sur le développement du logiciel quels ont été les points forts et faibles de chacun.

4.1. Séparation modèle-vue-présenteur

Les étapes initiales du développement de score ont motivé un état de l?art du développement d?interfaces graphiques. À cette période (v. 2014), le patron de conception « modèle-vue-présenteur » (Potel, 1996) est apparu comme une méthode pertinente et validée par de nombreux logiciels existants. Cependant, avec plusieurs années d?expérience de ce patron de conception, quelques réserves peuvent être émises. Ces remarques sont à considérer dans le contexte d?un logiciel développé avec la bibliothèque Qt, mais la plupart des bibliothèques d?interface graphique actuelles pour les langages à typage statique suivent des conceptions similaires.

Considérons l?historique du patron de conception modèle-vue-présenteur (Potel, 1996). D?une part, à l?époque de sa conception, les bibliothèques de composants étaient beaucoup moins riches qu?elles ne

sont aujourd?hui. D?autre part, les langages de programmation concernés par l?ouvrage de Mike Potel, C++ et Java, ont subi de nombreuses améliorations depuis 1996 qui permettent d?exprimer les patrons de conception avec plus de facilité.

Potel (1996) présente notamment des cas d?utilisation, tels que la création de vues en liste ou en arbre. Ce type de vues est fourni par Qt automatiquement : le travail de présentation est déjà effectué. La majorité du développement, dans un logiciel tel que ossia score, consiste en la spécification du modèle de données de ces vues et du positionnement de l?objet de liste dans l?interface graphique, ce qui ne requiert que peu de présentation.

Une partie importante du travail du présenteur est l?application des commandes de la vue vers le modèle de données. L?expérience de développement nous a montré qu?il était souvent aussi rapide de mettre en place la création des commandes dans la vue, car les détails d?implémentation de cette dernière, tels que le rendu, sont déjà gérés par la bibliothèque Qt.

Le support des fonctions lambda des éditions récentes du langage C++, associé au mécanisme de signaux et slots de Qt, rend ce procédé très simple. Un extrait d?un objet « inspecteur » pour l?un des processus de score est donné en exemple en annexe *A*.

4.2. Commandes et sérialisation

De manière générale, il est impossible d?empêcher les *crashs* dans un logiciel pouvant charger des greffons à l?intérieur du même processus : n?importe quel greffon VST peut par exemple causer une erreur de segmentation. Une fonctionnalité fortement appréciée des utilisateurs de score est alors la restauration, en cas de *crash*.

Les commandes stockent le chemin des objets du modèle mentionné en section III. 2., en variable membre. Elles possèdent une méthode de sérialisation/désérialisation binaire efficace. Chaque nouvelle commande est sérialisée dans un dossier temporaire du système et l?exécution d?une nouvelle commande induit, de plus, l?exécution de vérifications sur le modèle de données afin de garantir sa cohérence, ce qui a deux avantages :

- Cela permet de mettre en évidence les problèmes très facilement durant le développement.
- Cela empêche l?utilisateur de travailler avec un document corrompu et de le sauvegarder, ce qui pourrait causer des pertes de données.

L?interface des commandes est donnée pour référence en annexe *B*.

4.3. Extensibilité avec sûreté de typage pour les processus simples

Les auteurs d?*externals* pour les logiciels tels que Max/MSP ou PureData sont au fait des problèmes que peuvent causer les erreurs de type : le fait de lire ou écrire les bons types de données dans les entrées et sorties de ces objets repose sur le programmeur. Le compilateur n?empêchera pas, par exemple, d?écrire une chaîne de caractère là où un entier est attendu. C++ permet d?assurer un typage plus fort : cela réduit la possibilité d?écrire du code qui compile, mais échoue à l?exécution en raison d?erreurs de types.

Score offre une interface permettant d?écrire des processus sans crainte de commettre des erreurs de types. Cette interface utilise les capacités de réflexion actuelles de C++ qui reposent principalement sur le mécanisme des *templates*, ce qui crée un code complexe [(4)] pour l?auteur de la bibliothèque, mais très simple pour son utilisateur.

Un exemple d?un tel greffon qui applique un gain à un signal audio est présenté en annexe *C*.

Conclusions

En sus des considérations discutées dans ce document, l?une des raisons importantes ayant permis le développement rapide de score est l?adhérence à certaines pratiques de développement : l?utilisation de services d?intégration continue (Travis CI, AppVeyor), l?utilisation des compilateurs les plus récents en activant les derniers avertissements introduits, l?utilisation d?analyseurs statiques tels que Coverity ou clang-analyzer, la vérification régulière avec des outils d?analyse mémoire (en particulier AddressSanitizer) et l?écriture de tests (unitaires pour libossia et d?intégration pour score).
En travaillant sur ce projet, on a pu remarquer qu?un support natif de méthodes de réflexion dans le langage C++ permettrait de nombreuses simplifications. En particulier, on estime que les travaux réalisés par Herb Sutter (2019) pourraient réduire le code source de score de plusieurs milliers de lignes, car les problèmes que nous devons résoudre (génération d?interface graphique pour des modèles de données, par exemple) sont en adéquation parfaite avec les solutions apportées dans ce document.

Les développements plus récents de score ont porté sur trois axes : la musicalité, c?est-à-dire la gestion du tempo, des changements de mesure et de quantification ; les graphismes, qui permettent de créer des visuels utilisant le GPU ; la compilation à la volée (Celerier, 2019), qui permet de définir de nouveaux processus en C++ directement dans une partition, pour les partitions ayant besoin de réaliser du calcul haute performance. De ces trois axes, seul le premier a nécessité des modifications importantes du modèle de données existant pour prendre en compte les nouvelles fonctionnalités. Les deux autres ont pu se faire simplement *via* des ajouts de code (sans considérer toutefois le réglage de bogues), ce qui est une forme de validation des principes d?extensibilité appliqués lors du développement.

---

1. https://github.com/OSSIA/libossia [consulté le 24/11/2020]

2. Leur usage est détaillé ici : https://ossia.github.io [consulté le 24/11/2020]

3. https://github.com/jcelerier/coppa [consulté le 24/11/2020]

4. On pourra par exemple s?intéresser au fichier qui implémente l?exécution des greffons définis *via* cette méthode : https://github.com/OSSIA/libossia/blob/master/src/ossia/dataflow/safe_nodes/executor.hpp [consulté le 24/11/2020]

Pour citer ce document:
Jean-Michaël Celerier, Â« Architecture logicielle du projet ossia Â», *RFIM* [En ligne], Numéros, n° 7-8 - Culture du code, Mis Ã jour le 04/01/2021
URL: http://revues.mshparisnord.org/rfim/index.php?id=574