Maîtrise Génie Mathématique et Informatique
Université Paris IX Dauphine
Rapport de stage
Avril – septembre 2003

Code : 3/2003/F/LAB

Intégration de traitements video dans jMax

Carl Seleborg

Maître de stage : François Déchelle

jMax — video — traitement du signal — temps-réel — logiciels libres

Intégration de traitements video dans jMax

Rapport de stage
28 avril – 5 septembre 2003
82 pages

Ircam - Centre Georges Pompidou
1, place Igor-Stravinsky
75004 Paris
France
Responsable de stage : François Déchelle
Tel : 01 44 78 48 21
E-mail : Francois.Dechelle@ircam.fr

Mes remerciements à François Déchelle, Patrice Tisserand, Norbert Schnell, Riccardo Borghesi, Diemo Schwarz, Sylvie Benoît, Didier Périni et Florence Quilliard pour leur patience et leur aide durant ce stage.

Table des matières

Table des figures

Introduction

Ce rapport, présente le travail effectué au sein de l'Ircam (Institut de Recherche et de Coordination Acoustique/Musique) durant mon stage de maîtrise d'IUP Génie Mathématiques et Informatique de l'université Paris-Dauphine. Le travail était essentiellement centré sur le développement et le portage de modules externes de traitement video pour le logiciel **jMax**. L'environnement technique du stage reposait sur les principes du développement de logiciels libres, avec tous les outils classiques pour ce type de travail (les outils GNU†).

A la fin du stage, le portage du principal module de traitement video, **VideoDSP** était terminé, du moins pour la version Linux de jMax ; les autres modules étaient également pratiquement portés. Ces modules video permettront à jMax d'élargir son plublic, et s'intégreront dans la mission du pôle "Arts du spectacle" récemment créé à l'Ircam et qui lie le son à l'image, la danse et à d'autres disciplines artistiques vivantes. Principalement développeur, j'ai eu l'occasion avant tout de découvrir de nombreux outils de traitement du son, jMax en premier, et j'ai pu rencontrer les difficultés liées aux systèmes temps-réel†, au traitement du signal et au développement de logiciels libres en collaboration avec des développeurs du monde entier.

Ce rapport est divisé en quatre parties, outre l'introduction et la conclusion. Dans la première, nous présenterons l'Ircam et ses activités. Dans la seconde partie, nous parlerons du logiciel jMax et du travail effectué dessus et sur les modules de traitement video. La troisième partie sera un approfondissement de l'étude du moteur de calcul de jMax dans le but d'étudier les possibilités de le rendre indépendant des types de données traitées. Enfin, dans la quatrième et dernière partie, nous aborderons les expériences personnelles et les enseignements acquis au cours du stage. En annexe du document, le lecteur trouvera un glossaire[1] expliquant la plupart du vocabulaire technique employé dans ce rapport, une bibliographie et un index.

[1] Les termes définis dans le glossaire sont marqués d'un symbole† en forme de croix lors de leur première occurence dans le texte.

1 Présentation de l'Ircam

Cette partie présente de manière globale et succinte l'Ircam, où s'est déroulé le stage. Nous y verrons les missions de l'Ircam, son organisation, quelques détails financiers et enfin une petite liste de quelques logiciels célèbres produits à l'Ircam.

1.1 Mission de l'Ircam

L'Institut de Recherche et Coordination Acoustique/Musique est officiellement né en 1969 lorsque M. Georges Pompidou, alors Président de la République, demande à M. Pierre Boulez, compositeur contemporain de renom, de créer et diriger un institut musical rattaché au futur Centre National d'Art Contemporain, afin que soient également représentés dans ce haut-lieu de la culture contemporaine les arts du son et de la musique.

Les activités de l'institut tournent autour de trois pôles : la recherche, la création et la transmission. Il réunit en un même lieu des scientifiques et des musiciens afin de les inciter à explorer ensemble de nouvelles voies artistiques.

Recherche. La recherche vise essentiellement à découvrir et à développer des outils mathématiques et logiciels pour donner aux compositeurs et aux musiciens de nouvelles façons de concevoir et de créer la musique. L'Ircam travaille également en partenariat avec d'autres instituts de recherche et avec des acteurs du monde industriel.

Créer. L'Ircam se veut en quelque sorte l'avant-garde de la création musicale contemporaine. De nombreux compositeurs sont invités et 20–25 œuvres y sont créées tous les ans, associant techniques classiques et contemporaines. Les œuvres sont présentées au public par l'intermédiaire du festival annuel Agora à Paris et dans le reste du monde par de nombreux ensembles, dont l'Ensemble Intercontemporain.

Transmettre. L'Ircam possède un pôle pédagogie, qui gère essentiellement deux formations. La première est un cursus de composition musicale, la seconde est une formation doctorale scientifique. En outre, de nombreux stages de formation dans le domaine de l'informatique musicale sont proposés, et la Mediathèque offre une vaste base de connaissances musicales.

1.2 Organisation

L'organisation de l'Ircam reflète largement le caractère multipolaire de sa mission. Pierre Boulez est aujourd'hui directeur honoraire de l'institut.

La figure 1 (p. 4) donne une vue d'ensemble de l'organisation de l'Ircam et de son département Recherche et Développement.

1.3 Financement de l'institut

L'Ircam reçoit la plupart de son budget de la part du ministère de la culture. Pour l'exercice 2002, l'Ircam a reçu approximativement les sommes suivantes :

Affectation	**Montant (en euros)**
Fonctionnement	4 000 000
Recherche et développement	650 000
Equipement	1 000 000

L'Ircam touche également des subventions *via* son partenariat avec le CNRS. D'autres fonds proviennent des partenariats sur des projets particuliers ou sur des contrats.

1.4 Quelques produits et logiciels

L'Ircam développe de nombreux logiciels dédiés au traitement du son et à la création musicale. Parmi ces logiciels, voici les plus connus :

jMax jMax est environnement de programmation visuelle pour l'interaction musicale temps-réel. Ce logiciel est présenté plus en détail dans le reste de ce document. Il s'agit de l'une des principales plate-formes de développement de systèmes de synthèse sonore à l'Ircam.

OpenMusic Il s'agit d'un environnement de programmation objet entièrement visuel basé sur Lisp[†]. L'ensemble de classes et bibliothèques fournies en fait un environnement particulièrement adapté pour la composition musicale assistée par ordinateur. OpenMusic est un logiciel libre, développé par l'équipe des Représentations Musicales dirigée par Gérard Assayag.

Le Spatialisateur Développé sous forme d'un ensemble de patches et objets MAX (cf. figure 2, p. 7), le Spatialisateur est un puissant système de spatialisation acoustique du son dans une salle de concert. Le Spatialisateur a été utilisé pour jouer de nombreuses pièces issues des studios de l'Ircam, comme par exemple le récent opéra *K...* de Philippe Manoury. Le Spatialisateur est développé par l'équipe d'Acoustique des Salles, dirigée par Olivier Warusfell.

Modalys Ce logiciel de synthèse sonore permet de modéliser des instruments virtuels en fonction de leurs caractéristiques physiques (cordes, plaques, bois, etc.) L'utilisateur décrit son instrument dans le langage Scheme, un dialecte de Lisp, et le résultat est ensuite calculé par le moteur de rendu. Mais le logiciel fournit également une interface

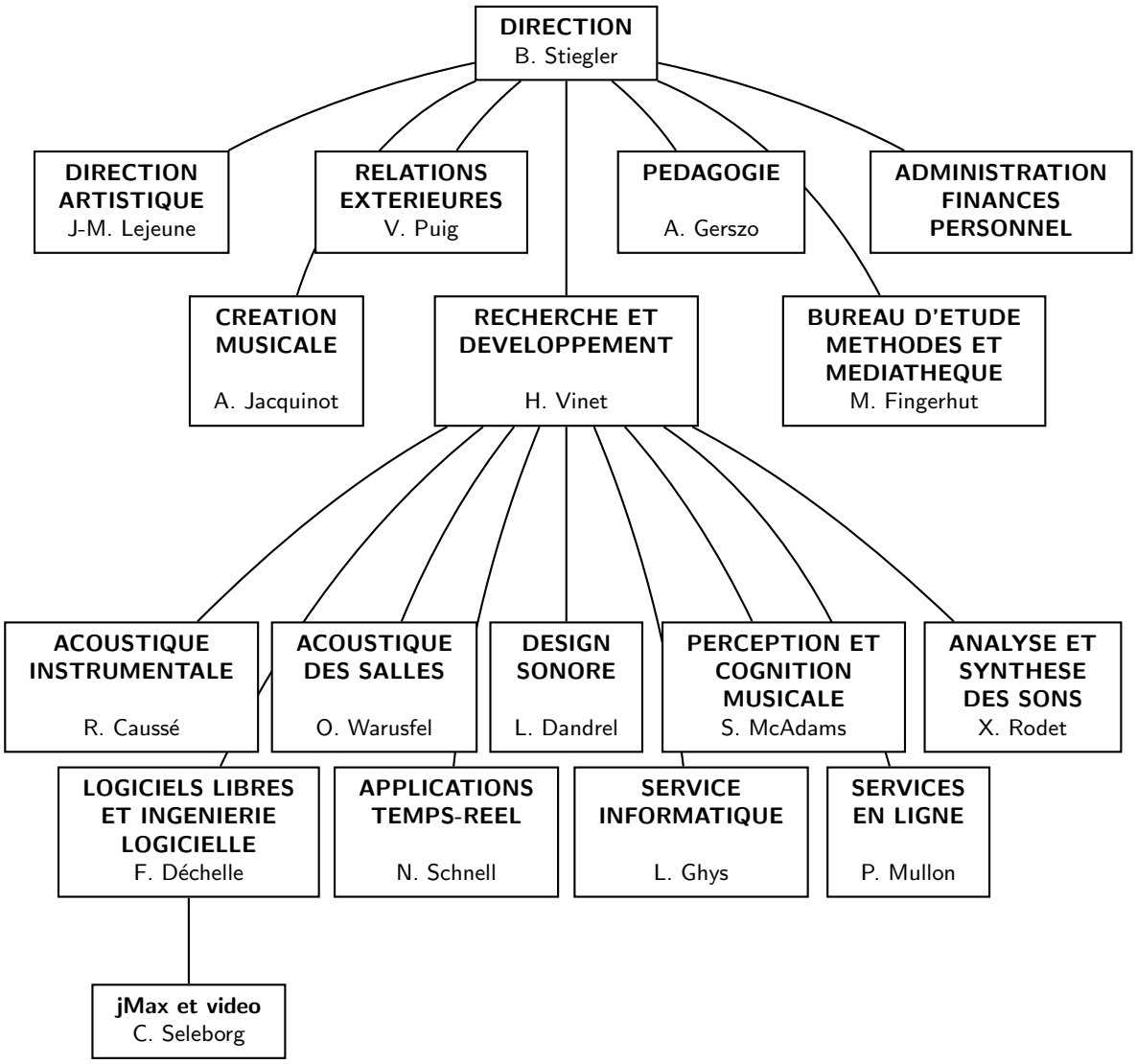


FIG. 1 – Organisation de l'Ircam et du département Recherche et Développement

graphique qui permet de s'affranchir de la programmation textuelle. Modalys a été créé par l'équipe d'Acoustique Instrumentale, dirigiée par René Caussé.

AudioSculpt Ce logiciel créé par l'équipe Analyse/Synthése (sous la direction de Xavier Rodet) permet à l'utilisateur de "sculpter" le son en modifiant visuellement les caractéristiques de ce son déterminés par une phase d'analyse préalable. Les principales applications sont la composition, le design sonore et la post-production. Le moteur de traitement sous-jacent, SuperVP (Super Vocodeur de Phase) avait été utilisé par l'équipe Analyse/Synthèse pour retravailler la voix du castrat dans le film *Farinelli*.

Diphone Créé par l'équipe Analyse/Synthèse, et utilisé dans les mêmes applications qu'AudioSculpt, Diphone Studio est utilisé pour réaliser des *morphings* sonores.

Cuidado Il s'agit d'un projet européen de création de bases de données contenant des échantillons sonores et des extraits musicaux. Le système est capable de reconnaître un échantillon sonore et de dire s'il détient ce son dans sa base, et indexe tous ses échantillons selon un certain nombre de descripteurs de plus ou moins haut niveau. Ce projet est développé dans le cadre d'une collaboration entre l'Ircam (équipes Analyse/Synthèse, Services en Ligne, Perception et Cognition Musicale) et des partenaires industriels et scientifiques (UPF MTG de Barcelone, CreamWare, Oracle, SONY CSL, ArtPages, et al.).

Le Forum Ircam Le Forum Ircam est un service proposé par l'Ircam et qui permet à ses adhérents d'avoir accès aux différents logiciels de l'institut, de recevoir de l'aide et de bénéficier de réductions pour les différents stages de formation en informatique musicale organisés régulièrement.

2 Travail réalisé — jMax et video

Cette section présente maintenant le logiciel jMax et le travail qui a été réalisé sur les extensions video, et en particulier le module VideoDSP.

2.1 Présentation de jMax

jMax est un langage de programmation visuel créé pour l'interaction musicale temps-réel. En réalité, jMax n'est que l'une des évolutions du logiciel *Patcher*, créé à l'Ircam en 1986 par Miller Puckett pour jouer la pièce musicale *Pluton* de Philippe Manoury ; ou pour être plus précis, il s'agit d'une réimplémentation logicielle de MAX/ISPW, une combinaison de MAX, le successeur de Patcher, et de l'*Ircam Signal Processing Workstation*, une architecture matérielle dédiée au traitement temps-réel du son (cf. figure 2, p. 7). jMax est un système client/serveur†, où l'interface graphique, écrite en Java†, ne fait qu'afficher les traitements qui seront effectués par FTS†, le moteur de calcul temps-réel écrit en C†. [1]

Dans cette section, nous présenterons brièvement le logiciel, ses principes et ses fonctions.

2.1.1 Le paradigme MAX

Comment apprivoiser le langage proposé par la machine, alors que bien des dimensions courantes de sa pratique échappent au compositeur et ne trouvent aucune transcription directe dans le langage musical ?
— Laurent Bayle, Directeur de l'Ircam, 1996

Le choix d'une représentation informatique des données qu'est habitué à manipuler l'utilisateur est important en général, et souvent crucial dans le domaine des applications musicales. Miller Puckett voulait, avec son système Patcher puis avec MAX, présenter de manière intuitive au musicien-utilisateur les traitements que ce dernier pouvait appliquer au son et à la musique dans le cadre d'une interprétation sur scène d'une œuvre musicale.

Le *paradigme*† retenu fut celui du "synthétiseur virtuel". Lors de la grande époque des synthétiseurs analogiques, on trouvait des appareils constitués de plusieurs modules (des oscillateurs, des effets, des égalisateurs, des sorties, etc.) que l'on reliait par des fils électriques (*patches* en anglais), définissant ainsi le parcours du signal électrique sonore. La figure 3 p. 8 en montre un exemple.

Ce paradigme, visuel, offre un moyen intuitif pour l'utilisateur de définir ses traitements [7]. Les segments de raccordement représentent des flux de données, qui peuvent être des échantillons audio ou des informations de

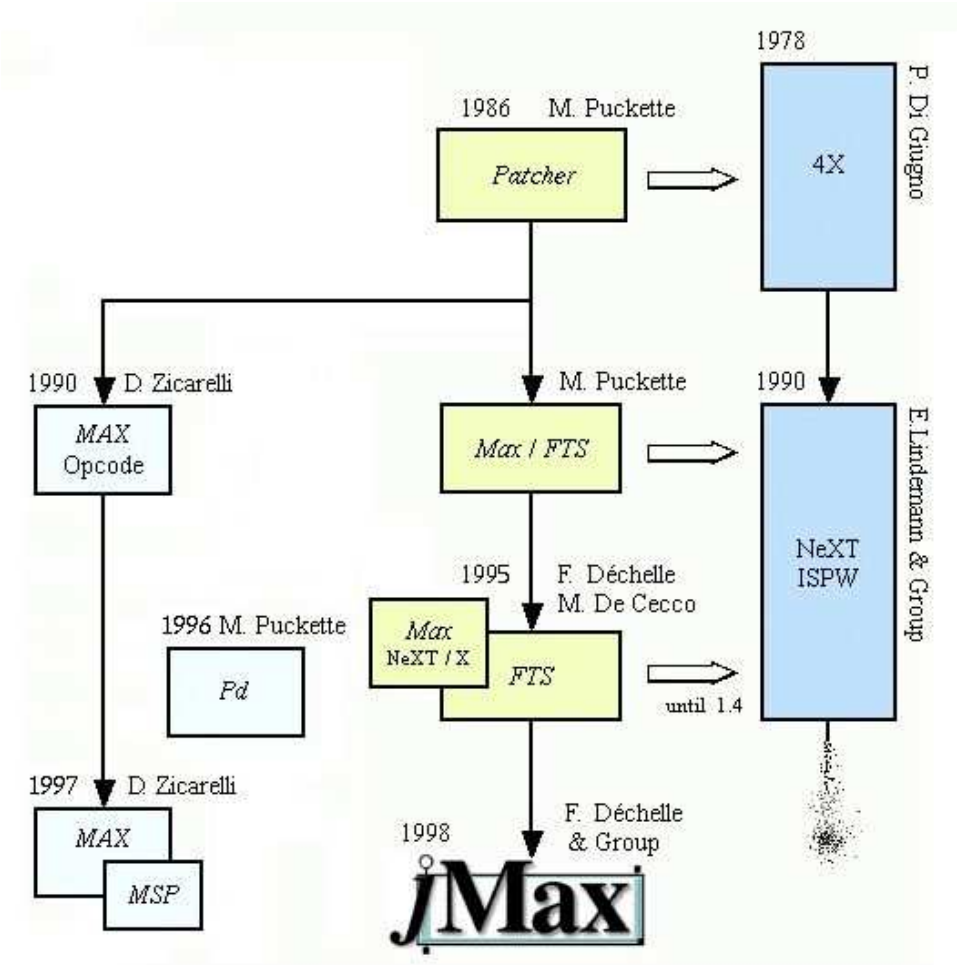


FIG. 2 – Diagramme de l'historique de jMax

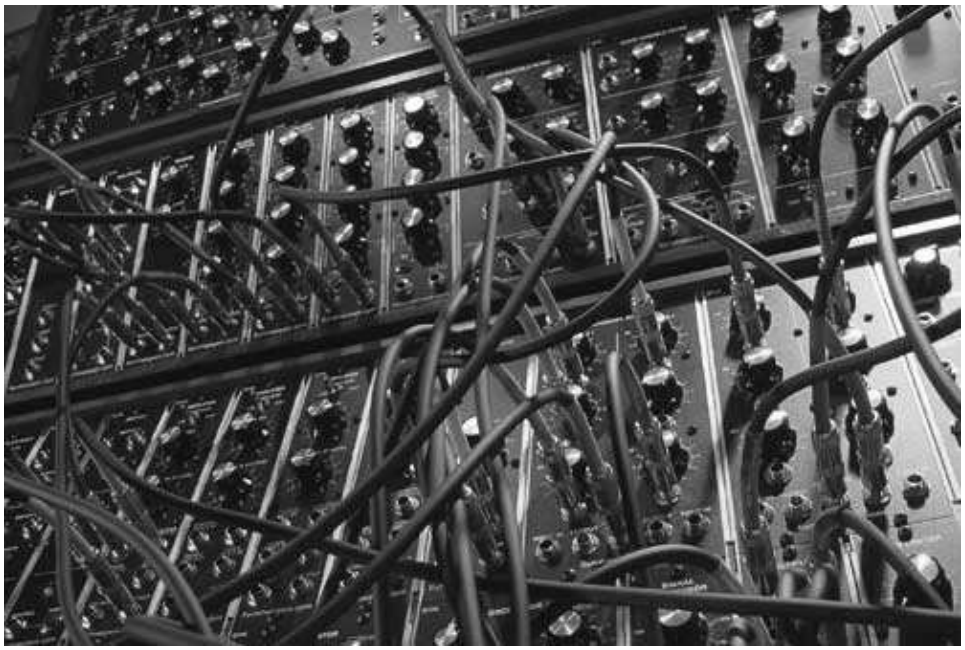


FIG. 3 – Photographie d'un synthétiseur analogique "patchable".

contrôle. L'utilisateur relie ainsi des objets entre eux : certains objets effectuent des traitements sur les signaux audio, et d'autres effectuent des calculs sur les données de contrôle ou offrent des moyens visuels de les paramétrer.

Au final, MAX (comme ses jeunes cousins PureData et jMax) est donc un environnement de programmation visuel, et les patches que créent les utilisateurs sont de véritables programmes (cf. figure 4 p. 9), dont le fonctionnement et l'interface sont mélangés à l'écran dans le cadre du paradigme du synthétiseur virtuel.

2.1.2 Principes de fonctionnement de FTS

FTS est le moteur de calcul de jMax. C'est un programme écrit en C qui est composé de plusieurs parties.

Système d'objets. FTS est muni d'un système objet, dans le sens où il sait manipuler des objets (dont il connaît l'interface, mais dont l'implémentation est fournie soit par FTS, soit par des modules externes, cf. section ci-après) et sait les faire communiquer entre eux par l'intermédiaire de messages. D'une manière générale, un objet FTS est une sorte de boîte noire possédant des ports d'entrée (les *inlets*) et des ports de sortie (les *outlets*), par lesquels l'objet reçoit ou envoie des messages (déclenchant des appels de méthodes) ou des signaux audio. Ces messages peuvent être de natures très variées, et

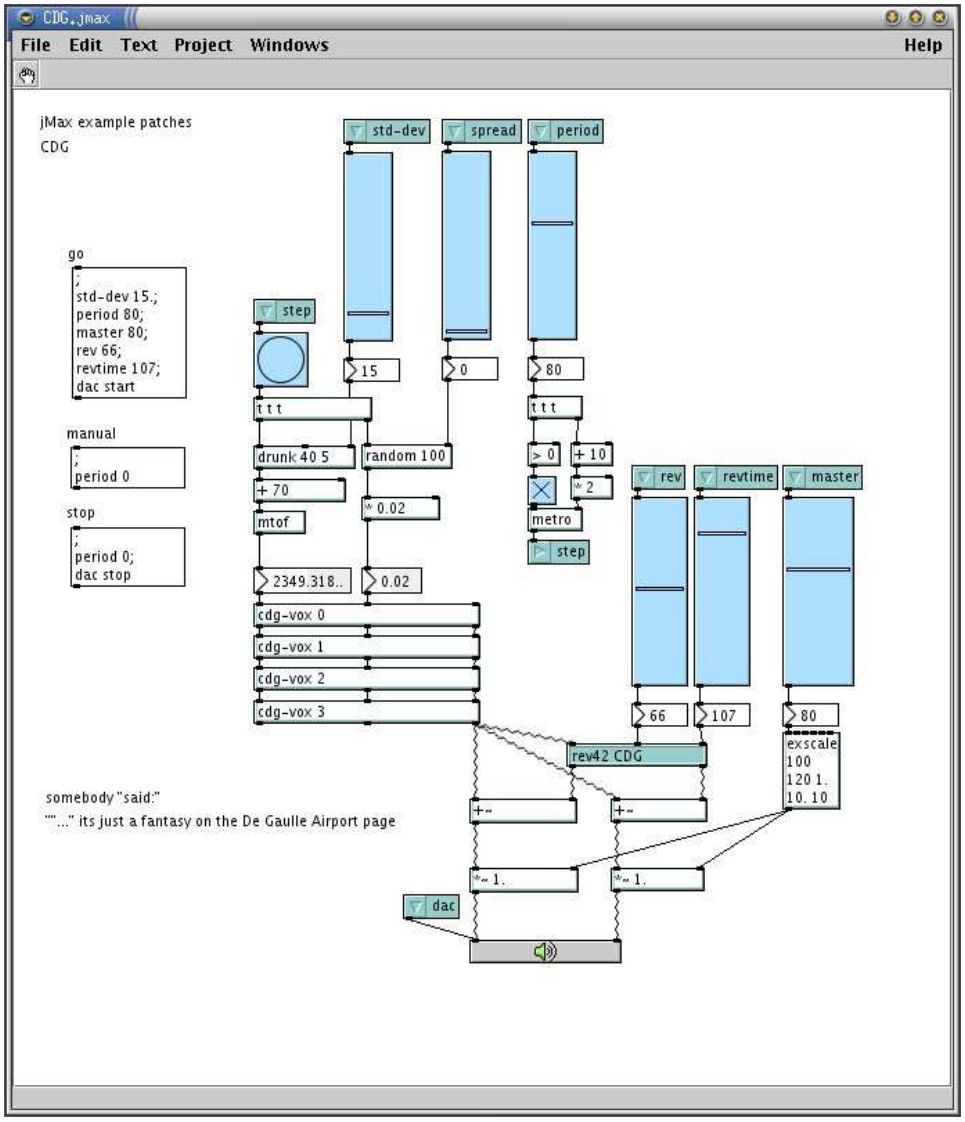


FIG. 4 – Exemple de patch jMax

peuvent représenter des nombres, des symboles, des chaînes de caractères, des vecteurs et bien d'autres types de données encore. En donnant des types aux *inlets* et aux *outlets* des objets, FTS peut contrôler que l'utilisateur ne tente pas d'établir une connexion entre deux objets qui n'aurait aucun sens, comme par exemple de relier une sortie qui envoie des nombres à une entrée censée recevoir un flux audio.

Ramasse-miettes temps-réel. Les données manipulées par FTS, et créées suite à l'exécution d'un programme (ou patch), doivent être libérées pour ne pas saturer la mémoire de la machine. Ce processus est automatiquement géré par FTS par le biais d'un ramasse-miettes, ou *garbage collector* en anglais, c'est-à-dire un système qui repère les données qui ne sont plus utilisées et les détruit, rendant les ressources qu'elles mobilisaient disponibles pour le système. Un travail important à été fourni pour faire de ce ramasse-miettes un mécanisme respectant les contraintes de l'exécution en temps-réel.

Couche d'abstraction matérielle. FTS est une version logicielle d'une architecture matérielle dédiée au traitement du signal développée à l'Ircam, nommée $4X$. Pour le projet de l'*Ircam Signal Processing Workstation*, FTS était responsable du traitement du son. Cependant, dès le départ, FTS fut conçu pour être largement indépendant de l'architecture matérielle sur laquelle il était exécuté, et dans cette optique, ses concepteurs l'ont doté d'une couche d'abstraction matérielle, afin de limiter les portions de code dépendantes de ce matériel. Cette conception s'est révélée judicieuse, et a permis de porter FTS sur Windows en moins d'un mois, alors qu'une version de MAX pour Windows était en préparation depuis plusieurs mois déjà.

Chaîne de compilation DSP. FTS traite le signal sous forme de chaîne de traitements DSP (*Digital Signal Processing*). Ces traitements sont organisés par l'utilisateur dans des patches. Comme tous les langages de programmation, FTS utilise des techniques de compilation pour augmenter les performances lors du traitement effectif du son. Il est donc muni d'un compilateur et d'une petite machine virtuelle capable d'effectuer les traitements de l'utilisateur, compilés en un programme linéaire.

2.1.3 Les modules de jMax

Les objets que l'utilisateur manipule dans jMax sont implémentés dans des modules externes, chargés par FTS lors du démarrage (l'article [4] donne plus de précisions sur la création de modules de ce type).

Ces modules, pour la plupart, se contentent de déclarer et implémenter des classes d'objets FTS. Certains, cependants, apportent plus de fonctionnalité que cela. C'est le cas, par exemple, des modules d'interface avec des périphériques de son (jMax utilise les pilotes ALSA[†] sous Linux ou

DirectSound† sous Windows, et charge l'un ou l'autre module en fonction de la plate-forme). C'est également le cas pour le module VideoDSP dont nous reparlerons plus loin dans ce rapport et qui intègre les flux video au sein de FTS.

Presque tout l'objet du stage à été de développer ou de porter des modules externes, capables d'intégrer le traitement de la video dans jMax. Après avoir brièvement présenté le logiciel et évoqué son fonctionnement, nous allons maintenant voir plus en détail le travail que j'ai effectué durant le stage.

2.2 La video dans jMax : motivations et difficultés

A l'origine, jMax était destiné à être utilisé pour développer des applications interactives pour le traitement temps-réel du son. Depuis déjà quelques années, les ordinateurs personnels ont suffisamment de puissance de calcul pour ne plus se trouver saturés par des manipulations trop complexes du son, et ceci à contribué à démocratiser l'utilisation du logiciel.

Mais ce n'est que très récemment, et nous n'en sommes encore qu'aux prémisses, que les ordinateurs sont devenus suffisamment puissants pour traiter la video en temps-réel. En effet, un flux video est substantiellement plus important qu'un flux audio, et une seule image de dimensions modestes peut demander plus de mémoire pour sa représentation numérique qu'un petit son enregistré en qualité élevée.

Quelques utilisateurs de jMax ont ainsi voulu explorer l'utilisation de ce système pour la création video temps-réel, et ont développé des extensions, sous forme de modules, pour rendre cela possible. Trois voies différentes ont été explorées pour y parvenir.

GridFlow. Le package *GridFlow*, développé par Mathieu Bouchard, est essentiellement une bibliothèque de calcul matriciel En effet, dans la philosophie de GridFlow, les images ne sont que des matrices (ou des n-uplets de matrices selon qu'on les décompose par canaux ou non). L'idée était d'utiliser le mécanisme de messagerie de jMax pour faire circuler des références à ces matrices à travers des objets GridFlow. Ainsi, FTS ne sait pas vraiment ce qu'il transporte, car il ne transporte que des nombres ; à l'intérieur de GridFlow se trouve des mécanismes pour stocker ces matrices.

DIPS. Les concepteurs de *DIPS*† ont songé à utiliser jMax pour mettre en œuvre de la synthèse 3D grâce à OpenGL†, une bibliothèque graphique 3D très puissante et très populaire. OpenGL fonctionne avec des *display lists*, c'est-à-dire avec des listes d'objets 3D composants une scène. Transposé à jMax, ces *display lists* deviennent des chaînes d'objets DIPS, chaque objet représentant un objet OpenGL avec ses paramètres. Là encore, le principe

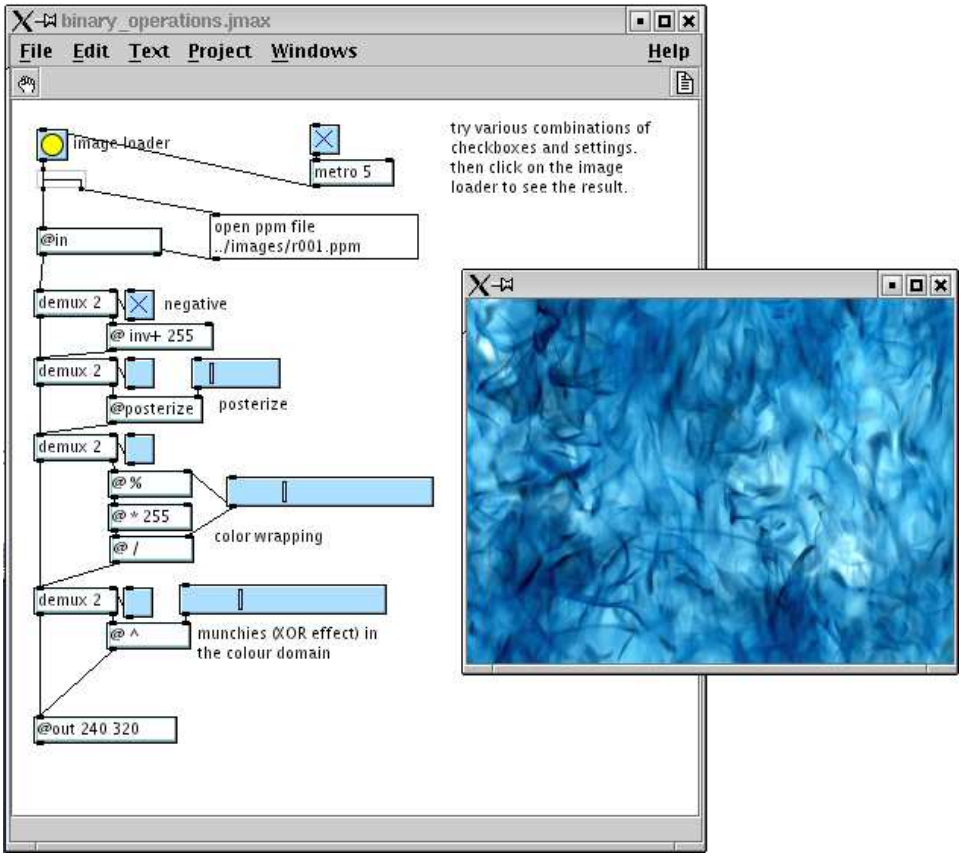


FIG. 5 – Exemple de patch GridFlow

repose sur l'utilisation des messages de contrôle de jMax, la gestion des données internes étant confiée au module et à la bibliothèque graphique.

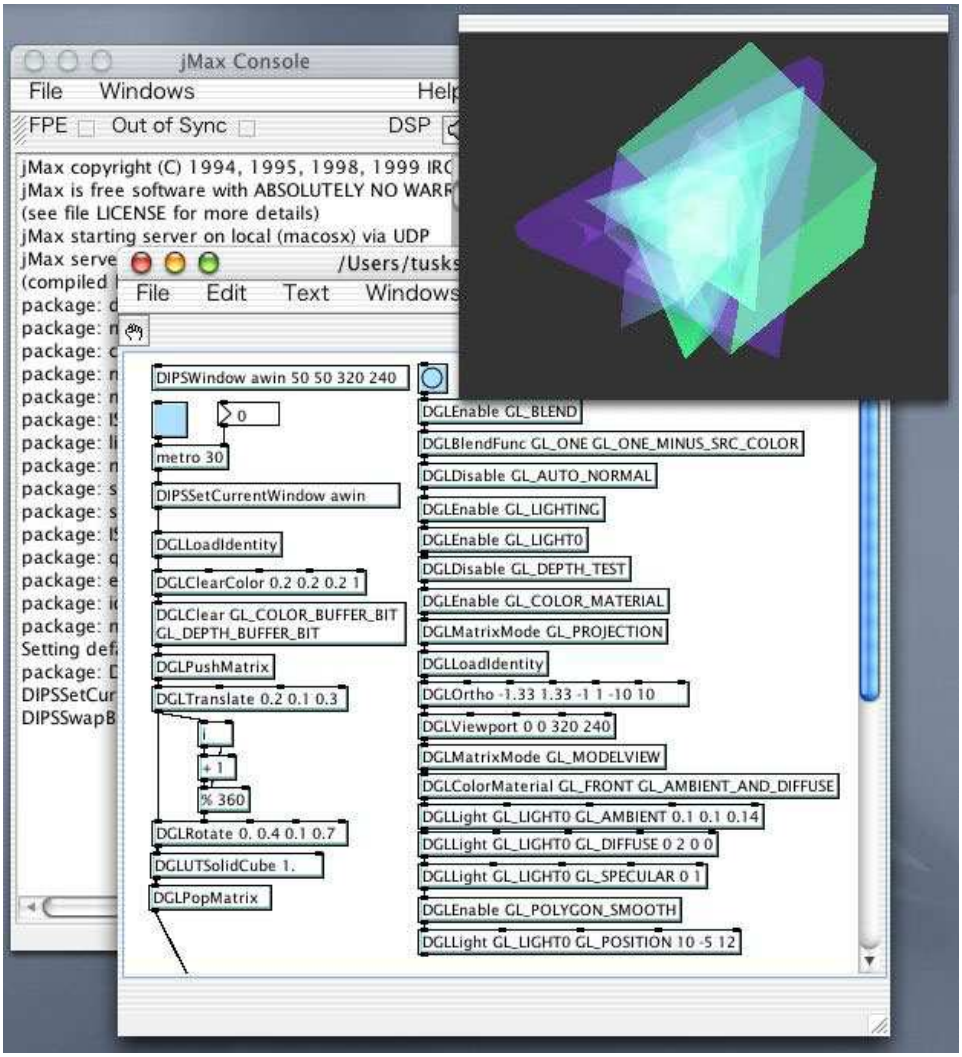


FIG. 6 – Exemple de patch DIPS

VideoDSP. *VideoDSP* est le module de Christian Klippel. Il s'agit, cette fois, d'un véritable effort d'intégration de traitement de flux video au cœur de FTS. Le module, en reprenant pour une large part le code originel de FTS, simule une seconde chaîne de compilation DSP pour la video (cf. paragraphe 3.1, p. 33), parallèle à la chaîne DSP dédiée au son déjà implantée dans FTS. Ainsi, c'est jMax qui gère la transmission des pixels des images d'un objet à l'autre, ce qui est très proche de ce qui se passe avec les échantillons sonores.

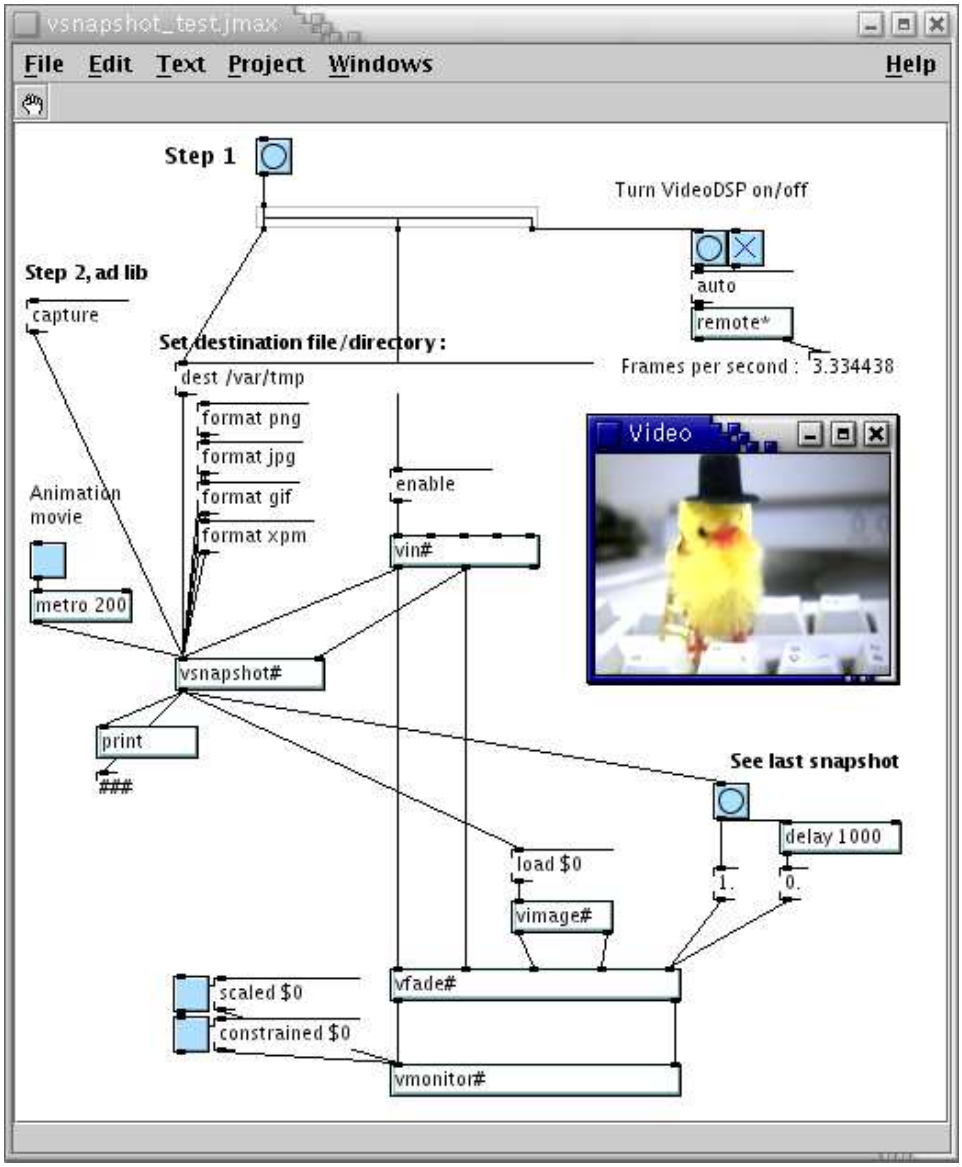


FIG. 7 – Exemple de patch VideoDSP

2.3 Portage des modules video

L'objet du stage a été de participer au portage (c'est-à-dire à la mise à jour du code source) des extensions video présentées au paragraphe précédent. En effet, ces extensions existaient pour la version 2.5 de jMax. Mais l'année dernière, l'équipe Temps-Réel de l'Ircam a sorti la version 4.0 de jMax, ce qui a introduit un certain nombre d'incompatibilités vis-à-vis des modules existants. Il a donc dans un premier fallu traduire ces modules pour les faire fonctionner à nouveau avec cette nouvelle version de jMax.

Automatisation du portage. L'une des premières tâches que j'ai effectuées au début du stage fut de créer un programme, écrit en Perl[†], capable de parcourir le code sources d'un module jMax et de corriger et mettre à jour le code du module pour jMax 4.0. Lorsque le programme ne peux pas corriger directement le code du module, il indique par un commentaire dans le code ce qu'il faut faire. Dans tous les cas, là où le programme fait une modification, il insère un commentaire à cet endroit du code pour indiquer ce qu'il a modifié. Ainsi, s'il fait des erreurs, un programmeur humain pourra facilement les corriger.

Ce programme, affectueusement baptisé `autoport` par l'auteur, a été réutilisé avec succès par d'autres équipes de l'Ircam dans leur travail de portage de divers modules jMax et est maintenant fourni avec les sources du logiciel, afin que les développeurs ayant créé leurs propres modules jMax puissent voir leur travail de portage facilité.

Le code source d'`autoport` est donné en annexe de ce document.

2.4 Le travail sur VideoDSP

Nous allons maintenant aborder la description de mon travail de portage, et notamment du module VideoDSP. En effet, c'est sur ce module que je me suis presque exclusivement concentré, le portage de DIPS ayant déjà été entammé et presque fini (il s'agit en fait d'une très mince enveloppe autour d'OpenGL) par Patrice Tisserand et celui de GridFlow, après avoir nécessitant une modification de l'analyseur lexical de jMax sur lequel François Déchelle a travaillé peu de temps avant la fin du stage, ayant finalement été assuré par Mathieu Bouchard[2].

Portage préliminaire. Porter l'ensemble du module VideoDSP aurait été une tâche trop longue, eu égard à l'objectif fixé pour le stage, car il réimplémente un grand nombre d'objets permettant de contrôler les flux video

[2]Il faut préciser en plus que GridFlow est presque entièrement codé en Ruby, un langage de script orienté objet, et que sa partie de code C n'est qu'une couche d'accès à Ruby. Porter GridFlow nécessitait de connaître une bonne partie Ruby. J'avais commencé le portage, mais bloqué par des problèmes avec ce langage, j'ai été "doublé" par Mathieu Bouchard.

et qui sont les répliques d'objets audio analogues fournis avec jMax. Il a donc été décidé de ne porter, dans un premier temps, que la partie essentielle du module, à savoir le compilateur de chaîne DSP et les fonctionnalités de contrôle les plus essentielles (mise en marche/arrêt des flux video, exécution image par image, etc.).

Bien sûr, `autoport` a été utilisé pour effectuer ce travail. Mais il s'est avéré que pour simuler le fonctionnement de FTS, qui en était à sa version 2.5.1 lorsque Christian Klippel à publié son module, VideoDSP utilise des fonctions de FTS qui n'existent plus dans la version 4.0, le compilateur DSP ayant subi quelques modifications lors du changement de version. Il a donc fallu réécrire une partie du code au cœur du module.

Les entrées/sorties video. Le module VideoDSP ne s'occupe que d'acheminer les signaux video à l'intérieur d'un graphe d'objets dans FTS (un *patch* peut être considéré comme un graphe dont les objets sont les sommets et dont les connexions sont les arêtes). Il ne fournit aucun objet capable de lire un signal video en entrée ni d'en afficher un en sortie.

Etienne Deleflie, un utilisateur de jMax et développeur, participe activement au développement de VideoDSP depuis longtemps et a fait quelques objets pour son usage personnel, notamment un afficheur et un lecteur de fichiers video non-compressés au format QuickTime[†].

J'ai pris la décision, en accord avec François Déchelle, de récupérer une partie de ce code déjà existant et de le compléter afin de créer un module unifié, le module *vio* (pour *Video Input/Output*), qui fournirait trois entrées de flux video et une sortie. Ce module contient cinq objets :

`vin#` (fig. 8) Cet objet lit un flux video en entrée d'un périphérique d'acquisition video (comme les *webcams* ou les cartes d'acquisition video) afin de l'injecter dans jMax sous forme de deux canaux video contenant les données Y et (U, V) des images video (cf. paragraphe 2.5.1 pour une discussion plus approfondie à propos des espaces de couleur). Cet objet n'existait pas, et je l'ai entièrement développé pour ce module. Ça a été l'occasion pour moi d'utiliser des appels systèmes très bas-niveau (la fonction `ioctl()` sous Linux permet de communiquer directement avec les pilotes de périphériques, elle est utilisée pour commander l'acquisition des images video). J'ai également mis en œuvre des techniques de programmation concurrente, en utilisant un *thread* d'exécution parallèle pour assurer un débit d'images maximum : pendant qu'une image est servie par morceaux à FTS à intervalles réguliers, l'image suivante est chargée par le pilote, son espace de couleur est converti et elle est copiée dans un espace mémoire réservé pour être à son tour servie à FTS.

`vcolour#` (fig. 9) Cet objet, à l'origine développé par Etienne Deleflie, synthétise un flux video uniforme de couleur constante à partir des trois

composantes Y, U et V (cf. paragraphe 2.5.1). Cet objet est utile pour des techniques "d'écran vert", où on substitue tous les pixels d'une certaine couleur de l'image à des pixels d'une autre image pour réaliser des incrustations d'images par exemple.

`vimage#` (fig. 10) Cet objet crée un flux video constitué d'une seule image fixe renvoyée en continu. On peut ainsi réaliser des masques d'images : d'une image en entrée, on ne conserve que les pixels correspondant à des pixels blancs sur le masque, ce qui permet de "découper" des parties d'images et faire des montages avec d'autres images. L'objet utilise *ImageMagick*, une bibliothèque exhaustive de manipulation d'images fixes, capable de reconnaître plus de 80 formats d'images différents.

`vmonitor#` (fig. 11) Cet objet affiche le flux video qu'on lui donne en entrée dans une fenêtre. Il est capable de redimensionner l'image affichée aux dimensions de la fenêtre. Pour effectuer l'affichage, l'objet utilise des techniques de *frame overlay*, où l'on passe l'image directement à la mémoire de la carte graphique afin d'améliorer les performances. A l'origine basé sur un objet similaire écrit par Etienne Deleflie, l'objet `vmonitor#` a été très largement réécrit et j'ai créé une couche d'abstraction permettant de le réimplémenter facilement en utilisant d'autres technologies. Sous Linux, avec un serveur graphique X (c'est le nom du système de fenêtrage d'UNIX), l'objet utilise la bibliothèque *XVideo* et l'extension *XSHM* (*X Shared Memory*) pour réaliser l'overlay. Sous Windows, une première implémentation a été écrite grâce à la bibliothèque SDL (cf. paragraphe 2.5.3, p. 28), mais une autre implémentation serait envisageable avec DirectX[3]. La technique mise en œuvre pour réaliser cette couche d'abstraction repose sur le principe des fonctions virtuelles dans l'héritage de classes proposé par le C++, mais implémenté en C.

`vsnapshot#` (fig. 12) L'objet permet de sauvegarder sous forme de fichiers d'images sur le disque des "instantanés" du flux video à certains moments précis. Là encore, la bibliothèque ImageMagick est utilisée pour permettre à l'utilisateur de choisir son format de fichier préféré (et pour permettre à l'auteur de s'affranchir des détails de codage des différents formats).

VideoDSP contient également un autre objet, mais qui ne fait pas partie du package *vio* en ce qu'il ne constitue pas vraiment une entrée/sortie mais plutôt un traitement video : l'objet `vfade#`. Il s'agit d'un objet capable de mélanger deux flux en entrée selon un certain coefficient et de produire en sortie le flux video résultant du mélange (cf. figure 10). En utilisant quelques objets simples de jMax, on peut faire un effet de "fondu enchaîné" grâce à cet objet. La figure 7 (p. 14) montre un autre exemple d'utilisation cet

[3]Voir DirectSound dans le glossaire.

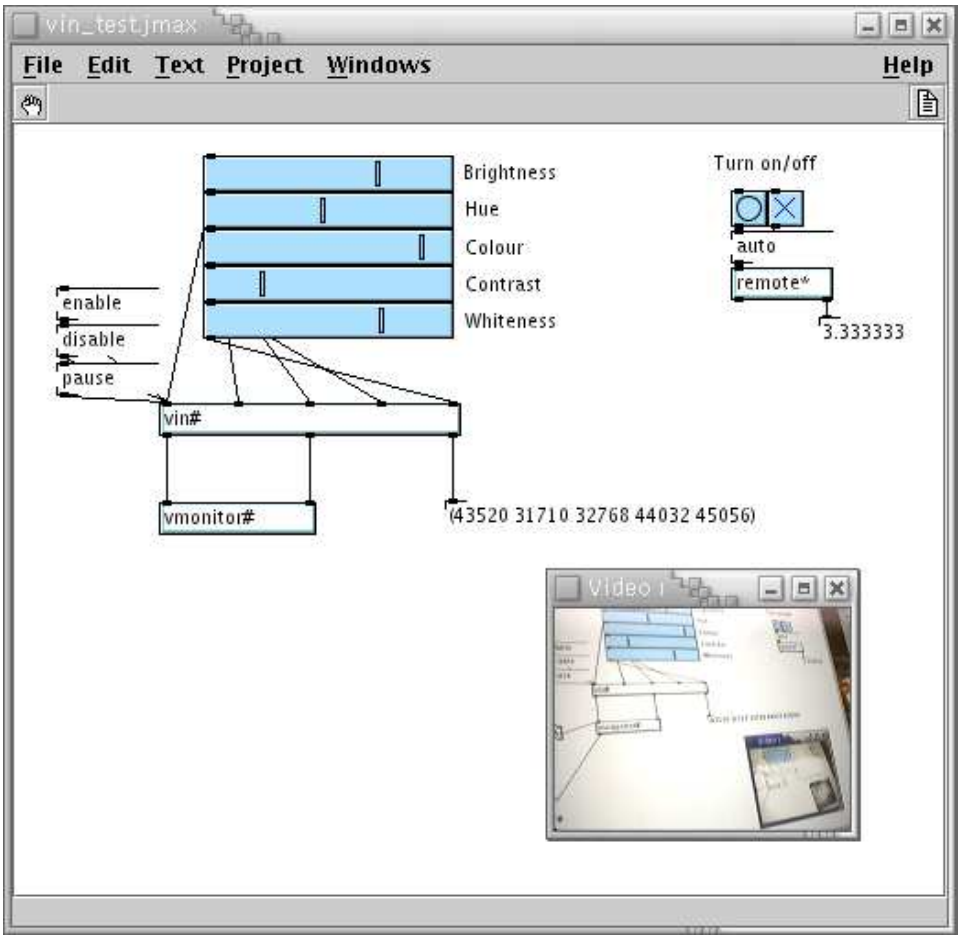


FIG. 8 – Exemple d'utilisation de l'objet `vin#`

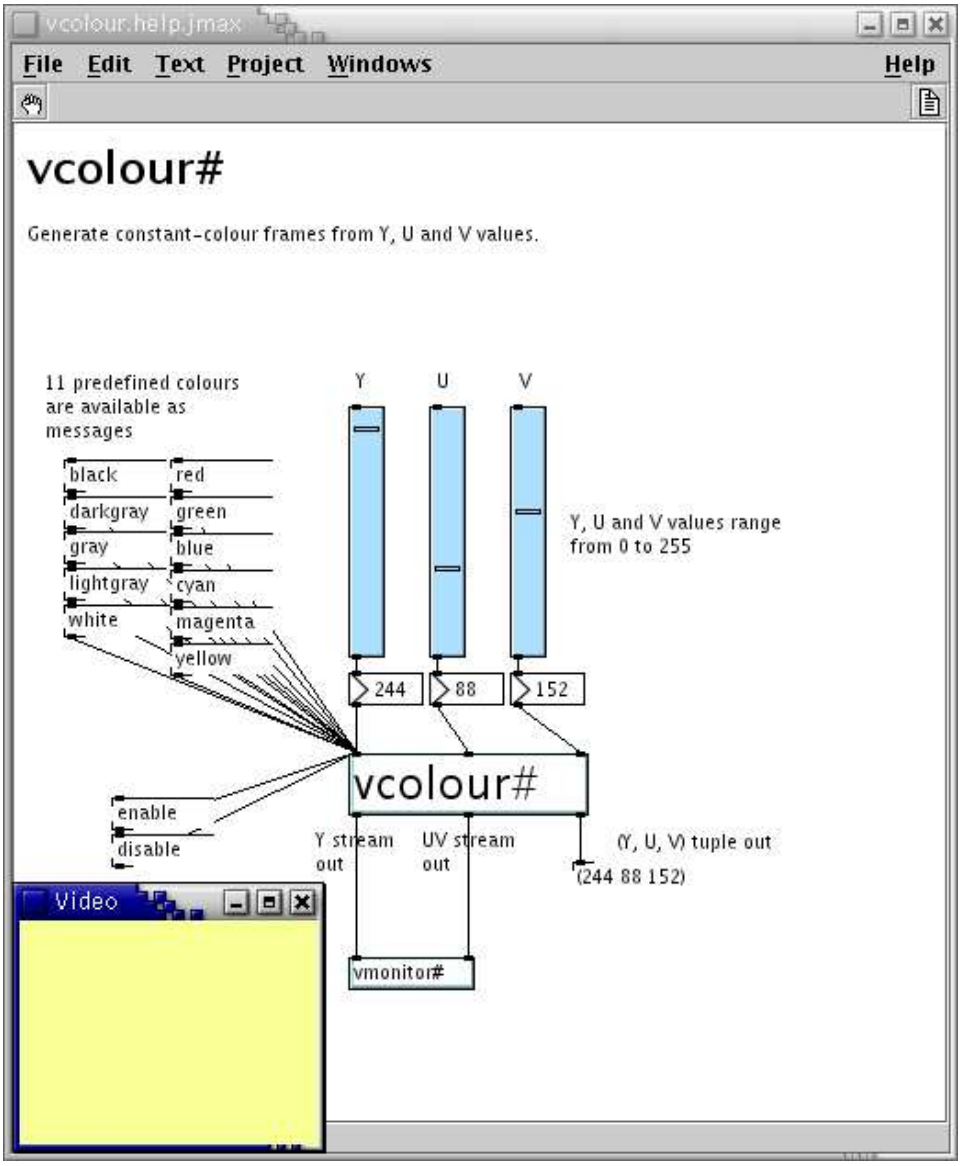


FIG. 9 – Exemple d'utilisation de l'objet `vcolour#`

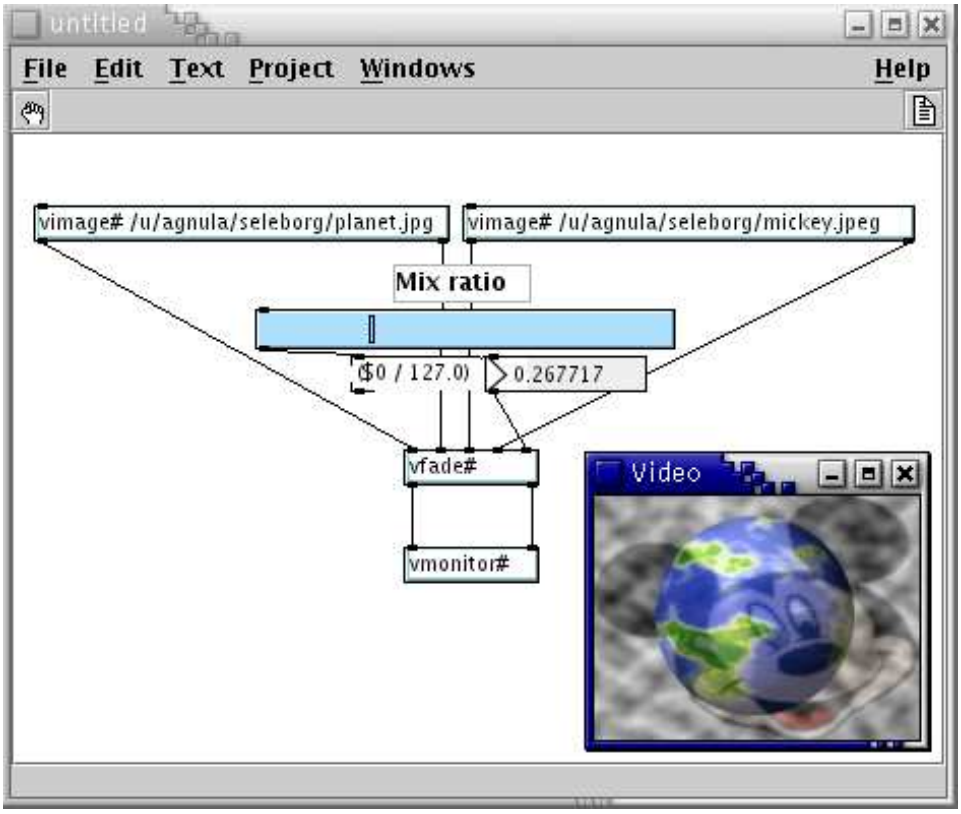


FIG. 10 – Exemple d'utilisation de l'objet `vimage#`

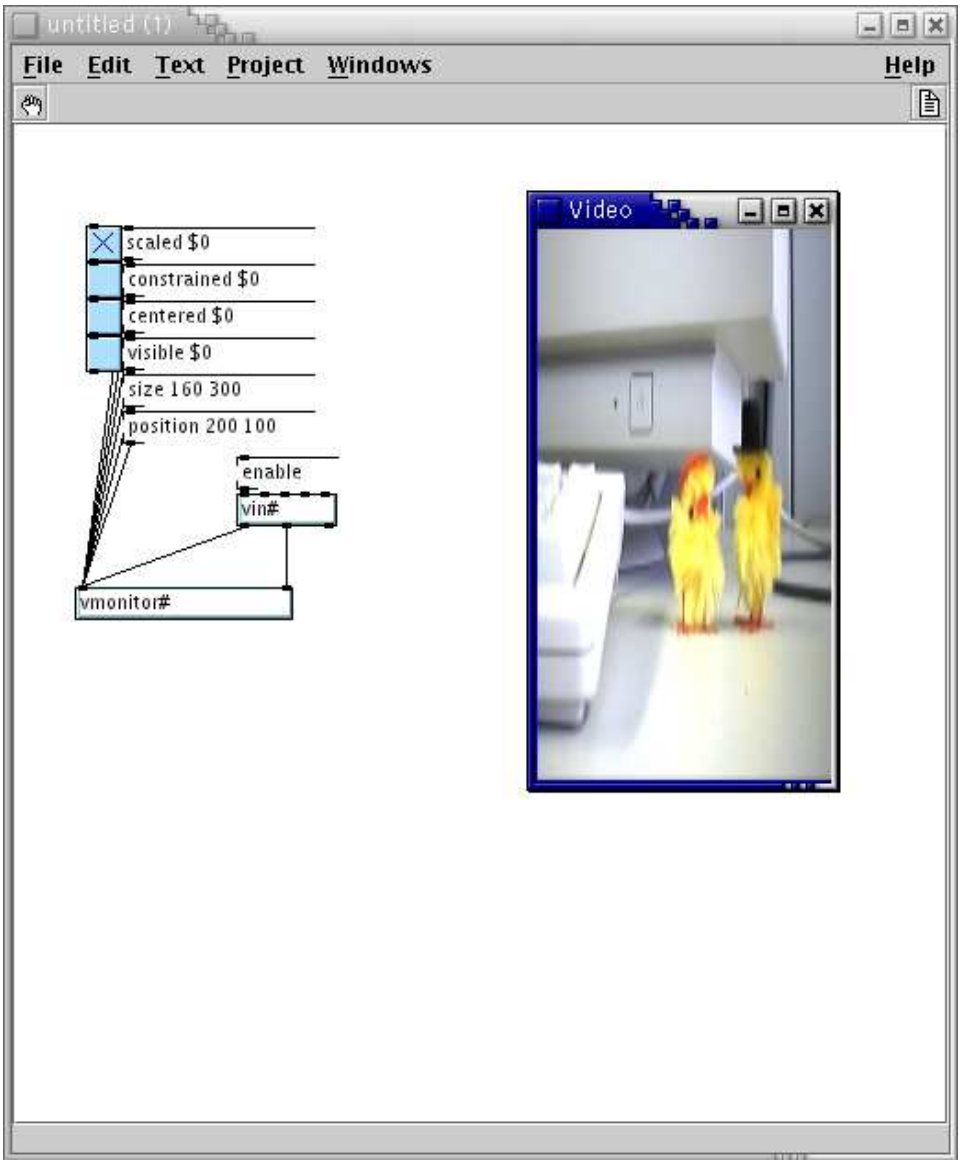


FIG. 11 – Exemple d'utilisation de l'objet `vmonitor#`

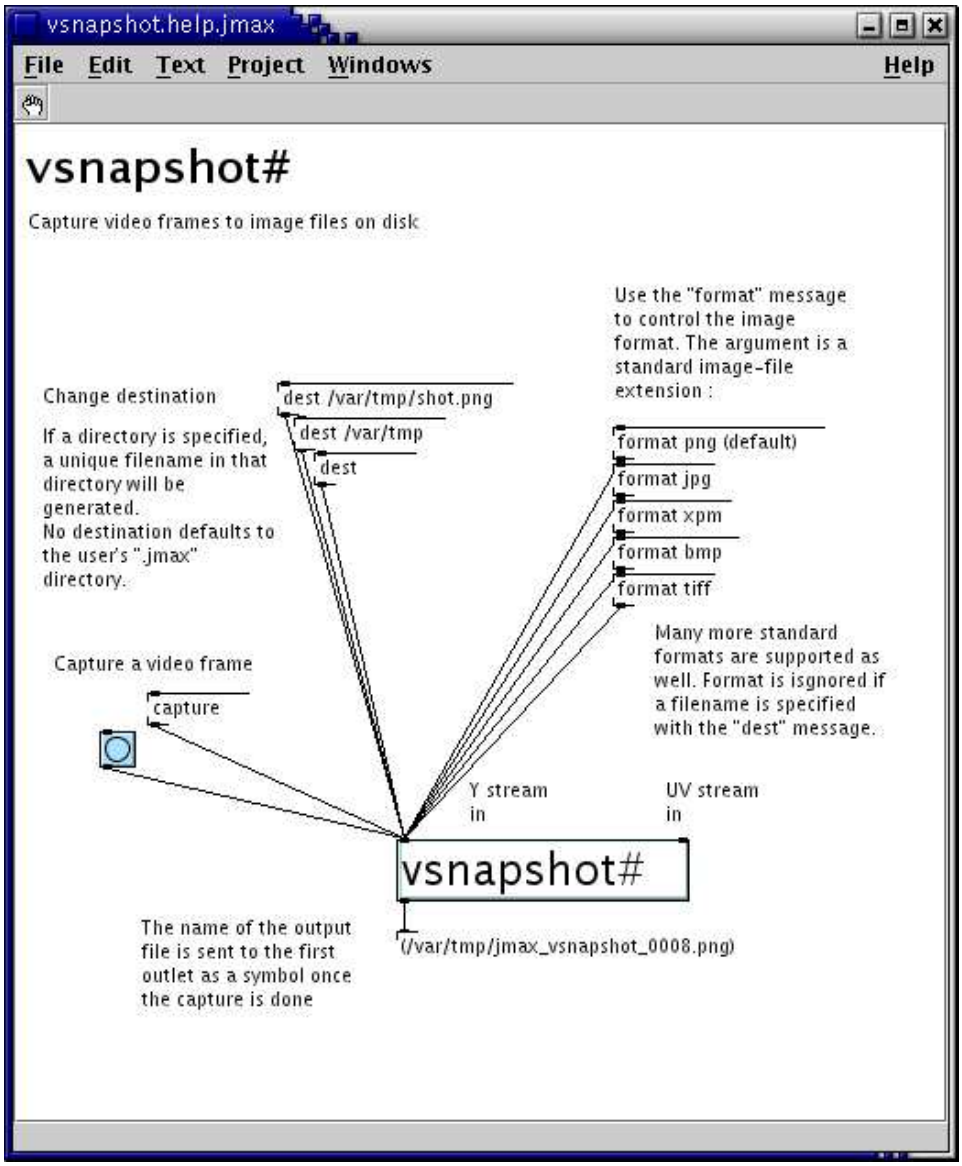


FIG. 12 – Exemple d'utilisation de l'objet `vsnapshot#`

objet : un “photomaton”. Dans cet exemple, l'objet `vfade#` ne laisse passer par défaut que le flux venant du `vin#`. Mais lorsque l'utilisateur clique sur le message *capture*, l'objet `vsnapshot#` prend la “photo” et la sauvegarde dans un fichier ; le nom du fichier est envoyé à l'objet `vimage#` qui charge l'image aussitôt, et enfin l'objet `vfade#` “bascule” pour ne montrer que l'image envoyée par le `vimage#`. L'objet `delay`, déclenché au moment de la photo, provoque au bout de quelques secondes le rebasculement du `vfade#` vers le `vin#`. L'effet final est que dans la fenêtre d'affichage, on voit ce qu'envoie la webcam ; lorsqu'on clique sur le message *capture*, l'image se fige quelques secondes pour qu'on puisse voir la photo, puis l'affichage de la webcam reprend et on peut réajuster la position du poussin ou changer le décor pour prendre la prochaine photo.

Patches d'aide. Dans jMax, on peut associer à chaque objet un patch d'aide, qui donne à la fois la documentation de l'objet et un ou plusieurs exemples d'utilisation. Tous les objets du module *vio* sont ainsi documentés (en anglais) par des patches d'aide, qui apparaissent lorsqu'on clique sur l'objet avec le bouton du milieu de la souris (ou en cliquant sur la molette pour les souris qui n'ont que deux boutons et une molette).

2.5 Quelques points difficiles

Sur le plan technique, ce stage à été l'occasion de relever certains défis auxquels je n'avais jamais confronté avant, notamment concernant les espaces de couleur.

2.5.1 La conversion d'espace de couleur

La plupart des informations sur les espaces de couleur et le sous-échantillonnage proviennent de [8].

Les espaces de couleur. Lorsqu'on souhaite représenter numériquement une image, on la représente habituellement pixel par pixel (une image étant considérée comme étant une matrice bidimensionnelle de pixels) et on écrit les pixels les uns après les autres ; tout le problème consiste à savoir comment attribuer une valeur à chaque pixel. Pour des images en échelle de gris, par exemple, c'est assez simple : on définit une plage de valeurs pour chaque pixel, par exemple 0–255, et on considère que chaque valeur représente une intensité de gris, 255 représentant le blanc, 0 le noir, l'intensité variant linéairement avec la valeur. Dans ce cas, une image (fixe) peut être considérée comme un signal bidimensionnel (abscisse et ordonnée pour chaque pixel) à un seul canal (l'intensité de gris).

En revanche, pour les images de couleur, c'est plus difficile, car il existe de nombreuses façons de synthétiser les couleurs, chacune ayant ses avan-

tages et ses inconvénients. Mais d'une manière générale, on délimite un pavé inclus dans un *espace vectoriel* dont les composantes ont une signification particulière dans le mode de synthèse qu'on utilise. Pour un espace à trois dimensions, l'image devient alors un signal bidimensionnel à trois canaux, puisque chaque pixel est caractérisé par deux coordonnées et trois valeurs. Numériquement, cela signifie qu'un pixel est donc représenté par autant de valeurs numériques que l'espace choisi comporte de composantes. Chaque appareil (ou organe, comme les yeux) capable de traiter des informations visuelles a un espace de couleur qui lui est plus particulièrement adapté. Nous allons donner maintenant plusieurs exemples d'espaces de couleur courants dans le domaine de l'imagerie informatique.

Quelques espaces de couleur classiques. En imagerie fixe, l'espace de couleur le plus classique est l'espace **RGB**, dont le nom indique les trois composantes : *Red* (rouge), *Green* (vert) et *Blue* (bleu). Cet espace correspond à une synthèse additive de couleurs, c'est-à-dire que chaque couleur est une somme pondérée de chaque composante. Ainsi, pour obtenir du jaune, il faut additionner du rouge et du vert, dont les quantités respectives varient en fonction de la nuance de couleur qu'on souhaite obtenir (plus les composantes sont faibles, plus l'image est sombre). Si on décide d'utiliser 256 valeurs différentes pour chaque composante, on peut représenter numériquement une image par des pixels de 24 bits. Cet espace de couleur est très facile à manipuler, et est très intuitif : on peut facilement trouver des valeurs proches d'une couleur qu'on cherche à obtenir. En revanche, il n'est pas très adapté pour la video car les opérations qu'on recherche en video sont parfois maladroites en RGB. C'est en RGB que la plupart des moniteurs reçoivent leurs informations.

L'espace de couleur **HSL** (pour *Hue, Saturation, Lightness*) tente de mimer la façon dont nous percevons les couleurs naturellement. De ce point de vue, il est très intuitif. En réalité, HSL est un ensemble de différents espaces de couleur, tous étant une transformation linéaire du modèle RGB.

Le modèle **CMYK** (*Cyan, Magenta, Yellow, Black*) s'utilise par synthèse soustractive de couleur, principalement par les imprimantes.

YUV est l'espace de couleur utilisé par les objets du module *vio*. Le vrai nom de YUV est *YCbCr*, et il fait la distinction entre "luma" et "chroma", c'est-à-dire qu'il sépare la luminosité de la couleur. Ainsi, Y représente la luminosité du pixel, et sa couleur est déterminée par U et V. YUV est l'espace de couleur utilisé pour la transmission d'images de télévision et on l'utilise également souvent dans les applications de compression et de traitement numérique de la video. Il avait à une certaine époque l'avantage que si une télévision était encore noir-et-blanc, elle pouvait se contenter de récupérer les données Y, qui fonctionnent alors comme une intensité de gris, et laisser de côté les informations de couleur.

YUV et le sous-échantillonnage. La principale difficulté rencontrée par rapport aux espaces de couleur fut la conversion entre les différentes versions de YUV. En effet, YUV est un espace de couleur qui se prête très bien au *sous-échantillonnage*, afin d'obtenir une certaine compression des flux video sans trop surcharger le processeur.

Dans le modèle **YUY2**[4] par exemple, deux pixels successifs partagent les valeurs de U et de V, et les deux pixels s'écrivent donc $(Y_1, U), (Y_2, V)$, ce qui signifie que si on code chaque composante sur 8 bits, on n'utilise plus que 16 bits par pixel au lieu des 24 originaux. On dit dans ce cas que les échantillons de U et V ont une periode d'échantillonnage horizontale de 2. La plupart du temps, il est très difficile à l'œil nu de faire la différence entre une image où les pixels sont codés sur 24 bits et sa contrepartie ainsi sous-échantillonnée. C'est avec ce modèle que les objets du module *vio* fonctionnent : ils utilisent deux canaux, l'un pour les valeurs de Y, et l'autre pour les valeurs alternatives de U et de V.

On peut pousser le sous-échantillonnage plus loin en utilisant les mêmes valeurs de U et de V sur deux lignes consécutives d'une image. Ainsi, pour quatre pixels, on utilise quatre valeurs de Y, et seulement une de U et une de V ; on a alors des fréquences d'échantillonnage horizontale *et* verticale de 2 pour U et V.

Les différents espaces de couleur de VideoDSP. La difficulté vient lorsque, avec l'aide d'une bien maigre documentation, il faut parvenir à convertir entre les espaces de couleur et en particulier entre les différentes versions de YUV. En effet, à l'intérieur de VideoDSP, plusieurs modèles de couleurs se côtoient :

- Le périphérique d'entrée video (une webcam dans mon cas) fournit une image en YUV420p, c'est-à-dire en YUV, avec des périodes de sous-échantillonnage horizontal et vertical de 2, et le "p" signifiant *planaire*, c'est-à-dire que les trois ensembles de valeurs Y, U et V se trouvent séparés en mémoire dans cet ordre. Ainsi, pour une image de $160 \times 120 = 19200$ pixels, on trouve d'abord 19200 valeurs de Y (toutes les valeurs sont codées sur 8 bits, soit un octet), puis 4800 valeurs de U et enfin 4800 valeurs de V.
- Il faut ensuite convertir cela en un YUV422 utilisé en interne par les objets du module *vio*, à savoir un canal de 19200 valeurs de Y, et un second canal de 9600 valeurs de U entrelacées avec 9600 valeurs de V. Or, VideoDSP, à chaque *tick* d'horloge, ne permet aux objets que de transmettre ou de recevoir qu'une partie de l'image courante (les tampons d'entrée/sortie des objets mesurant 8192 octets, une image entière est donc traitée en trois ticks d'horloge). Il faut donc, à chaque tick d'horloge, se repérer dans l'image courante, repérer le début des

[4] Le code "YUY2" est en réalité un code FOURCC, cf. paragraphe 3.3, p. 41).

pixels du morceau courant dans l'image fournie par le périphérique d'entrée, et ce dans les trois plages de valeurs (on parle de *planes* en anglais, d'où l'adjectif *planar* pour qualifier ce mode), puis les réécrire correctement dans les tampons de sortie de l'objet `vin#` afin de les envoyer aux autres objets. Le tout doit être effectué le plus rapidement possible, afin de respecter les contraintes de temps-réel.

– L'objet `vmonitor#`, dans son implémentation pour UNIX, lui, écrit les image dans la mémoire de la carte video par le mécanisme d'*overlay* décrit plus haut. Or, la carte video, elle, s'attend à recevoir une image en YUY2, c'est-à-dire le même format que celui utilisé en interne par VideoDSP sauf que toutes les valeurs sont entrelacées dans un même canal, comme décrit plus haut.

– La bibliothèque ImageMagick, elle, fournit les images en RGB. Là encore, il faut convertir cela en YUV avant de les injecter dans les "circuits" de VideoDSP, ou inversement (cf. paragraphe suivant).

Conversion entre RGB et YUV. Comme évoqué ci-dessus, les images fournies par la bibliothèque ImageMagick sont codées en RGB, et il faut donc les convertir en un signal bicanal YUV avant de les envoyer dans les "tuyaux".

Une rapide recherche sur Internet nous donnes les formules de conversion suivantes pour calculer les valeurs Y, U et V à partir des valeurs R, G et B d'un pixel de l'image :

$$\begin{aligned} Y &= 0.256R + 0.504G + 0.098B + 16 \\ U &= -0.148R - 0.291G + 0.439B + 128 \\ V &= 0.439R - 0.368G - 0.071B + 128. \end{aligned}$$

$$\begin{aligned} R &= \max(0, \min(255, 1.164(Y - 16) + 1.596(V - 128))) \\ G &= \max(0, \min(255, 1.164(Y - 16) - 0.813(V - 128) - 0.391(U - 128))) \\ B &= \max(0, \min(255, 1.164(Y - 16) + 2.018(U - 128))) \end{aligned}$$

Ces formules sont relativement simples à appliquer directement. Cependant, dans le flux YUV envoyé en sortie de l'objet `vimage#` (comme pour les autres objets du package *vio*), les valeurs de U et de V sont sous-échantillonnées, et deux pixels adjacents partagent les mêmes valeurs. Le calcul effectif pour convertir deux pixels RGB ((R_1, G_1, B_1) et (R_2, G_2, B_2)) en deux pixels YUV ((Y_1, U, V) et (Y_2, U, V)) devient donc :

$$\begin{aligned} Y_1 &= 0.256R_1 + 0.504G_1 + 0.098B_1 + 16 \\ Y_2 &= 0.256R_2 + 0.504G_2 + 0.098B_2 + 16 \\ U &= \frac{-0.148(R_1 + R_2) - 0.291(G_1 + G_2) + 0.439(B_1 + B_2) + 256}{2} \\ V &= \frac{0.439(R_1 + R_2) - 0.368(G_1 + G_2) - 0.071(B_1 + B_2) + 256}{2}. \end{aligned}$$

Le sous-échantillonnage se fait ainsi en moyennant les valeurs théoriques de U et de V et en les redistribuant sur deux pixels adjacents (c'est-à-dire qu'on finit par écrire la séquence d'octets suivante : $(Y_1, U), (Y_2, V)$).

Nous avons ainsi évoqué quelques difficultés liées à l'utilisation des espaces de couleur, et notamment par rapport au sous-échantillonnage et au calcul des positions des différents octets à l'intérieur des images lorsque les topologies de l'image source et de l'image destination (dont on ne peut remplir qu'un tronçon à la fois, du fait qu'une image se transmet en plusieurs ticks d'horloge dans VideoDSP) diffèrent. L'implémentation de l'objet `vin#` donnée en annexe p. 49 en fournit un exemple.

2.5.2 La programmation multi-thread

Plusieurs objets du package *vio* utilisent des *threads* d'exécution concurrents afin d'améliorer les performances. L'utilisation de threads dans un programme est toujours un exercice délicat. Dans ce paragraphe, je vais présenter l'exemple de l'objet `vin#`, l'objet qui permet d'utiliser le périphérique d'acquisition video, dont le code source sous Linux est donné en annexe p. 49.

Lectures asynchrones. Dans la plupart des cas, le contrôle et le pilotage du périphérique d'acquisition video passe par l'appel système `ioctl()` (cf. [6]), auquel on donne :

- un descripteur de fichier (dans les systèmes UNIX, les périphériques sont symboliquement représentés par des fichiers) pour identifier le périphérique,
- un ordre sous forme de constante entière. Pour l'acquisition video, les noms de ces constantes sont pour la plupart de la forme "`VIDIOC[S|G]*`", `S` indiquant un changement (*set*) de paramètre, et `G` indiquant une lecture (*get*).
- un pointeur vers une zone mémoire permettant d'échanger des données avec le périphérique (la topologie de cette zone dépend de l'ordre donné).

Une fois le périphérique ouvert, on l'initialise en indiquant une méthode de réception des données des images. Dans l'implémentation de `vin#`, j'ai choisi de demander au périphérique de remplir une zone de mémoire partagée.

Ensuite, la capture d'images peut commencer. Chaque image se capture en trois temps :

1. Un ordre est donné au périphérique pour lancer la capture de l'image : c'est l'ordre `VIDIOCMCAPTURE` (le "`M`" remplace ici le "`S`" ou le "`G`" habituels, et signifie "*memory map*"). La capture est alors démarrée de manière asynchrone, c'est-à-dire que l'appel à `ioctl()` se termine immédiatement, et le programme continue son activité pendant que le

périphérique remplit la zone mémoire prédéterminée (ce qui, avec une webcam branchée sur le port USB, prend un certain temps puisqu'elle ne peut envoyer qu'une dizaine d'images à la seconde).

2. Une fois que l'application est prête à recevoir l'image, elle doit se synchroniser avec le périphérique, c'est-à-dire qu'elle doit attendre que toute l'image a été capturée. Ceci se fait grâce à l'ordre `VIDIOCSYNC`, qui a pour effet de rendre l'appel à `ioctl()` bloquant jusqu'à ce que toute l'image ait été capturée (bien sûr, si la capture de l'image est déjà terminée, l'appel à `ioctl()` se finit immédiatement).
3. L'image est maintenant entièrement contenue dans le buffer mémoire, on peut donc la traiter (dans le cas de `vin#`, on effectue une converion d'espace de couleur en même temps qu'une copie vers une autre zone mémoire).

Synchronisation avec VideoDSP. Le principal souci pour un système temps-réel est la performance. Au-delà du fait qu'une image est servie à VideoDSP en plusieurs morceaux, il est impensable de lancer la capture, l'attente et le traitement d'une image au moment où VideoDSP demande à l'objet `vin#` de lui servir une nouvelle image. Il faut donc que lorsque VideoDSP demande le premier morceau d'une nouvelle image, celle-ci soit déjà présente en mémoire, prête à être servie. Ceci ne peut être réalisé que grâce à un *thread* s'exécutant en parallèle du reste. Ainsi, pendant qu'une image est servie, la suivante est déjà en train d'être capturée (la figure 13 p. 29 montre de manière schématisée le déroulement de l'exécution des deux *threads*). Bien sûr, la nature asynchrone de la communication avec le périphérique rend la seule capture d'images possible sans threads. Mais le réel intérêt du thread vient du fait que le traitement de l'image qui suit la capture (conversion d'espace de couleur et copie vers une autre zone mémoire) s'effectue également en parallèle, ce qui constitue un gain de temps.

L'utilisation de threads sous Linux passe nécessairement par une bibliothèque externe. Le paragraphe suivant évoque l'utilisation de ces bibliothèques.

2.5.3 L'utilisation de bibliothèques externes

Nous avons déjà évoqué jusqu'ici plusieurs bibliothèques de fonctions externes sur lesquelles s'appuient les objets de `vin#`. Ces bibliothèques fournissent de nombreuses fonctionnalités qu'il n'est plus nécessaire de reprogrammer soi-même.

Mais la facilité d'utiliation de ces bibliothèques externes est souvent conditionnée par la documentation qu'elles proposent. Ainsi, les documentations de certaines bibliothèques comme *quicktime4linux* (utilisée pour développer un lecteur de fichiers video) et celle de *XVideo* sont quasi inexistantes,

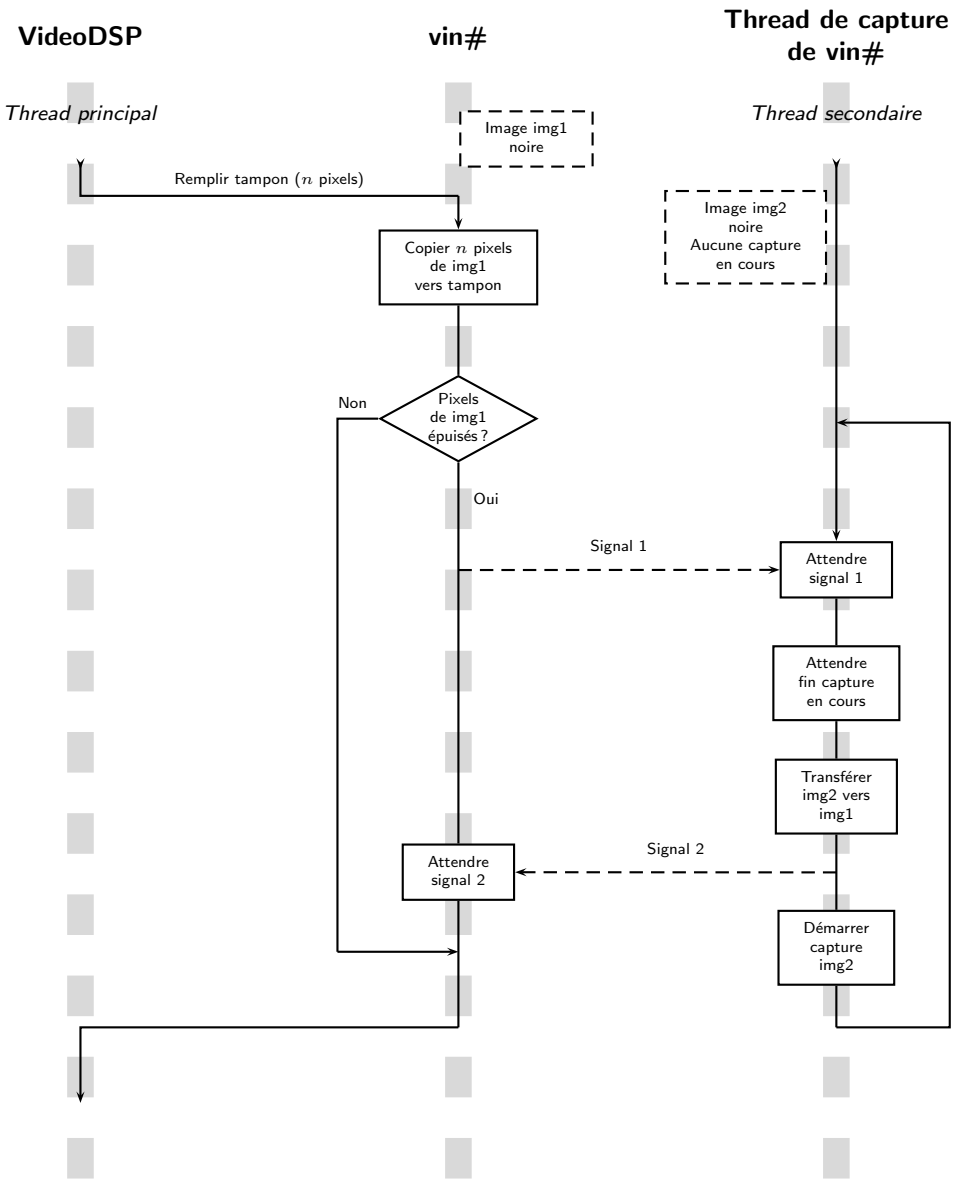


FIG. 13 – Diagramme temporel des événements lors d'une capture d'image

et le seul moyen de comprendre comment utiliser ces bibliothèques a été de regarder du code existant qui les utilisait. Dans ces conditions, développer des objets robustes et fonctionnels peut prendre un temps considérable.

La bibliothèque **SDL** (*Simple DirectMedia Layer*) est une bibliothèque offrant la plupart des services nécessaires à la programmation de jeux video. Elle offre l'avantage d'être disponible sur les mêmes plate-formes que jMax, à savoir Linux, Windows et MacOS X. Le module *vio* utilise SDL pour la programmation multi-threads et pour l'implémentation Windows de `vmonitor#`.

ImageMagick, nous l'avons déjà évoqué, est une bibliothèque de chargement et de traitement d'images numériques. Elle est utilisée par l'objet `vimage#` pour charger un grand nombre de formats d'images et par l'objet `vcapture#` pour enregistrer des images. Cette bibliothèque existe également sur les mêmes systèmes que jMax.

Les bibliothèques **XSHM** et **XVideo**, qui sont des extensions du système graphique d'UNIX X-Windows, sont utilisées par l'objet `vmonitor#` pour afficher le plus efficacement possible les images traitées sous jMax. Ces deux bibliothèques manquent totalement de documentation, et il a fallu regarder leurs fichiers d'en-tête C et du code existant pour comprendre leur fonctionnement.

Le cas de **quicktime4linux**, une bibliothèque d'encodage/décodage de fichiers QuickTime pour la video professionnelle sous Linux, s'est avéré plus douloureux encore, le code source ne compilant même pas sans plusieurs corrections du code et de la configuration du projet.

2.5.4 Le débogage d'un module jMax

Les modules externes de jMax sont écrits en C, et on peut dès lors utiliser le débogueur GDB (avec son interface textuelle) pour traquer les erreurs et les violations de segmentation.

Mais là où déboguer un programme ordinaire est relativement immédiat, avec jMax les choses se compliquent un peu, car lorsqu'on lance le logiciel, on lance en réalité l'interface graphique, écrite en Java, qui elle-même instancie FTS, le programme qu'on cherche à déboguer.

Par ailleurs, ce qu'on cherche à déboguer réellement, ce sont les modules qui eux sont chargés dynamiquement après le démarrage du programme. Certes, il est possible de lancer FTS directement en précisant le patch à charger, mais certains problèmes auxquels j'ai été confronté ne semblaient apparaître qu'en manipulant les objets, ce qui impliquait d'avoir l'interface graphique présente.

J'ai donc mis au point une procédure spéciale de lancement de jMax pour déboguer le module *vio* et VideoDSP :

1. Tuer les processus FTS qui seraient restés actifs à cause d'une erreur de manipulation de périphérique et de programmation concurrente. La

commande UNIX est : `killall -9 fts`.

2. Recompiler VideoDSP pour prendre en compte les derniers changements apportés au code. La commande pour effectuer cela est le très fameux `make`.
3. Installer la bibliothèque de mon module et les fichiers attachés au bon endroit : `make install`.
4. Lancer jMax : `jmax`.
5. Attendre trois secondes que FTS ait démarré et que tous les modules aient été chargés : `sleep 3`.
6. Obtenir le *Process ID* de FTS pour pouvoir y rattacher le débogueur : `ps -C fts`.
7. Lancer le débogueur pour le rattacher au processus de FTS : `gdb`.

Au final, la commande UNIX pour effectuer tout cela devient :

```
$ killall -9 fts ; make && make install && jmax && sleep 3
&& ps -C fts && gdb
```

Pour certains bogues liés à la mémoire, j'ai eu également recours à un outil de vérification d'utilisation de la mémoire, *valgrind*. Il s'agit d'un outil qui vérifie chaque lecture et écriture en mémoire et qui détecte les accès à des données non-initialisées ou à des zones non allouées.

2.6 Autres tâches réalisées

Au cours du stage, bien que mon principal travail a été de porter VideoDSP, j'ai eu l'occasion de réaliser d'autres tâches à la demande de mon maître de stage. Entre autres, j'ai contribué au démarrage d'une véritable documentation pour jMax, documentation qui manquait cruellement, et j'ai pu corriger certains bogues dans l'interface graphique du logiciel.

2.6.1 Correction de l'interface graphique

L'interface graphique de jMax souffrait d'un problème quant à sa fonctionnalité de "copier-coller", et il semblait que celle-ci n'était accessible qu'après quelques manipulations inutiles. Puisque je connaissais le langage Java, on m'a demandé de corriger ce problème.

Après m'être plongé dans le code de l'interface graphique, qui ne compte pas moins de 650 classes différentes, j'ai constaté que le problème était moins trivial que ce dont il avait l'air : ce qui ne semblait être rien d'autre qu'un simple problème de raccourci clavier s'est avéré être un véritable problème d'utilisation de modèles objets et des composants Swing[†].

J'ai donc eu l'occasion de corriger ainsi quelques classes du code Java en remaniant les associations entre objets et en clarifiant l'utilisation d'un

modèle de conception (ou *design pattern* en anglais) objet : le modèle d'*action*, très utilisé dans les interfaces créées avec Swing.

2.6.2 Outils de documentation

L'une des plus grande carences de jMax est la documentation, qu'elle soit destinée aux utilisateurs ou au développeurs. François Déchelle a exprimé le souhait de doter jMax d'une documentation complète, disponible à la fois au format HTML pour être lue dans un navigateur Web et au format PDF pour pouvoir être imprimée avec une qualité satisfaisante. J'ai été chargé de me renseigner sur DocBook, une norme d'écriture de documentation électronique qui permet de générer entre autres les deux formats cités ci-avant.

DocBook est en réalité une DTD pour SGML ou XML, c'est-à-dire que c'est un ensemble de règles d'écriture, qui précise comment structurer un document et comment préciser sa mise en forme. Il existe plusieurs outils permettant de créer des documents avec DocBook, et mon rôle a été de trouver ses outils et de mettre en place une base de travail pour commencer la documentation. Pour illustrer les capacités du système, j'ai démarré deux documents en anglais : *jMax — User's Manual* et *jMax — Developer's Manual*. Ces documents seront plus tard intégrés au projet jMax et deviendront les deux documents de référence pour l'utilisation de jMax et le développement de modules externes.

2.6.3 Ecriture d'articles

François Déchelle, chef de l'équipe Logiciels Libres et Ingénierie Logicielle au sein de laquelle s'est déroulé le stage, a mis en place un site Internet dédié aux logiciels libres et l'Ircam.[5] Le site contient principalement des articles, et une partie du site est consacrée à jMax.

Le site est bilingue, et contient des articles écrits en français et en anglais. François Déchelle m'a, dans un premier temps, demandé de traduire en anglais un article intitulé *Création de package externe jMax*, écrit par Patrice Tisserand. Puis, j'ai pu écrire un second article, en anglais également, intitulé *Porting packages to jMax 4.0*, petit guide de portage de modules externes pour la version 4.0 de jMax, et qui faisait suite à la création du programme `autoport` dont c'était justement le travail. Les deux articles sont disponibles en ligne ([4] et [5]).

[5]Le site se trouve à l'adresse suivante : `http ://freesoftware.ircam.fr`

3 Etude de l'architecture de FTS et réflexions

Le présent paragraphe est une étude de l'architecture de FTS, et plus particulièrement de la partie ayant trait au séquençage temps-réel des traitements effectués sur les flux de données. Il présente un compte-rendu des observations que j'ai faites quant à l'adaptation de FTS en vue de traiter des flux de données de nature différentes, tels que les flux video, et expose quelques points importants à prendre en compte pour rendre FTS capable de traiter des flux génériques.

Nous allons dans ce paragraphe explorer plus en profondeur le fonctionnement de FTS. Ceci va nous permettre ensuite d'en faire sortir les limitations, afin de pouvoir proposer des perspectives d'évolution.

3.1 Retour sur le fonctionnement de FTS

Voyons ici quelles sont les grands mécanismes de fonctionnement de la chaîne de traitement du signal audio dans FTS.

L'ordonnanceur. Au sein de FTS, les objets DSP (qu'on différencie des objets de contrôle, qui ne traitent pas de signal mais se contentent de réagir aux messages ou d'en envoyer à leur tour) sont placés dans une liste, appelée *chaîne DSP*.

A chaque sortie DSP[6] d'un objet est associée une zone de mémoire tampon, ou *buffer*, par laquelle va transiter le signal DSP en direction des entrées des objets connectés à cette sortie. Ce tampon est associé à un compteur de référence : à chaque fois qu'un objet est connecté à la sortie correspondante, le compteur est incrémenté. Il est important de noter qu'au début de la procédure d'ordonnancement, ces tampons n'ont pas encore de réelle zone mémoire allouée ; ce ne sont, au départ, que des objets abstraits. De plus, ces tampons sont placés dans un *pool*, de manière à ce que FTS en ait une connaissance au niveau global.

Lorsque l'utilisateur demande à activer le traitement du signal dans jMax, la chaîne DSP est ordonnancée, c'est-à-dire que FTS détermine un ordre d'exécution pour les objets DSP. Cet ordre est déterminé en effectuant un tri topologique sur le *graphe* dont les sommets sont les objets DSP et dont les arêtes sont les connexions entre les objets. Bien sûr, l'utilisation d'un tri topologique interdit la présence de circuits dans le graphe, et ceux-ci sont signalés à l'utilisateur lorsqu'ils sont repérés.

Chaque objet désigné par le tri topologique comme étant "le suivant" dans la liste des objets à s'exécuter au cours d'un *tick* d'horloge (typiquement 1.5 ms) doit lui-même allouer la mémoire pour ses tampons de sortie. L'opération s'effectue comme suit :

[6] Comme les objets eux-mêmes, les sorties des objets peuvent être des sorties DSP ou des sorties de contrôle. Les entrées fonctionnent sur le même principe.

1. L'objet commence par décrémenter les compteurs de référence de tous les tampons auxquels sont connectées ses entrées. Les tampons dont le compteur de référence atteint 0 sont rendus au *pool* de tampons.
2. L'objet alloue ensuite ses tampons de sortie, qu'il demande au *pool* de tampons.

Cette manière de procéder permet de diminuer le nombre de tampons utilisés en les "partageant" entre objets (Cf. paragraphe 3.2).

Le compilateur. Une fois déterminé l'ordre d'exécution des objets, FTS crée en mémoire une représentation linéaire du programme qui sera ensuite exécuté par FTL (Cf. plus bas).

Il est important de comprendre ce que signifie l'expression "exécuter un objet". En réalité, ce n'est bien sûr pas un objet qu'on exécute, mais l'une des ses méthodes. Les objets FTS sont, en effet, munis de méthodes, comme dans n'importe quel autre langage objet. Les objets DSP ont la particularité, par rapport aux objets de contrôle, de déclarer une (ou plusieurs[7]) méthode particulière qu'on appelle "méthodes DSP". Cette méthode se différencie des autres par la déclaration de ses paramètres. Là où les méthodes classiques des objets FTS partagent toutes une même signature[8], les méthodes DSP ont une déclaration bien particulière qui précise le nombre et le type de leurs paramètres (ces paramètres peuvent être les adresses des tampons d'entrée et de sortie, la taille de ces tampons, un pointeur vers l'objet lui-même, etc.). Lorsqu'on exécute un objet DSP, on invoque en réalité sa méthode DSP. Tous les paramètres d'une méthode DSP sont passés sous forme de valeurs 32 bits, qu'il s'agisse d'entiers, de flottants ou de pointeurs.

La représentation que prend le programme en mémoire après la compilation s'appelle *bytecode* : il s'agit d'une suite d'octets ayant, pour la machine virtuelle qui va exécuter ce bytecode, une certaine signification. Le compilateur crée donc ce *bytecode*, dont la figure 14 (p. 35) montre une schématisation simplifiée.

La machine virtuelle FTL. Une fois le programme compilé en *bytecode*, il est exécuté par FTL, la machine virtuelle de FTS chargée "d'exécuter" le programme de l'utilisateur en appelant les objets DSP dans l'ordre déterminé par l'ordonnanceur.

En réalité, FTL n'est qu'un petit automate déterministe à états finis (Cf. [3]). Il parcourt le *bytecode* et, en fonction de la valeur de la valeur de l'entier lu sous le "curseur", il passe dans l'un des états suivants :

[7] Bien qu'il soit possible de donner à un objet plusieurs méthodes DSP, c'est très rare de le faire en pratique.

[8] Cf. implémentation de l'objet `vin#`, p. 49

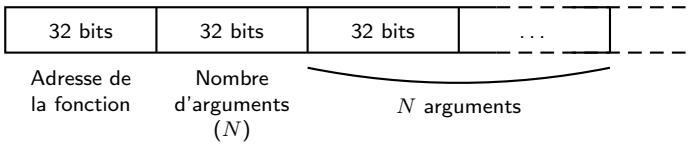
1. l'état `OPCODE` représente trois instructions possibles : appel de méthode, fin d'exécution du programme ou encore "aucune opération" (cette dernière instruction n'a aucun effet).
2. l'état `CALL_FUN` indique que les octets qui suivent vont décrire les paramètres d'un appel de méthode.
3. l'état `CALL_ARGC` indique que sous la tête de lecture se trouve un entier indiquant le nombre de paramètres pour l'appel de méthode.
4. l'état `CALL_ARGV` indique que sous la tête de lecture se trouve l'un des paramètres. L'automate reste dans cette état autant de fois qu'il y a de paramètres.

La figure 15 p. 36 donne le graphe de l'automate.

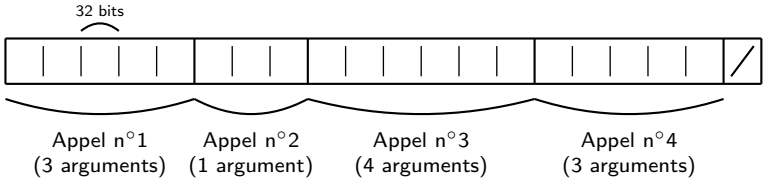
3.2 Les optimisations dans FTS

FTS est une application temps-réel, et de ce fait, elle se doit de minimiser le plus possible le temps de latence lors du traitement du signal audio. Pour atteindre ce but, un certain nombre de techniques d'optimisation ont été utilisées pour rendre l'exécution de FTS plus rapide. Nous allons exposer dans ce paragraphe deux de ces techniques, particulièrement liées à l'architecture de la machine [2].

Le pipeline de l'unité de calculs flottants. Le processeur d'un ordinateur exécute des opérations, qu'on considère souvent comme "atomiques", c'est-à-dire qu'on les voit comme les opérations de base que peut effectuer



(a) Topologie d'un appel de fonction FTL



(b) Un programme FTL est principalement une suite d'appels

FIG. 14 – Topologie du *bytecode* de FTL en mémoire.

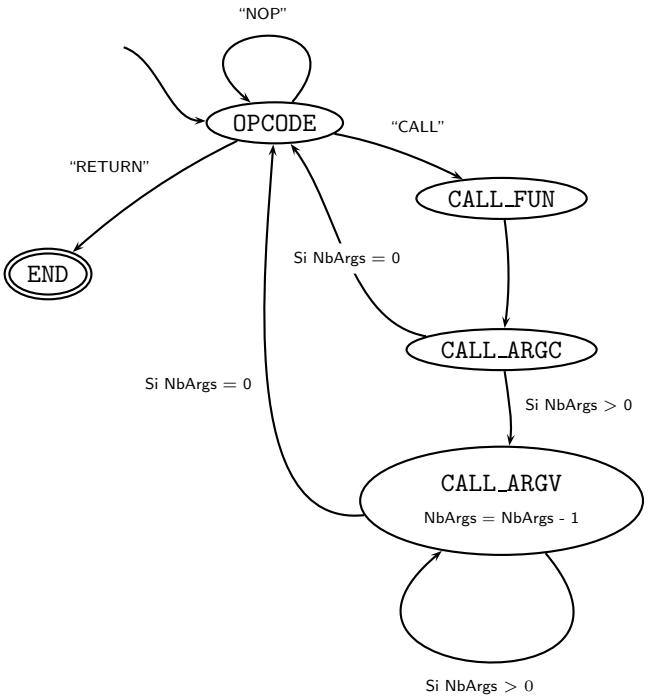


FIG. 15 – Graphe de l'automate de FTL

le processeur, et qu'on assemble judicieusement pour créer de nouvelles instructions plus puissantes (c'est d'ailleurs exactement sur ce même principe que repose jMax, avec ses "petits" objets qu'on assemble dans des patches pour créer des traitements complexes).

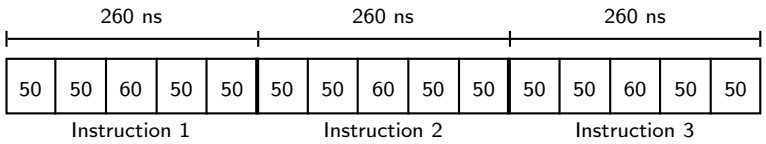
Cependant, ces instructions, si on se place au niveau *interne* du processeur, ne sont pas tout à fait atomiques : l'exécution de chaque instruction se décompose en plusieurs étapes : lecture de l'instruction, décodage de celle-ci et lecture des registres, exécution et calcul, accès mémoire, écriture du résultat, etc. Ces étapes dépendent bien entendu de l'architecture du processeur.

La première approche, évidente et presque naïve, est de faire passer les instructions dans cette "chaîne d'exécution" les unes après les autres, la suivante n'entrant dans la chaîne que lorsque la courante a terminé. Pour exécuter n instructions, il faut donc n fois le temps cumulé pour passer dans chacune des étapes de la chaîne.

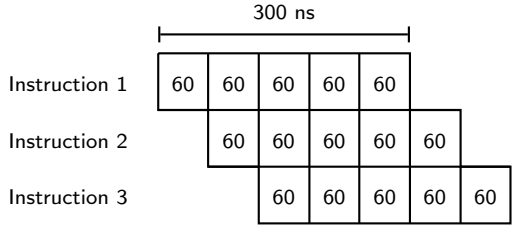
La technique du *pipeline* permet de se faire chevaucher les instructions au cours de leur exécution. On fait entrer une instruction dans la première section de la chaîne, puis, dès qu'elle passe dans la seconde section, on fait entrer l'instruction suivante dans la première section qui vient de se libérer. Ainsi, une fois que la première instruction sort de la chaîne, l'instruction suivante ne sortira qu'après le petit temps qu'il faut pour compléter la *dernière* étape de la chaîne, ce qui est bien sûr beaucoup plus rapide que lorsqu'il faut attendre que l'instruction suivante traverse toute la chaîne. La figure 16 illustre cette différence.

Nous n'exposerons pas ici les quelques problèmes liés à cette technique, comme la dépendance d'une instruction vis-à-vis du résultat de l'instruction précédente, ou encore les branchements (sauts) dans les programmes. Ces problèmes "brisent" la chaîne et peuvent faire perdre ses avantages à la technique du pipeline, mais ce sont des problèmes mineurs, qui n'apparaîssent pas fréquemment dans les parties critiques de FTS.

Comment FTS tire-t-il partie de cette technique ? Le plus gros travail que doit effectuer l'application est un travail de calcul sur les échantillons audio, c'est-à-dire sur des valeurs flottantes. Chaque objet de la chaîne DSP est appelé à son tour pour effectuer son traitement sur les données. Si on ne demandait à chaque objet que de traiter un seul échantillon à la fois, les instructions seraient très hétérogènes : instructions de saut, tests et calculs de valeurs flottantes, ainsi mélangées, feraient justement ressortir les problèmes évoqués plus haut. Ce que fait FTS, c'est qu'il demande à chaque objet d'effectuer ses calculs sur un *vecteur* d'échantillons (64 échantillons à la fois). Ceci a l'avantage que, une fois quelques tests et branchement effectués, on ne fait plus que des calculs sur des nombres à virgule flottante pour un petit moment. Or, l'unité de calculs flottants présente sur la plupart des ordinateurs modernes utilise justement un *pipeline*, ce qui signifie que durant les calculs sur un vecteur, les opérations sont homogènes, et il



(a) Lors d'une exécution non-*pipelinée*, le temps total d'exécution des trois instructions est de 3×260 ns $= 780$ ns



(b) Lors d'une exécution *pipelinée*, les instructions se chevauchent, et le temps total est de $7 \times 60 = 420$ ns

FIG. 16 – Exécution de trois instructions sans puis avec pipeline.

est rare que le calcul sur un échantillon dépende du résultat du calcul sur l'échantillon précédent. Dans ce cas, la technique de *pipeline* fait merveille, et on a pu constater un gain proche de 1000% en efficacité de calcul lorsque le *pipeline* est ainsi exploité.

Le compilateur est également capable d'optimiser le code produit pour tirer parti de cette architecture, notammenent en "déroulant" les boucles. Plutôt que d'exécuter n fois le corps d'une boucle, il peut arranger le code produit pour exécuter $\frac{n}{4}$ fois le corps de la boucle répété quatre fois. Ainsi, on diminue le nombre de sauts et de tests à effectuer, et le pipeline est mieux exploité.

Mémoire cache et localité des accès mémoire. Une seconde technique d'optimisation, liée cette fois-ci au fonctionnement de la mémoire à été employée dans FTS.

A l'époque où les cadences des processeurs se comptaient encore en dizaines de méga-hertz, la mémoire vive était l'un des composants les plus rapides de l'ordinateur, et les accès mémoire effectués par les processeurs de l'époque ne posaient pas vraiment de problème de performance. Mais aujourd'hui, les choses ont changé, la mémoire est devenue trop lente par rapport aux puces giga-hertziennes devenues courantes, et chaque accès mémoire impose une forte pénalité en termes de performances.

Pour pallier à ce problème, on a introduit d'autres types de mémoires, plus rapides (mais aussi de moindre capacités, car plus chères à fabriquer), les mémoires caches. Aujourd'hui, un PC "standard" dispose d'une mémoire vive, puis de deux ou trois couches de mémoire chache (chaque couche ayant une capacité plus faible mais étant plus rapide que les précédentes).

Lorsque le processeur demande à lire une donnée en mémoire, celle-ci est transmise à la mémoire cache la plus proche du processeur, mais elle n'est pas transmise seule : elle est accompagnée de ses voisines, dans l'espoir que les prochains accès seront justement des accès à ces données voisines. Dans ce cas, étant donné qu'elle sont déjà dans le cache, l'accès est très rapide. On peut ainsi stocker un petit nombre d'"ensembles" de données dans le cache. Par contre, si le processeur requiert l'accès à une donnée qui ne se trouve pas dans le cache, il faut libérer une place dans celui-ci, puis rapatrier la donnée demandée. Ce phénomène, chronophage, s'appelle en anglais *cache miss*.

Pour limiter l'apparition de ce phénomène, FTS tente d'utiliser le moins de mémoire possible pour exécuter la chaîne DSP. Si tout le programme et une bonne partie des tampons d'échantillons sont dans le cache, l'exécution est très rapide car on limite le nombre d'accès à des données qui ne s'y trouvent pas. De ce fait, la taille des vecteurs, 64 échantillons, ce qui fait des tampons de 128 octets (16 bits par échantillon), résulte d'un compromis entre le gain donné par l'utilisation du pipeline (qui est plus avantageuse pour de grandes tailles de vecteurs) et la taille de la mémoire cache (moins on a de tampons dans le cache, plus les accès à la mémoire centrale lente sont nombreux).

C'est pour cela aussi qu'un programme DSP a une représentation mémoire très compacte, et qu'on a choisi d'utiliser un nombre variable d'arguments pour les méthodes DSP, afin de ne pas utiliser plus de mémoire que nécessaire.

3.3 Caractéristiques de flux de données video

La problématique soulevée par mon stage tourne autour de l'intégration de traitements de flux *génériques* dans FTS. En effet, aujourd'hui FTS n'est capable que de traiter des flux d'échantillons audio de 16 bits, codés sous forme de nombres à virgule flottante. Quelles sont les modifications à apporter à FTS en vue d'un traitement complètement générique des données ?

Micro-échantillons et macro-échantillons. La représentation du son sous forme d'échantillons est assez simple : les échantillons sont très petits (de 16 à 32 bits dans la plupart des applications), et ils sont traités par blocs pour plus d'éfficacité.

Le cas d'un flux video est légèrement différent : faut-il considérer que les échantillons sont les *pixels* des images, ou doit-on au contraire considérer les

images elles-mêmes comme des échantillons ? Il semble raisonnable de choisir la première alternative : le fait d'avoir des échantillons petits (là encore, de 8 à 32 bits dans la plupart des cas, voire même 1 bit pour les images monochromes) rend leur déplacement plus facile et plus efficace. Mais alors apparaît un problème peu courant avec le son : certains traitements video nécessitent d'avoir l'image entière. C'est le cas notamment lorsqu'on veut appliquer un filtre grâce à une matrice de convolution. Or, le cas de VideoDSP le montre bien, pour garder des tailles de tampon raisonnables, l'image doit être "découpée" en plusieurs tronçons envoyés successivement les uns après les autres entre les objets. Or, appliquer une matrice de convolution sur une image dont on n'a qu'un morceau à la fois est une opération délicate.

On peut introduire la distinction entre micro-échantillons (les pixels) et macro-échantillons (les images). Certains traitements se font directement sur le micro-échantillons (dans VideoDSP, il suffit de modifier les valeurs du flux Y pour modifier la luminosité de l'image, mais on n'en n'a pas besoin dans son intégralité), alors que d'autres doivent être appliqués aux macro-échantillons (appliquer un flou à une image, par exemple, nécessite d'avoir l'image entière). Comment faire alors pour contenter ces deux sortes d'objets dans une même chaîne de traitements ? Agrandir la taille des tampons pour y faire tenir les images entières provoquerait à coup sûr une grande baisse de performances pour les raisons de localité de mémoire, détaillées ci-avant dans le paragraphe sur la mémoire cache.[9] On pourrait envisager de garder des tampons les plus petits possibles et de n'utiliser de grands tampons que pour les objets qui ne peuvent traiter que des images entières. FTS se chargerait alors de "rassembler" les bouts de l'image dans un tampon avant de les servir à l'objet, puis de les "désassembler" après le traitement, mais cela reviendrait à attendre plusieurs cycles d'horloge dès qu'un seul objet existerait qui aurait besoin de travailler sur l'image entière. On peut également envisager une autre approche, mais qui serait légèrement moins générale : si un objet ne fait que filtrer l'image avec une matrice de convolution, on pourrait modifier FTS pour que celui-ci soit capable d'appliquer directement une matrice de convolution à des parties de l'image, en faisait automatiquement les calculs nécessaires pour prendre en compte les "recouvrements" au bords des tronçons. Cette solution pourrait se révéler un moyen efficace pour appliquer des matrices de convolution (quelque soit la dimension du signal), mais ne permettrait toujours pas de gérer efficacement tous les types de traitements particuliers sur l'ensemble de l'image. De plus, il serait tout aussi envisageable de déléguer ce travail à un objet particulier, sans réel besoin de modifier FTS.

[9] En effet, la taille en octets de la mémoire cache des processeurs actuels est du même ordre de grandeur que le nombre d'octets nécessaires pour coder numériquement une image de 320×240 pixels avec 16 bits par pixel, ce qui impliquerait que même un très petit nombre de tampons pouvant contenir une telle image ne peuvent tenir tous en même temps dans le cache, provoquant ainsi de nombreux *cache misses* à chaque *tick* d'horloge.

Le traitement de la video semble inexorablement impliquer une baisse substantielle de performances à cause de la taille des données à traiter. Cependant, il semble nécessaire d'introduire cette notion d'*échelles* d'échantillons. Le développement d'objets video a montré plusieurs fois la nécessité d'avoir une sorte de "signal" de synchronisation pour délimiter le début d'un nouveau macro-échantillon (ce travail était fait grâce à une variable membre des objets en question qui contenait le nombre de pixels déjà traités et qu'on remettait à zero à chaque fois qu'il devenait au moins aussi grand que le nombre de pixels contenus dans une image).

Nature et codage des échantillons. Le travail sur VideoDSP a fait resortir une autre difficulté. Les objets du module *vio* sont tous implémentés pour traiter des images YUV422. Ce choix résulte d'un débat qui a eu lieu entre les différents développeurs intéressés par le traitement de la video sous jMax. Certains voulaient traiter la video en RGB, d'autres en YUV, chacun ayant ses arguments. Mais il semblerait que le besoin existe de pouvoir traiter les images avec différents codages pour les pixels. Ceci nous amène à nous demander comment FTS pourrait faciliter la coexistance de plusieurs types de codages au sein d'un même patch pour des données de même nature (video, son, etc.)

Comment caractériser un type de codage ? Pour les flux video, il existe les codes FOURCC (*Four Character Code*, cf. [8]), qui indiquent tout ce qu'il faut savoir sur le codage d'une image video. De cet identificateur, on tire les caractéristiques de l'image, pour peu qu'on connaisse le code FOURCC. Il existe une classification similaire pour l'audio (au format PCM), mais pas pour d'autres types de flux multimédia hypothétiques. Cette classification est centralisée (Les nouveaux codes FOURCC sont enregistrés auprès de Microsoft).

Mais cette simple classification limite quelque peu l'intérêt d'un système capable de traiter n'importe quel type de flux de données. L'audio, par exemple, est codé grâce à des nombres flottants sur 16 bits, alors que la video se code plutôt pas pixels entiers de 8 bits. Or, si certains traitements ont besoin d'associer une *sémantique* aux données, d'autres peuvent tout simplement permettre d'appliquer des opérations à des valeurs. Un objet pourrait par exemple appliquer une multiplication par un scalaire à tous les micro-échantillons d'un tampon, ou encore calculer une moyenne sur ces valeurs. Bien entendu, ces opérations ne sont pas les mêmes pour des flottants et pour des entiers, et l'objet devrait fonctionner quelque soit le type de média, même si celui-ci n'est pas connu par le créateur de l'objet. Il s'agit dès lors de déterminer la quantité d'informations dont il est nécessaire de disposer afin de tirer le meilleur parti d'une vision générique des flux de données :

– Nature des échantillons (entier 8 bit non-signé, flottant 16 bits, etc.)

- Type de media (audio, video, etc.)
- Codage (PCM, YUV420, RGB32, etc.)
- Taux d'échantillonnage (audio à 44100 Hz, video à 25 images/seconde, etc.)
- Taille d'un micro-échantillon (une image : 160×120 pixels), etc.

Négociations de types. Dans la version actuelle de jMax (4.1), les entrées/sorties DSP d'un objet sont associées à un symbole (c'est-à-dire une chaîne de caractères) qui précise la nature du signal et qui permet à jMax de refuser à l'utilisateur d'effectuer une connexion qui n'aurait pas de sens (comme tenter de relier une sortie émettant de l'audio à une entrée qui prend un nombre entier). Tous les objets audio de jMax utilisent pour cela le même symbole. En pratique, cela rend possible l'élaboration d'extensions telles que VideoDSP qui associent un autre symbole aux entrées/sorties video de ses objets afin que l'utilisateur ne puisse pas connecter une sortie audio à une entrée video. Mais ce type d'identification de flux montre vite ses limites si on souhaite faire de jMax un outil capable de traiter n'importe quel type de données. Puisqu'il n'est pas possible de prévoir à l'avance quels types de données vont pouvoir être traités par un objet donné, il faut donner aux objets la possibilité de déclarer eux-mêmes leurs capacités. Ceci pourrait se faire au moment où une connexion doit être établie. L'objet qui reçoit la connexion est informé du type de signal qu'on souhaite lui faire parvenir et peut, en fonction des informations qu'il a sur ce signal, accepter ou refuser la connexion. Ainsi, une opération de multiplication par un nombre flottant pourrait accepter tout type de signal codé à partir de flottants, alors qu'un traitement video de détection de formes pourrait n'accepter qu'un flux video codé en RGB par exemple. Cette idée pourrait même être étendue pour que deux objets "négocient" le type de flux afin que celui-ci soit le mieux adapté aux deux objets (le simple rejet ou acceptation d'un signal pose problème quand la source est capable d'envoyer plusieurs types de signaux) : chaque objet donne une liste de types qu'il peut traiter, avec un ordre de préférence, et jMax choisit le type commun le mieux apprécié des deux objets.

Structure des tampons. Tant qu'il ne s'agit que d'audio, les tampons de jMax sont parfaitement adaptés. Comme décrit plus haut, il s'agit en réalité d'objets particuliers possédant un compteur de référence et une zone mémoire. Cependant, d'autres types de données peuvent nécessiter d'autres types de tampons, qui pourraient être de tailles variables, ou qui pourraient être associés à plusieurs zones mémoires, ou même à des espaces sur le disque si les zones mémoires demandées sont trop grandes.

Importance des fréquences d'échantillonnage. Dans le cas d'un flux audio, il est indispensable de se conformer au taux d'échantillonnage déterminé

à l'avance (44100 Hz, par exemple), car un "trou" dans un flux audio est perçu de manière forte par l'oreille humaine. Une application qui traite du son doit donc absolument respecter la cadence à laquelle circulent les échantillons. Dans le cas de la video, un léger retard est moins grave : l'image est un peu moins fluide, mais ce n'est pas perçu avec la même gravité par l'œil. D'ailleurs, l'exemple de la webcam le montre bien : dans la configuration utilisée lors du développement, la webcam ne pouvait envoyer plus d'une dizaine d'images par seconde, mais si l'image était un peu saccadée, cela suffisait largement pour expérimenter des traitements. Une carte d'acquisition TV, par exemple, peut fournir 25 images par seconde. Il en resulte que le taux d'échantillonnage n'a pas la même importance pour tous les types de flux, et si dans le cas de l'audio, on précise à l'avance la fréquence d'échantillonnage à laquelle on désire travailler, on peut, dans le cas de la video, se contenter de ce qu'on reçoit.

Quelques exemples de types de flux. Un logiciel comme jMax, programmable, qui serait capable de traiter n'importe quel type de flux de données trouverait de nombreuses applications, et pas seulement dans le domaine du multimédia. FTS est suffisamment bien conçu pour pouvoir être embarqué, ce qui permettrait de créer de nombreux appareils de traitement de données :

- Un module de traitement de flux TCP pourrait permettre d'expérimenter et de créer des firewalls et des anti-virus, par exemple,
- en traitant les données génériques au niveau de l'octet, on peut créer des chaînes de traitements pour la compression ou le cryptage de données (activité qui nécessite réellement des tampons conçus de manière sécurisée),
- un module pourrait traiter des flux de données XML et en extraire des données pertinentes, etc.

4 L'expérience du stage

Nous avons jusqu'ici parlé du travail, mais très peu des conditions dans lesquelles il s'est déroulé. L'objet de cette partie est de préciser le déroulement du stage, d'exposer mes points de vue et l'expérience que j'en ai retiré et enfin de détailler l'environnement technique.

4.1 Déroulement du stage

Temps de formation. Dès le début du stage, j'ai été confronté à des outils et des logiciels que je ne connaissais pas — jMax en premier. Le premier jour, je l'ai passé à lire des articles sur jMax et sur l'architecture de FTS. Puis, peu à peu, j'ai regardé du code de modules externes jMax, afin de me familiariser avec l'API de programmation de FTS. Au bout d'une semaine, j'avais écrit un petit objet FTS capable de générer des nombres aléatoires suivant une loi normale.

Il m'a fallu environ trois semaines pour véritablement prendre en main tous les outils que j'allais utiliser pour travailler (CVS, Emacs, automake-/autoconf, etc. Cf paragraphe 4.2).

Analyse et évaluation des problèmes. Très tôt dans le stage, mon travail a été d'étudier le portage des modules video existants. J'ai donc étudié en détail le code des modules existants ainsi que les quelques pages de documentation qui existaient à ce sujet ; toutes ces connaissances ont bientôt été compilées dans le programme `autoport`.

La "tournée administrative". Lors de mon arrivée à l'Ircam, j'ai dû visiter un certain nombre de services pour obtenir toutes les pièces nécessaires à mon séjour dans l'institut, et notamment le badge d'accès et la clé de mon bureau. J'ai également dû passer par le service informatique afin de procéder à l'ouverture de mon compte UNIX sur le réseau de l'institut. Deux jours de démarches au total.

Intégration dans l'équipe. L'équipe des Logiciels Libres et Ingénierie Logicielle, composée de François Déchelle et de Patrice Tisserand, m'a très bien accueilli lors de mon arrivée. J'ai bénéficié d'un poste dans le bureau de l'équipe ce qui m'a permis d'avoir réponse à mes très nombreuses questions, et j'ai pu travailler en compagnie de deux (et parfois trois) autres stagiaires, dont l'un qui a repris mon travail pour le porter sous Windows peu avant la fin de mon stage.

4.2 Le développement d'un logiciel libre

Travailler sur un logiciel libre s'est révélé être une expérience très positive et enrichissante. Il s'agit d'une façon de travailler relativement différente des projets logiciels propriétaires, mais elle impose un ceratin nombre d'outils et d'habitudes qu'il faut apprendre à utiliser rapidement.

Synchronisation des sources. jMax étant un logiciel libre, son code source est disponible publiquement, et travaillent dessus des développeurs qui ne sont pas tous à l'Ircam. Pour rendre cette répartition du travail possible, jMax est hébergé en tant que projet sur SourceForge.net, un site Internet central pour la création de logiciels libres. Il s'agit d'un site fournissant de nombreux services pour des projets *Open Source*, tels que la gestion et le contrôle de version des sources, la mise en place de listes de diffusion électroniques, des outils de soumission de suggestion ou de bogues, etc.

Pour le développeur, l'outil central dans la gestion de ses sources est CVS, *Concurrent Versions System*, un système qui permet à plusieurs développeurs de travailler en même temps sur un même répertoire de code source et qui gère la synchronisation des fichiers modifiés pour que tout le monde puisse travailler avec des sources à jour.

L'utilisation de CVS impose au développeur de régulièrement mettre à jour son répertoire local, de soumettre ses changements régulièrement, en indiquant à chaque fois ce qu'il a modifié et d'une maniège générale de faire attention à ne pas "casser" le répertoire en oubliant d'inclure des fichiers nécessaires ou en déclarant certaines propriétés importantes des fichiers (le développeur doit déclarer manuellement si un certain fichier est un fichier binaire, afin que CVS ne le modifie pas automatiquement comme il le fait pour les fichiers source).

Compiler pour plusieurs plate-formes. Les projets de logiciels libres suivant les conventions GNU utilisent des méthodes qui permettent de créer des programmes capables d'être compilés et exécutés sur de nombreuses variantes d'UNIX, en prenant en compte les nombreuses petites différences qui existent d'une variante à l'autre.

Pour cela, les projets utilisent des outils (libres, bien sûr) qui permettent d'écrire de la manière la plus générique et portable qui soit. Ainsi, jMax utilise :

- `make` pour ne recompiler que les parties d'un projet qui ont changé,
- `autoconf` pour vérifier certaines caractéristiques du système sur lequel le projet est compilé (présence des bibliothèques externes, de fichiers d'en-tête C, de commandes particulières, de comportements de certaines fonctions lorsque celui-ci diffère selon les plate-formes, etc.)
- `automake` pour générer, à partir des sources du programme les *Makefiles*, c'est-à-dire les fichiers de commandes pour `make`, qui contiennent

automatiquement toutes les dépendances entre les différents fichiers sources (afin que si un fichier est modifié, tous les autres fichiers qui en dépendent soient recompilés à leur tour).

La plupart de ces outils prennent en entrée un ou plusieurs fichiers précisant le travail à effectuer, et cela signifie autant de syntaxes étranges et exotiques à apprendre et à maîtriser.

Echanges avec d'autres développeurs. Comme la plupart des logiciels libres, jMax a une liste de diffusion électronique pour que les développeurs puissent échanger des idées et coordonner leurs efforts.

Durant mon travail, j'ai eu à contacter plusieurs de ces développeurs, et en particulier ceux qui avaient créés les modules video (GridFlow et VideoDSP). Les échanges se faisaient systématiquement en anglais.

4.3 Implication dans jMax

L'une des autres conséquences du fait que jMax soit un logiciel libre est que chacun peut s'y impliquer comme il l'entend. Or, pour un musicien, il est très stimulant et motivant de travailler au développement d'un logiciel qu'on peut être amené à utiliser dans son propre travail artistique. J'ai donc très vite imaginé un grand nombre d'utilisations que je pourrais faire de ce logiciel, et j'ai commencé à le regarder d'un œil critique en me demandant ce que je pourrais faire pour l'améliorer.

J'ai donc utilisé à plusieurs reprises le *Bug Tracker* mis en place par SourceForge.net pour déclarer quelques bogues de jMax, et cela m'a naturellement conduit à en corriger moi-même (Cf. paragraphe 2.6.1).

J'ai également eu l'occasion d'apporter de l'aide aux utilisateurs au travers de la liste de diffusion.

4.4 L'émulation positive de l'environnement

Effectuer ce stage à l'Ircam a été pour moi une expérience enrichissante à plusieurs niveaux : techniquement, bien sûr, j'ai appris des choses, mais j'ai également pu découvrir les applications artistiques de ce travail. Régulièrement, les différentes équipes de recherche donnent des conférences pour exposer leurs travaux, ce qui a été pour moi l'occasion de découvrir de nombreuses technologies pour lesquelles les idées d'applications ne m'ont jamais manqué.

J'ai eu l'occasion de parler avec de nombreux chercheurs et musiciens, et j'ai même eu l'occasion de tester un système de suivi de partition mis au point à l'Ircam et utilisé que par quelques rares compositeurs aujourd'hui.

Aujourd'hui, je compte utiliser les technologies que j'ai découvertes à l'Ircam pour mon propre travail artistique, en particulier en situation de

concert. De ce point de vue, ce stage représente une expérience extraordinairement riche.

4.5 Environnement technique

Voici une brève description de l'environnement technique dans lequel j'ai travaillé :

- un PC muni d'un processeur Intel Pentium IV à 2.4Ghz et 512 Mo de mémoire
- un écran 22"
- une distribution Linux RedHat 7.3
- une connexion au réseau de l'Ircam, lui-même relié au réseau RENATER
- d'autres périphériques amusants (mais néanmoins indispensables à mon travail...) : webcam, carte son reliée à une table de mixage, etc.

L'environnement de développement était un environnement standard pour le développement OpenSource : Emacs, GCC, automake/autoconf, GDB, etc. Le projet jMax est hébergé par SourceForge.net, et nous avons utilisé CVS pour la gestion de versions de fichiers.

Conclusion

Dans son ensemble, ce stage a été extrêmement enrichissant, tant par le travail que par l'environnement. Sur le plan de la formation, il m'a permis d'explorer plus en profondeur les dimensions pratiques de certaines matières qui m'avaient beaucoup intéressé, en particulier le traitement du signal, les langages et automates et l'architecture des machines. Je n'ai pas éprouvé de manque flagrant en particulier, et les quelques lacunes que j'avais au début du stage ont vite été comblées. Il en ressort que, plus qu'une connaissance particulière, c'est la généralité et la diversité de l'enseignement de l'IUP qui m'ont permis d'appréhender avec succès — je pense — ce stage. Si je devais recommencer ce même stage, j'attacherais sans doute plus d'importance à l'environnement du logiciel, en rendant mon code plus portable encore et en faisant plus attention à mon utilisation des logiciels `autoconf` et `automake`. Ce stage n'a pas été moins enrichissant sur le plan personnel, et la perspective de continuer ma formation à l'Ircam et d'explorer plus avant les technologies liées à la musique m'enchante.

Annexes

Implémentation de l'objet `vin#`

```
/*
 * jMax
 * Copyright (C) 1994, 1995, 1998, 1999 by IRCAM-Centre Georges Pompidou, Paris, France.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 *
 * See file LICENSE for further informations on licensing terms.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
 */

/*
 * Implementation of the [vin#] object. This object reads a video stream from
 * a video input device.
 * Author : Carl Seleborg
 */


#include <fts/fts.h>
#include <vdsp/vdsp.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/time.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <linux/videodev.h>
#include <SDL/SDL.h>
#include <SDL/SDL_thread.h>
#include "vio.h"
#include <assert.h>


/*
 * Pointer to a palette handler (function that can handle a certain type
 * of video-input palette).
 * We use this kind of pointer because it is possible that in the future, other
 * palettes have to be supported. The conversion between those palettes has to be
 * handled differently, so we use these generic palette-conversion functions to do
 * the job. If, for instance, your webcam provides you with RGB (though it should be
 * some sort of YUV most of the time), you can simply write a handler for that RGB palette
 * and change the few parts of the code where this is hanlede (there is a "switch" statement
 * a little further down the code where palette handlers are assigned).
 */
```

```
struct _vin_t;
typedef void (*palette_handler_t)(struct _vin_t*);


/* States of a vin* object */
typedef enum
{
  cam_disabled,
  cam_enabled,
  cam_paused
} cam_state_t;


/**
 * Yuv image. Both y_data and uv_data are V_SIZE bytes long.
 */
typedef struct
{
  unsigned char* y_data;
  unsigned char* uv_data;
} yuv_image_t;


/*
 * vin* object structure.
 */
typedef struct _vin_t
{
  /* camin_t derives from fts_object_t */
  fts_object_t  super;

  /* name of the input devide */
  fts_symbol_t  device;

  /* Capabilities of the capture devide */
  struct video_capability caps;

  /* Picture properties */
  struct video_picture pic;

  /* Channel properties */
  struct video_channel* channels;

  /* Memory buffer */
  struct video_mbuf mbuf;

  /* Memory map information */
  struct video_mmap mmap;

  /* file descriptor of the device */
  int fd;

  /* Palette handler */
  palette_handler_t palette_handler;

  /* Start of memory map */
  unsigned char* data;

  /* Size of memory map */
  size_t data_size;

  /* Is enabled/disabled/paused ? */
```

```
  cam_state_t state;

  /* Current frame offset */
  size_t offset;

  /* Working thread */
  SDL_Thread* thread;

  /* Mutex used for synchronization */
  SDL_mutex* capture_mutex;
  SDL_mutex* server_mutex;

  /* Condition variable for thread synchronization */
  SDL_cond* capture_cond;
  SDL_cond* server_cond;

  /* Frame load scheduling */
  int frame_beeing_loaded;
  int frame_to_load;

  /* Two offscreen images that we will swap to increase efficiency */
  yuv_image_t frames[2];

  /* Total bytes per pixel in the input format (gives us the size of a frame in bytes) */
  size_t bytes_per_pixel;

  /* This boolean variable protects us from getting stuck in an infinite loop
   * if the user has wired the output params to some input slider.
   */
  unsigned char internal_change;

  /* Tells the capture thread that the imge parameters have changed. */
  int parameters_have_changed;
} vin_t;


fts_symbol_t vin_output_symbol = 0;



/******************************************************************************/

/* Function declarations */
static int vin_init_input_device(vin_t* self);
static int vin_open_input_device(vin_t* self);
static int vin_close_input_device(vin_t* self);
static void vin_start_capturing_next_frame(vin_t* self);
static void vin_enable(fts_object_t* o, int winlet, fts_symbol_t s,
           int ac, const fts_atom_t* at);
static void vin_disable(fts_object_t* o, int winlet, fts_symbol_t s,
      int ac, const fts_atom_t* at);
static void vin_output_image_params(fts_object_t* o, int winlet, fts_symbol_t s,
            int ac, const fts_atom_t* at);


/* Palette handlers */
static void vin_handle_YUV420P(vin_t* self);

/* Thread routine */
static int vin_thread_routine(void* arg);

/* Symbols */
```

```
/* Opens the video device and starts sending images */
static fts_symbol_t VIN_S_ENABLE = 0;

/* Stops sending images and stops the video device */
static fts_symbol_t VIN_S_DISABLE = 0;

/* Stops capturing images, and keeps sending the last image captured */
static fts_symbol_t VIN_S_PAUSE = 0;


/* Inlets and their corresponding parameter */
#define VIN_INLET_BRIGHTNESS     0
#define VIN_INLET_HUE            1
#define VIN_INLET_COLOUR         2
#define VIN_INLET_CONTRAST       3
#define VIN_INLET_WHITENESS      4




/************************************************************************/
/* Object implementation.                                               */
/************************************************************************/

/**
 * Object constructor.
 *
 * @param o A pointer to this object.
 * @param winlet <i>unused</i>
 * @param s <i>unused</i>
 * @param ac Argument count
 * @param at Argument array
 **/
static void vin_init(fts_object_t* o, int winlet, fts_symbol_t s, int ac, const fts_atom_t* at)
{
  vin_t* self = (vin_t*)o;
  int i;

  self->fd = -1;
  self->device = 0;
  self->channels = 0;
  self->state = cam_disabled;
  self->offset = 0;
  self->frame_beeing_loaded = FALSE;
  self->frame_to_load = 0;
  self->frames[0].y_data = (unsigned char*)fts_malloc(V_SIZE);
  self->frames[0].uv_data = (unsigned char*)fts_malloc(V_SIZE);
  self->frames[1].y_data = (unsigned char*)fts_malloc(V_SIZE);
  self->frames[1].uv_data = (unsigned char*)fts_malloc(V_SIZE);
  self->internal_change = FALSE;
  self->parameters_have_changed = 0;

  self->capture_mutex = SDL_CreateMutex();
  self->capture_cond = SDL_CreateCond();

  self->server_mutex = SDL_CreateMutex();
  self->server_cond = SDL_CreateCond();


  /* Start with initial black picture */
```

```
  memset(self->frames[0].y_data, 16, V_SIZE);
  memset(self->frames[0].uv_data, 128, V_SIZE);
  memset(self->frames[1].y_data, 16, V_SIZE);
  memset(self->frames[1].uv_data, 128, V_SIZE);


  /* Get the name of the device from command-line, or default to "/dev/video0" */
  for(i = 0; i < ac; i++)
  {
    if(fts_is_symbol(&at[i]) && !self->device)
      self->device = fts_get_symbol(&at[i]);
  }

  if(!self->device)
    self->device = fts_new_symbol("/dev/video0");

  /* Open the input device */
  if(vin_init_input_device(self))
  {
    /* Error!!! Something bad happened, we can't continue. */
    if(self->fd != -1) close(self->fd);
    self->fd = -1;
    return;
  }

  // vin_enable(o, 0, 0, 0, 0);

  fts_vdsp_add_object(o);
  fts_log("[vin#] opened device '%s' for video input\n", self->device);
}




/**
 * Object destructor.
 *
 * @param o A pointer to this object.
 * @param winlet <i>unused</i>
 * @param s <i>unused</i>
 * @param ac Argument count
 * @param at Argument array
 **/
static void vin_delete(fts_object_t* o, int winlet, fts_symbol_t s, int ac, const fts_atom_t* at)
{
  vin_t* self = (vin_t*)o;

  /* vin_close_input_device(self); */
  vin_disable(o, 0, 0, 0, 0);

  /* Destroy channels-info array */
  fts_free(self->channels);

  /* Destroy image buffers */
  fts_free(self->frames[0].y_data);
  fts_free(self->frames[0].uv_data);
  fts_free(self->frames[1].y_data);
  fts_free(self->frames[1].uv_data);

  /* Destroy synchronization resources */
  SDL_DestroyMutex(self->capture_mutex);
  SDL_DestroyCond(self->capture_cond);
```

```
  SDL_DestroyMutex(self->server_mutex);
  SDL_DestroyCond(self->server_cond);

  fts_vdsp_remove_object(o);
}



/**
 * Put method. Inserts it's code into the DSP chain upon request.
 *
 * @param o A pointer to this object.
 * @param winlet <i>unused</i>
 * @param s <i>unused</i>
 * @param ac Argument count
 * @param at Argument array
 **/
static void vin_put(fts_object_t* o, int winlet, fts_symbol_t s, int ac, const fts_atom_t* at)
{
  vin_t* self = (vin_t*)o;
  fts_vdsp_descr_t* vdsp = (fts_vdsp_descr_t*)fts_get_pointer(at);
  fts_atom_t a[4];

  fts_set_pointer(a, self);
  fts_set_symbol(a + 1, fts_vdsp_get_output_name(vdsp, 0));
  fts_set_symbol(a + 2, fts_vdsp_get_output_name(vdsp, 1));
  fts_set_int(a + 3, fts_vdsp_get_output_size(vdsp, 0));

  vdsp_add_funcall(vin_output_symbol, 4, a);
}



/**
 * DSP function. Grabs a frame from the webcam and
 * copies as much as possible from it to the buffer.
 *
 * Parameters are :
 * - self : a pointer to the object itself
 * - buffer : the buffer to which the stream will be written
 *
 * @param argv Array of parameters passed to the function by the DSP engine.
 **/
static void vin_output(fts_word_t* argv)
{
  vin_t* self = (vin_t*)fts_word_get_pointer(argv);
  unsigned char* out_y = (unsigned char*)fts_word_get_pointer(argv + 1);
  unsigned char* out_uv = (unsigned char*)fts_word_get_pointer(argv + 2);
  int output_size = fts_word_get_int(argv + 3) * 4;
  int output_frame = 1 - self->frame_to_load;

  /* Synchronize with VideoDSP */
  if(pdata[0].count == 0)
    self->offset = 0;

  output_size = MIN(output_size, V_SIZE - self->offset);

  memcpy(out_y, self->frames[output_frame].y_data + self->offset, output_size);
  memcpy(out_uv, self->frames[output_frame].uv_data + self->offset, output_size);
```

```
  self->offset += output_size;

  if(self->offset >= V_SIZE)
  {
    /* fts_post("out : %d\n", output_frame); */

    /* Start capturing next frame and swap frames
     * Next time, we'll start serving the freshly loaded frame, and
     * the thread will be capturing the next one in the background.
     *
     * This function blocks if the frame beeing captured has not yet finished.
     */
    if(self->state != cam_disabled)
      vin_start_capturing_next_frame(self);

    /* Set to beginning of frame */
    self->offset = 0;
  }
}




/**
 * Enable method. When the vin is enabled, the video device is opened.
 *
 * @param o A pointer to this object.
 * @param winlet <i>unused</i>
 * @param s <i>unused</i>
 * @param ac <i>unused</i>
 * @param at <i>unused</i>
 **/
static void vin_enable(fts_object_t* o, int winlet, fts_symbol_t s, int ac, const fts_atom_t* at)
{
  vin_t* self = (vin_t*)o;

  if(self->state == cam_disabled)
  {
    // Enable the video input device
    if(!vin_open_input_device(self))
    {
      self->state = cam_enabled;
      self->thread = SDL_CreateThread(vin_thread_routine, self);
      vin_start_capturing_next_frame(self);
    }
    else
      fts_post("[vin#] Error opening input device %s (%s)\n", self->caps.name, self->device);
  }
}




/**
 * Disable method. When the vin is disable, the video device is closed.
 *
 * @param o A pointer to this object.
 * @param winlet <i>unused</i>
 * @param s <i>unused</i>
 * @param ac <i>unused</i>
 * @param at <i>unused</i>
 **/
```

```
static void vin_disable(fts_object_t* o, int winlet, fts_symbol_t s, int ac, const fts_atom_t* at)
{
  vin_t* self = (vin_t*)o;

  if(self->state != cam_disabled)
  {
    self->state = cam_disabled;

    /* Unlock the capture thread */
    SDL_CondSignal(self->server_cond);

    /* Wait for the thread to finish up */
    /* This also cleans up resources held by the SDL_thread* object */
    SDL_WaitThread(self->thread, 0);

    /* Disable the video input device */
    if(vin_close_input_device(self))
      fts_post("[vin#] Error closing input device %s (%s)\n", self->caps.name, self->device);
  }
}
```

```
/**
 * Pause method. Invoking this method toggles the state of the camin object between the enabled
 * and the paused states. When the camin is paused, it leaves the video input device open but
 * stops capturing new images, sending out the same last image over and over again until the
 * camin is "unpaused" again (i.e. until it returns to the "enabled" state).
 *
 * @param o A pointer to this object.
 * @param winlet <i>unused</i>
 * @param s <i>unused</i>
 * @param ac <i>unused</i>
 * @param at <i>unused</i>
 **/
static void vin_pause(fts_object_t* o, int winlet, fts_symbol_t s, int ac, const fts_atom_t* at)
{
  vin_t* self = (vin_t*)o;

  if(self->state == cam_enabled)
  {
    // Pause the camin object
    self->state = cam_paused;
  }
  else if(self->state == cam_paused)
  {
    // Reenable the camin object
    self->state = cam_enabled;
  }

}
```

```
/**
 * Param method. This method is called when a numeric parameter of the camin object is changed.
 * These parameters typically are brightness, contrast, etc.
 *
 * @param o A pointer to this object.
 * @param winlet Number of the inlet of the parameter. Determines the parameter to change.
```

```
 * @param s <i>unused</i>
 * @param ac Argument count
 * @param at Argument array
 **/
static void vin_param(fts_object_t* o, int winlet, fts_symbol_t s, int ac, const fts_atom_t* at)
{
  vin_t* self = (vin_t*)o;
  int value = 0;

  if(self->internal_change) return;

  if(ac != 1) return;
  value = fts_get_int(at);

  if(value < 0) value = 0;
  if(value > 65535) value = 65535;

  switch(winlet)
  {
  case VIN_INLET_BRIGHTNESS:
    self->pic.brightness = value;
    break;

  case VIN_INLET_HUE:
    self->pic.hue = value;
    break;

  case VIN_INLET_COLOUR:
    self->pic.colour = value;
    break;

  case VIN_INLET_CONTRAST:
    self->pic.contrast = value;
    break;

  case VIN_INLET_WHITENESS:
    self->pic.whiteness = value;
    break;

  default:
    /* Oops! */
  }

  self->parameters_have_changed = TRUE;

  vin_output_image_params(o, 0, 0, 0, 0);
}

/************************************************************************/
/* Class configuration.                                                 */
/************************************************************************/

/**
 * Instantiation method.
 */
static void vin_instantiate(fts_class_t* cl)
{
  fts_class_init(cl, sizeof(vin_t), vin_init, vin_delete);

  fts_class_message_varargs(cl, fts_s_put, vin_put);

  fts_class_message_varargs(cl, VIN_S_ENABLE, vin_enable);
```

```
  fts_class_message_varargs(cl, VIN_S_DISABLE, vin_disable);
  fts_class_message_varargs(cl, VIN_S_PAUSE, vin_pause);
  fts_class_inlet_bang(cl, 0, vin_output_image_params);
  fts_class_inlet_int(cl, 0, vin_param);
  fts_class_inlet_int(cl, 1, vin_param);
  fts_class_inlet_int(cl, 2, vin_param);
  fts_class_inlet_int(cl, 3, vin_param);
  fts_class_inlet_int(cl, 4, vin_param);

  fts_vdsp_declare_outlet(cl, 0);
  fts_vdsp_declare_outlet(cl, 1);
  fts_class_outlet_varargs(cl, 2);

  vin_output_symbol = fts_new_symbol("vin_output");
  vdsp_declare_function(vin_output_symbol, vin_output);
}



/**
 * Class initilization function.
 */
void vin_class_config(void)
{
  VIN_S_ENABLE = fts_new_symbol("enable");
  VIN_S_DISABLE = fts_new_symbol("disable");
  VIN_S_PAUSE = fts_new_symbol("pause");

  fts_class_install(fts_new_symbol("vin#"), vin_instantiate);
}

/*****************************************************************************/
/* Video input device related code                                           */
/*****************************************************************************/

/**
 * This function initializes the video input device and queries it for it's
 * capabilities to make sure it can be used. It does not open the device.
 *
 * @param self Instance of vin_t.
 */
int vin_init_input_device(vin_t* self)
{
  int fd = -1;
  int i;

  /* Set errno to 0 to make sure we trap the latest error */
  errno = 0;

  self->bytes_per_pixel = 0;

  /* Open video device */
  fd = open(self->device, O_RDONLY);


  /* Error opening the device? */
  if(fd < 0)
  {
    fts_log("[vin#] error opening device '%s' for reading (errno = %d)\n", self->device, errno);
    fts_object_error(&self->super, "error opening device '%s' for reading\n", self->device);
    return -1;
```

```
}

self->fd = fd;

/* Query video device for capabilities */
memset(&self->caps, 0, sizeof(self->caps));
if(ioctl(fd, VIDIOCGCAP, &self->caps) == -1)
{
  fts_log("[vin#] could not get capabilities for device '%s'\n", self->device);
  fts_object_error(&self->super, "could not get capabilities for device '%s'\n", self->device);
  return -1;
}


/* Check that we can capture from this device */
if(!self->caps.type & VID_TYPE_CAPTURE)
{
  fts_log("[vin#] cannot capture images from device '%s' (%s)\n", self->device, self->caps.name);
  fts_object_error(&self->super, "cannot capture images from device '%s' (%s)\n",
       self->device, self->caps.name);
  return -1;
}

/* Get channels */
self->channels = (struct video_channel*)fts_malloc(self->caps.channels
               * sizeof(struct video_channel));
for(i = 0; self->channels && i < self->caps.channels; i++)
{
  self->channels[i].channel = i;
  if(ioctl(self->fd, VIDIOCGCHAN, &self->channels[i]) == -1)
  {
    fts_log("[vin#] cannot get information for channel %d on device %s (%s)\n",
      i, self->device, self->caps.name);
    fts_object_error(&self->super, "cannot get information for channel %d on device %s (%s)\n",
         i, self->device, self->caps.name);
  }
  else
    fts_post("[vin#] Using channel %d : %s\n", i, self->channels[i].name);
}


/* Query for picture capabilities and check that we support the given palette */
if(ioctl(fd, VIDIOCGPICT, &self->pic) == -1)
{
  fts_log("[vin#] could not query device '%s' (%s) for picture capabilities\n",
    self->device, self->caps.name);
  fts_object_error(&self->super, "could not query device '%s' (%s) for picture capabilities\n",
       self->device, self->caps.name);
  return -1;
}

switch(self->pic.palette)
{
case VIDEO_PALETTE_YUV420P:
  self->palette_handler = vin_handle_YUV420P;
  self->bytes_per_pixel = 2;
  break;

default:
  self->palette_handler = 0;
  break;
}
```

```
  /* Do we have a palette_handler? */
  if(!self->palette_handler)
  {
    char* msg = "cannot handle images produced by device";
    fts_log("[vin#] %s '%s' (%s)\n", msg, self->device, self->caps.name);
    fts_object_error(&self->super, "%s '%s' (%s)\n", msg, self->device, self->caps.name);
    return -1;
  }


  /* Ok, we have all the information we need about the device. Now, we can
   * open it again and start capturing as soon as we become "enabled".
   */

  close(self->fd);
  self->fd = -1;

  vin_output_image_params(&self->super, 0, 0, 0, 0);

  return 0;
}




/**
 * This function opens the video input device for reading. Usually, once
 * the video input device is open, it cannot be opened by another application
 * (or another instance of the vin_t object).
 *
 * @param self Instance of vin_t.
 * @return Returns 0 on success, non-null on error.
 */
static int vin_open_input_device(vin_t* self)
{
  int fd = -1;
  int bytes_per_tick = 0;
  int bytes_per_frame = 0;


  /* Set errno to 0 to make sure we trap the latest error */
  errno = 0;


  /* Open video device */
  fd = open(self->device, O_RDONLY);


  /* Error opening the device? */
  if(fd < 0)
  {
    fts_log("[vin#] error opening device '%s' for reading (errno = %d)\n", self->device, errno);
    fts_post("[vin#] Error opening video input device %s (%s)\n", self->caps.name, self->device);
    return -1;
  }

  self->fd = fd;


  /* Set channel to channel 0 */
  if(self->channels)
```

```
  {
    self->channels[0].norm = 0;
    self->channels[0].channel = 0;
    ioctl(self->fd, VIDIOCSCHAN, &self->channels[0]);
  }


  /* Create the memory mapping */
  self->mbuf.frames = 1;
  bytes_per_tick = fts_get_vtick_size() * 4;
  bytes_per_frame = V_SIZE * self->bytes_per_pixel;

  if(bytes_per_tick % bytes_per_frame == 0)
    self->mbuf.size = bytes_per_frame;
  else
    self->mbuf.size = bytes_per_tick * (bytes_per_frame + bytes_per_tick
          - bytes_per_tick % bytes_per_frame) / bytes_per_tick;

  self->mbuf.offsets[0] = 0;


  /*
    fts_post("using buffer of %d bytes\n", self->mbuf.size);
    fts_post("V_SIZE = %d, vtick_size = %d\n", V_SIZE, fts_get_vtick_size());
  */
  if(ioctl(self->fd, VIDIOCGMBUF, &self->mbuf) == -1)
  {
    // Add error report here - I'm getting lazy!
    fts_post("[vin#] Error opening video input device %s (%s)\n", self->caps.name, self->device);
    return -1;
  }

  self->data_size = self->mbuf.size;
  self->data = mmap(0, self->data_size, PROT_READ, MAP_SHARED, self->fd, 0);

  if(self->data == MAP_FAILED)
  {
    fts_log("[vin#] could not create memory mapping for device %s (%s)\n",
      self->device, self->caps.name);
    fts_post("[vin#] Error opening video input device %s (%s)\n",
       self->caps.name, self->device);
    return -1;
  }

  /* Initialize structure for capture */
  self->mmap.frame = 0;
  self->mmap.width = V_WIDTH;
  self->mmap.height = V_HEIGHT;
  self->mmap.format = self->pic.palette;

  /* Reset image parameters (they can have changed while the device was disabled) */
  ioctl(self->fd, VIDIOCSPICT, &self->pic);

  return 0;
}




/**
 * Closes the video input device.
 *
```

```
 * @param self Instance of vin_t.
 * @return Always returns 0.
 **/
static int vin_close_input_device(vin_t* self)
{
  /* Unmap the memory map */
  if(self->data) munmap(self->data, self->data_size);

  /* Close the input device */
  if(self->fd != -1) close(self->fd);

  return 0;
}

/*****************************************************************************/
/* Palette handlers                                                          */
/*****************************************************************************/

/**
 * This function handles a video stream in planar YUV 4:2:0 format.
 *
 * @param self Instance of vin_t.
 **/
static void vin_handle_YUV420P(vin_t* self)
{
  int frame;
  unsigned char* y_data;
  unsigned char* u_data;
  unsigned char* v_data;
  int i, j, k;

  SDL_mutexP(self->capture_mutex);
  frame = self->frame_to_load;
  SDL_mutexV(self->capture_mutex);

  y_data = self->data + self->mbuf.offsets[frame];
  u_data = self->data + self->mbuf.offsets[frame] + V_SIZE;
  v_data = self->data + self->mbuf.offsets[frame] + V_SIZE + (V_SIZE >> 2);
  memcpy(self->frames[frame].y_data, y_data, V_SIZE);

  for(i = 0, k = 0; i < (V_SIZE >> 2);)
  {
    for(j = 0; j < (V_WIDTH >> 1); j++, i++, k += 2)
    {
      self->frames[frame].uv_data[k]
  = self->frames[frame].uv_data[k + V_WIDTH]
  = u_data[i];

      self->frames[frame].uv_data[k + 1]
  = self->frames[frame].uv_data[k + V_WIDTH + 1]
  = v_data[i];
    }

    k += V_WIDTH;
  }
}

/*****************************************************************************/
/* Threading code                                                            */
/*****************************************************************************/

/**
```

```
 * Starts the capture of the next frame.
 **/
static void vin_start_capturing_next_frame(vin_t* self)
{
  SDL_mutexP(self->capture_mutex);

  /* Wait until no frame is beeing loaded */
  while(self->frame_beeing_loaded)
    SDL_CondWait(self->capture_cond, self->capture_mutex);

  /* Swap frames */
  self->frame_to_load = 1 - self->frame_to_load;

  /* Signal thread to start loading */
  self->frame_beeing_loaded = TRUE;
  SDL_mutexV(self->capture_mutex);
  SDL_CondSignal(self->server_cond);
}




/**
 * Thread startup routine. This basically waits for an image buffer to be filled, then
 * converts it to YUV4:2:2 and then starts over.
 *
 * @param arg Instance of vin_t.
 **/
static int vin_thread_routine(void* arg)
{
  vin_t* self = (vin_t*)arg;
  int i;

  while(self->state != cam_disabled)
  {
    /* Wait for main thread to tell us to go on */
    SDL_mutexP(self->capture_mutex);
    while(self->frame_beeing_loaded == FALSE && self->state != cam_disabled)
      SDL_CondWait(self->server_cond, self->capture_mutex);
    SDL_mutexV(self->capture_mutex);


    /* If we have to leave, we do it right away */
    if(self->state == cam_disabled)
    {
      SDL_mutexP(self->capture_mutex);
      self->frame_beeing_loaded = FALSE;
      SDL_mutexV(self->capture_mutex);
      SDL_CondSignal(self->capture_cond);
      break;
    }


    SDL_mutexP(self->capture_mutex);
    self->mmap.frame = self->frame_to_load;
    SDL_mutexV(self->capture_mutex);

    ioctl(self->fd, VIDIOCMCAPTURE, &self->mmap);

    /* If the image parameters have changed, we
     * can acknowledge them here, instead of
     * overloading the main thread.
```

```
     *
     * TODO : use a mutex here to protect that variable.
     */
    if(self->parameters_have_changed)
    {
      ioctl(self->fd, VIDIOCSPICT, &self->pic);
      self->parameters_have_changed = 0;
    }



    /* Wait for next frame to be loaded */
    ioctl(self->fd, VIDIOCSYNC, &self->mmap.frame);

    /* Convert the image - if the input device is "paused", then we keep our old image */
    if(self->state == cam_enabled)
    {
      self->palette_handler(self);
    }
    else if(self->state == cam_paused)
    {
      int src = 1 - self->mmap.frame;
      int dest = self->mmap.frame;
      memcpy(self->frames[src].y_data, self->frames[dest].y_data, V_SIZE);
      memcpy(self->frames[src].uv_data, self->frames[dest].uv_data, V_SIZE);
    }

    /* fts_log("in  : %d\n", self->frame_to_load); */

    SDL_mutexP(self->capture_mutex);
    self->frame_beeing_loaded = FALSE;
    SDL_mutexV(self->capture_mutex);
    SDL_CondSignal(self->capture_cond);
  }

  SDL_mutexP(self->capture_mutex);
  self->frame_beeing_loaded = FALSE;
  SDL_mutexV(self->capture_mutex);

  /* Wait for all images to finish loading */
  for(i = 0; i < 2; i++)
    ioctl(self->fd, VIDIOCSYNC, &i);

  return 0;
}

/********************************************************************************/
/* Other private methods                                                        */
/********************************************************************************/

static void vin_output_image_params(fts_object_t* o, int winlet, fts_symbol_t s,
            int ac, const fts_atom_t* at)
{
  vin_t* self = (vin_t*)o;
  fts_atom_t a[5];

  if(!self->internal_change)
  {
    fts_set_int(a + 0, self->pic.brightness);
    fts_set_int(a + 1, self->pic.hue);
    fts_set_int(a + 2, self->pic.colour);
```

```
    fts_set_int(a + 3, self->pic.contrast);
    fts_set_int(a + 4, self->pic.whiteness);

    self->internal_change = TRUE;
    fts_outlet_varargs(o, 2, 5, a);
    self->internal_change = FALSE;
  }
}


/** EMACS **
 * Local variables:
 * mode: c
 * c-basic-offset:2
 * End:
 */
```

Code source du programme `autoport`

```
#!/usr/bin/perl -w
#
# jMax
# Copyright (C) 1994, 1995, 1998, 1999 by IRCAM-Centre Georges Pompidou, Paris, France.
#
# This program is free software; you can redistribute it and/or
# modify it under the terms of the GNU Lesser Lesser General Public License
# as published by the Free Software Foundation; either version 2.1
# of the License, or (at your option) any later version.
#
# See file COPYING.LIB for further informations on licensing terms.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU Lesser General Public License for more details.
#
# You should have received a copy of the GNU Lesser Lesser General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
#
#
# This script is here to help porting jMax packages from version 2.5 to
# version 4.0. It saves the developer from doing stupid "search-and-replace"
# work by automatically changing deprecated function calls into their new counterparts
# and by doing other simple substitutions. What the "autoporter" cannot do, it
# informs the user of, giving indications as to what should be done and where.
#
# This script (as it says above) comes WITHOUT ANY WARRANTY. Make sure you do a
# complete backup of the sources of the package you wish to convert before
# using this tool.
#


### Program ######################################################################


print("This is autoport, version 0.1, by Carl Seleborg\n");
print("Copyright (C) 2003 by IRCAM-Centre Georges Pompidou, Paris, France\n");




# Parse command-line and return associative array with values set.
my %options = &ParseArguments();
my $package_name = $options{"package-name"};
my $package_directory = $options{"src-location"};
print("Could not find directory '$package_directory' - abort\n") and exit(1)
  if !(-e $package_directory && -d $package_directory);


# Make backup if requested
if($options{'copy-location'})
{
  if(!-e $options{'copy-location'} && system "mkdir $options{'copy-location'}")
  {
    print("autoport : error : failed to create directory $options{'copy-location'} - abort\n");
    exit(1);
  }
  if(system "cp -dpr $options{'src-location'}/* $options{'copy-location'}")
```

```
  {
    print("autoport : error : failed to copy source directory - abort\n");
    exit(1);
  }
}



print("autoport : Attempting port of package " . $package_name . " ...\n");


# Gather list of files
my @files = &GetListOfFiles();

# print("autoport : found the following source files :\n");
# foreach $file (@files)
# {
#     print("\t$file\n");
# }


# Scan all files and locate important information
%info = &ScanFilesAndCollectInfo(\@files);
$info{"package-name"} = $package_name;


# Checking that all is there
if(!($info{"package-init"}{"name"}))
{
  printf("autoport : warning : could not find module-init function (port already done?)\n");
}


&DoSubstitutionsInFiles(\@files, \%info);



### Functions ###########################################################################

###
# Gets the list of C source files in the package's directory and returns an
# array of them.
#
sub GetListOfFiles
{
  # The list of filenames
  @list = ();

  open(FILELIST, 'find ' . $package_directory . ' -name "*.[ch]" -print |')
    or print("autoport : error : failed to get list of files - abort\n") and exit(1);
  while($filename = <FILELIST>)
  {
    chomp($filename);
    push(@list, $filename);
  }

  return @list;
}


###
# Prints usage syntax.
```

```
#
sub PrintUsage
{
  print("Usage : autoport name-of-package [options]\n");
  print(" Options can be the following :\n");
  print("  -c dir : If specified, copies the source directory to the 'dir' directory\n");
  print("           before modifying the files\n");
  print("  -d dir : If specified, tells autoport where to find the sources\n");
  print("           (default is 'name-of-package/')\n");

  return 1;
}



###
# Parses the command-line arguments and returns an associative array
# with key-value pairs. The following keys are returned :
#  - package-name : name of the package
#  - copy-location : if set, tells where to make the backup
#  - src-location : tells where the source files are
#
sub ParseArguments
{
  my %values;
  my $arg;

  &PrintUsage() and exit(1) if $#ARGV == -1;

  $values{'package-name'} = shift @ARGV;
  $values{'src-location'} = $values{'package-name'};

  while($#ARGV > -1)
  {
    $arg = shift @ARGV;

    if($arg eq "-c")
    {
      print("autoport : error : -c must be followed by directory name - abort\n") and exit(1)
  if $#ARGV == -1;

      $values{'copy-location'} = shift @ARGV;
    }
    elsif($arg eq "-d")
    {
      print("autoport : error : -d must be followed by directory name - abort\n") and exit(1)
  if $#ARGV == -1;

      $values{'src-location'} = shift @ARGV;
    }
    else
    {
      print("autoport : error : unknown option '$arg' - abort\n");
      exit(1);
    }
  }

  return %values;
}


###
```

```
# Reads and stores a whole file into a string
#
# @param file Name of the file to load
#
sub LoadFile
{
  my $filename = shift;
  my $contents = "";
  my $chunk;

  open(FILEHANDLE, "<" . $filename)
    or print("autoport : error : failed to open file '$filename' for reading - abort\n") and exit(1);

  while(!eof(FILEHANDLE))
  {
    read(FILEHANDLE, $chunk, 8192);
    $contents .= $chunk;
  }

  close(FILEHANDLE);

  return $contents;
}



###
# Writes a file back to disk from a string
#
# @param filename Name of the file
# @param contents Contents of the file
#
sub WriteFile
{
  my ($filename, $contents) = @_;

  open(FILEHANDLE, ">" . $filename)
    or print("autoport : error : failed to open file '$filename' for writing - abort\n") and exit(1);

  print("autoport : error : failed to write to file '$filename' - abort\n") and exit(1)
    if(syswrite(FILEHANDLE, $contents, length $contents) == -1);
}


###
# Scans all files in the list of files and collects information about
# modules, classes, objects, etc.
#
# @param files List of files to scan
# @return A hash containing the information :
#   - package-init
#      - name : name of the function
#      - file : file containing that function
#
sub ScanFilesAndCollectInfo (\@)
{
  my $ref_files = shift;
  my ($file, $conents);
  my $struct_member = '\s*([^,]+|"[^"]*")\s*';
  my %result;

  foreach $file (@$ref_files)
```

```
  {
    $contents = &LoadFile($file);

    # printf("autoport : scanning file '%s'...\n", $file);

    # Search for module initialization functions
    if($contents =~
       /fts_module_t \s+ \w+ \s* = \s* \{ $struct_member , $struct_member ,
       $struct_member , $struct_member , $struct_member \}\s*;/x)
    {
      $result{"package-init"}{"name"} = $3;
      $result{"package-init"}{"file"} = $file;
      # printf("autoport : (%s) found module initialization function : '%s'\n", $file, $3);
    }
  }

  return %result;
}


###
# Does substitutions in the given files
#
# @param files List of files to modify
# @param info The information previously collected by ScanFilesAndCollectInfo
#
sub DoSubstitutionsInFiles (\@\%)
{
  my ($ref_files, $ref_info) = @_;
  my ($file, $contents, $old, $new);

  print("autoport : processing files :\n");

  foreach $file (@$ref_files)
  {
    $contents = &LoadFile($file);

    # Remove module declaration (fts_module_t)
    $old = "fts_module_t\\s*[^;]*;";
    $new = "";
    $contents = &DoSubstitutionsInString($contents, $old, $new, 1);


    # Substitute package-module init function name
    # from $info{"package-init"}{"name"} to package_config
    if($ref_info->{"package-init"}{"name"})
    {
      $old = 'void\s+' . $ref_info->{"package-init"}{"name"};
      $new = 'void ' . $ref_info->{"package-name"} . "_config";
      $contents = &DoSubstitutionsInString($contents, $old, $new);
    }


    # fts_new_symbol_copy -> fts_new_symbol
    $old = 'fts_new_symbol_copy';
    $new = 'fts_new_symbol';
    $contents = &DoSubstitutionsInString($contents, $old, $new);

    # Replace "post" with "fts_post"
    $contents = &DoSubstitutionsInString($contents, 'post', 'fts_post');

    # Replace ivec_atom_get with (ivec_t*)fts_get_object
```

```
    $contents = &DoSubstitutionsInString($contents, "ivec_atom_get", "(ivec_t*)fts_get_object");


    # Replace fvec_atom_get with (fvec_t*)fts_get_object
    $contents = &DoSubstitutionsInString($contents, "fvec_atom_get", "(fvec_t*)fts_get_object");


    # Replace vec_atom_get with (vec_t*)fts_get_object
    $contents = &DoSubstitutionsInString($contents, "vec_atom_get", "(vec_t*)fts_get_object");


    # Remove fts_symbol_name macro
    $contents = &DoSubstitutionsInString($contents, "fts_symbol_name", "", 1);


    # Instanciation and constructor/destructor declaration
    $contents = &FiddleWithInstanciationFunctions($contents);


    # Replace obsolete calls to fts_bloc_malloc() and friends.
    $contents = &DoSubstitutionsInString($contents, "fts_block_alloc", "fts_malloc");
    $contents = &DoSubstitutionsInString($contents, "fts_block_zalloc", "fts_zalloc");
    $contents = &DoSubstitutionsInString($contents, "fts_block_free", "fts_free");
    $contents = &DoSubstitutionsInString($contents, "fts_do_block_alloc", "fts_do_malloc");
    $contents = &DoSubstitutionsInString($contents, "fts_do_block_zalloc", "fts_do_zalloc");
    $contents = &DoSubstitutionsInString($contents, "fts_do_block_free", "fts_do_free");



    # Replace fts_method_define calls by fts_class_inlet_* or fts_class_message_*
    $contents = &DoMethodDefinitionReplacements($contents, $file);


    # Replace fts_get_ptr by ftr_get_pointer
    $contents = &DoSubstitutionsInString($contents, "fts_get_ptr", "fts_get_pointer");
    $contents = &DoSubstitutionsInString($contents, "fts_word_get_ptr", "fts_word_get_pointer");
    $contents = &DoSubstitutionsInString($contents, "fts_set_ptr", "fts_set_pointer");
    $contents = &DoSubstitutionsInString($contents, "fts_word_set_ptr", "fts_word_set_pointer");


    # Warn if fts_alarm_t or fts_file_get_read_path are used
    if($contents =~ /fts_alarm_t/)
    {
      printf("autoport : warning : fts_alarm_t has been replaced by FTS timebases\n" .
       "                     (see fts_timebase*() functions in <fts/time.h>) (in file %s)\n", $file);
    }
    if($contents =~ /fts_file_get_read_path/)
    {
      printf("autoport : warning : fts_file_get_read_path must not be used anymore (in file %s)\n",
       $file);
    }
    if($contents =~ /fts_param_/)
    {
      printf("autoport : warning : use of param has changed. Use fts_param_t objects instead now,\n" .
       "                     see <fts/param.h> for details. (in file %s)\n", $file);
    }

    &WriteFile($file, $contents);

    printf("\t%s\n", $file);
  }
}
```

```
###
# Substitutes a s2 with s3 (along with a comment to declare the modification) in s1.
#
# @param s1 The string to search
# @param s2 The string to find
# @param s3 The string to replace s2 with
# @param remove If true, the matched pattern is removed
#
# @return The modified s1
#
sub DoSubstitutionsInString
{
  my ($s1, $s2, $s3, $s4) = @_;
  my $r = $s1 =~ /(^(.*)($s2))/mgx;

  return $s1 unless $r;

  $old_trimed = Trim($3);
  $new_trimed = Trim($s3);

  if(!$s4)
  {
    $s1 =~ s/^(.*)($s2)/\/\* CODE MODIFIED BY autoport :
      \'$old_trimed\' -> \'$new_trimed\' \*\/\n$1$s3/mgx;
  }
  else
  {
    $s1 =~ s/^(.*)($s2)/\/\* CODE REMOVED BY autoport : \'$2\' \*\/\n$1/mgx;
  }
  return $s1;
}


sub Trim
{
  $_ = shift;
  s/^\s*(.*)/$1/;
  s/(.*\S)\s*$/$1/;
  return $_;
}


###
# Fiddles with the instanciation functions in a source file.
#
# @param contents The string holding the contents of the file
#
# @return Returns the modified contents of the file
#
sub FiddleWithInstanciationFunctions
{
  my $file_contents = shift;
  my $r = 0;
  my $sep = '\s*,\s*';
  my $fun_ptr = '(\(\w+\))?\w+';
  my $regexp1 = '((fts_status_t\s+(\w+)_instantiate\s*' .
                '\(\s*fts_class_t\s*\*\s*(\w+)\s*,[^)]*\))\s*\{([^\n]|\n[^}])*\n\})';
  my $regexp2 = '((fts_status_t\s+(\w+)_instantiate\s*' .
                '\(\s*fts_class_t\s*\*\s*(\w*)\s*,[^)]*\))\s*;)';
```

```
  # First step : find an unmodified instanciation function :
  $r = $file_contents =~ /$regexp1/;

  while($r)
  {
    $class_arg_name = $4;
    $obj_name = $3;
    $contents = $1;
    $old_contents = $1;
    $constructor_name = "";
    $destructor_name = "";
    $constructor_statement = "";
    $destructor_statement = "";

    # printf("%d\n", $contents =~ quotemeta($2));
    $def = quotemeta($2);
    $contents = &DoSubstitutionsInString($contents, $def,
           "void ${obj_name}_instantiate(fts_class_t* $class_arg_name)");
    # $contents =~ s/$def/void ${obj_name}_instantiate(fts_class_t* $class_arg_name)/;

    # Search for constructor
    if($contents =~ /(^.*fts_method_define\w*\s*\(\s*[^,]+,\s*fts_SystemInlet\s*,
          \s*fts_s_init\s*,\s*($fun_ptr)[^)]*\)\s*;)/mx)
{
  # printf("Found constructor declaration : %s (constructor named %s)\n", $1, $2);
  $constructor_name = $2;
  $constructor_statement = $1;

  # Remove it from function code
  $contents = &DoSubstitutionsInString($contents, quotemeta($1), "", 1);
}

# Search for destructor
if($contents =~ /(^.*fts_method_define\w*\s*\(\s*[^,]+,\s*fts_SystemInlet\s*,
      \s*fts_s_delete\s*,\s*($fun_ptr)[^)]*\)\s*;)/mx)
{
  # printf("Found destructor declaration : %s (destructor named %s)\n", $1, $2);
  $destructor_name = $2;
  $destructor_statement = $1;

  # Remove it from function code
  $contents = &DoSubstitutionsInString($contents, quotemeta($1), "", 1);
}

# Search for class initialization
if($contents =~ /(fts_class_init\s*\(\s*(\w+) $sep ([^,]*) $sep [^)]*\))\s*;/mx)
{
  $constructor_name = "0" if !$constructor_name;
  $destructor_name = "0" if !$destructor_name;
  $contents = &DoSubstitutionsInString($contents, quotemeta($1),
              "fts_class_init\($2, $3, $constructor_name, $destructor_name\)");
}

# Search for return statements
while($contents =~ /[^\']\(return\s+(\S+\s*);)/mx)
{
  $contents = &DoSubstitutionsInString($contents, quotemeta($1), "return;");
}


# printf("autoport : found instantiation function for object %s (class param %s) : \n%s\n",
```

```
#       $obj_name, $class_arg_name, $contents);

$quoted = quotemeta($old_contents);
$file_contents =~ s/$quoted/$contents/gmx;
$r = $file_contents =~ /$regexp1/;
}



# Treat function declarations the same way
if($file_contents =~ /$regexp2/gxm)
{
  $class_arg_name = $4;
  $obj_name = $3;
  $def = quotemeta($2);
  $file_contents = &DoSubstitutionsInString($file_contents, $def,
              "void ${obj_name}_instantiate(fts_class_t* $class_arg_name)");
}

return $file_contents;
}


###
# Replaces method definitions by the new calls to fts_class_inlet* and fts_class_define*
#
# @param contents The text to modifiy
#
# @return The modified text
#
sub DoMethodDefinitionReplacements
{
  $_ = shift;
  $filename = shift;


  while(/(^.*[^\'\s]\s*|^\s*)(fts_method_define(\w*)([^;]*)\s*;)/mgx)
  {
    $instr = $2;
    $suffix = $3;
    $params = $4;

    $params =~ /\s*\(([^,]*),([^,]*),.*\)$/m;
    $class = $1;
    $winlet = $2;
    $method = "";

    printf("FOUND instruction <%s> in file %s : ", $instr, $filename);


    # We have two cases : either the method is declared to take a single atom as argument,
    # (because it's declared with a fts_s_[type] argument or because it's defined using
    # a fts_method_define_[type] version of the fts_method_define function), either
    # the method is a little more sophisticated (with a varargs and other strange messages).
    if($params =~ /fts_s_(int|float|number|symbol|bang|list|anything|void)/
       || $suffix =~ /_(int|float|number|symbol|bang|list|anything|void)/)
    {
      print("atom");

      if($params =~ /fts_SystemInlet/)
      {
  printf("autoport : warning : %s\n", $instr);
```

```
  printf("                   : defining a method for fts_SystemInlet that matches a " .
         "fts_s_[atomic] selector is probably wrong (in file %s)\n", $filename);
      }

      # Deduce type from suffix or from parameter?
      $type = "";
      if($suffix =~ /_(int|float|number|symbol|bang|list|anything|void)/)
      {
  $type = $1;
  $params =~ /\s*\(([^,]*,){2} (.*)\)/x;
  $method = $2;
      }
      elsif($params =~ /fts_s_(int|float|number|symbol|bang|list|anything|void)/)
      {
  $type = $1;
  $params =~ /\s*\(([^,]*,){3} ([^,]*) (,.*)?\)/x;
  $method = $2;
      }

      $_ = &DoSubstitutionsInString($_, quotemeta($instr),
            "fts_class_inlet_$type($class,$winlet,$method);");
    }
    else
    {
      print("message");

      # Here, two cases again : if fts_SystemInlet is specified, we must use the
      # fts_class_message_varargs() function. Otherwise, we use the fts_class_inlet_varargs()
      # function.
      $params =~ /\s*\(([^,]*), ([^,]*), ([^,]*), ([^,]*) (,.*)?\)/x;
      $symbol = $3;
      $method = $4;

      if($params =~ /fts_SystemInlet/)
      {
  $_ = &DoSubstitutionsInString($_, quotemeta($instr),
              "fts_class_message_varargs($class,$symbol,$method);");
      }
      else
      {
  $_ = &DoSubstitutionsInString($_, quotemeta($instr),
              "fts_class_inlet_varargs($class,$symbol,$method);");
      }
    }

    print("\n");

  }

  return $_;
}
```

Glossaire

ALSA *(p. 10)* : Le projet *Advanced Linux Sound Architecture* vise à doter Linux d'une architecture de périphériques et de pilotes audio performante et puissante. Plusieurs systèmes audio ont été développés sous Linux, mais ALSA semble être le plus prometteur et est adopté par la plupart des logiciels de traitement du son et de création musicale sous Linux.

C *(p. 6)* : Le langage C est probablement le langage de programmation le plus répandu. Bien qu'il soit ancien, il est toujours d'actualité grâce à sa puissance et à son efficacité. Le langage C possède de très nombreuses implémentations, tant libres que propriétaires, et on en trouve sur la plupart des architectures matérielles existantes. Dans les années 90, un langage orienté-objet, le C++, est parti du C comme base pour y rajouter des fonctionnalités propres à son paradigme objet.

Architecture client/serveur *(p. 6)* : Ce type d'architecture décrit un certain découpage fonctionnel d'un système informatique. Dans un logiciel monolithique, les fonctionnalités du logiciel et la communication avec l'utilisateu sont écrites dans un même programme. Dans une architecture de type client/serveur, le système est découpé en deux parties distinctes, l'une effectuant le travail pour lequel le logiciel a été créé (le serveur), et l'autre permettant à l'utilisateur de communiquer avec le serveur afin de contrôler ce travail. Un navigateur Internet est le client d'un serveur Internet : ce dernier fournit des contenus (il doit être efficace et sécurisé), alors que le premier affiche ces contenus (il doit être simple à utiliser, afficher correctement les pages et exister sur diverses plate-formes). Un tel découpage permet de rendre un système plus souple en séparant correctement les fonctions. Dans le cas de jMax, l'interface graphique représente le client, alors que FTS représente le serveur.

DIPS *(p. 11)* : *Digital Image Processing with Sound* est le module de synthèse 3D pour jMax développé par Digital Art Creation. Il utilise OpenGL comme bibliothèque de primitives 3D pour calculer le rendu d'une scène 3D.

DirectSound *(p. 11)* : Microsoft, en souhaitant doter son système Windows d'une puissante bibliothèque de fonctions multimedia, a mis au point DirectX. Cette bibliothèque se découpe en plusieurs parties, et DirectSound est la partie qui concerne le son et les entrées/sortie audio du système.

FTS *(p. 6)* : *Faster Than Sound* est le moteur de calcul temps-réel de jMax.

GNU *(p. 1)* : Le projet GNU, dont l'acronyme récursif dévoile un ceratin côté revendicatif (***G****NU's* ***N****ot* ***U****NIX*, GNU n'est pas UNIX), fut lancé en 1984 dans l'espoir de fournir aux utilisateurs une version d'UNIX libre et

gratuite, avec tous les outils associés. Plus tard, le site Internet `www.gnu.org` est également devenu le site officiel de la *Free Software Foundation*, une organisation qui milite pour la défense et la diffusion du logiciel libre.

Java *(p. 6)* : Java est le langage de programmation multi-plate-formes créé par Sun Micrososystems en 1995, et qui est rapidement devenu très populaire. Il prend ses sources syntaxiquement dans les langages C et C++, mais s'est débarassé de certaines notions jugées trop complexes. Aujourd'hui, le marché couvert par Java est gigantesque, du fait de sa simplicité et de sa puissance.

Lisp *(p. 3)* : Lisp, dont le nom provient de la contraction de *List Processing*, est une famille de langages regroupant des variantes fonctionnelles, impératives et mêmes orientées-objet. Lisp est devenu très populaire par sa simplicité syntaxique et sa très grande expressivité. Les principaux dialectes de Lisp aujourd'hui sont Scheme et Common Lisp. Le très célèbre éditeur de texte Emacs repose très largement sur sa propre version de Lisp (elisp), dont il tire la puissance qui fait sa renommée.

OpenGL *(p. 11)* : *OpenGL*, pour *Open Graphics Library*, est une bibliothèque de fonctions graphiques destinées à la synthèse 3D d'images. Cette bibliothèque libre d'utilisation est très populaire dans le monde de l'imagerie 3D.

Paradigme *(p. 6)* : Un paradigme est, selon la définition de Kuhn, une école de pensée, une philosophie. Dans le domaine des langages de programmation, on retrouve quelques grands paradigmes comme la forme procédurale ou la forme orienté-objet. Le paradigme d'un langage de programmation va souvent définir l'expressivité qu'il met au service de l'utilisateur. Ainsi, le langage implémenté par jMax ne permet pas d'exprimer le même type d'idées aussi simplement qu'un langage textuel orienté objet.

Perl *(p. 15)* : Perl, pour *Practical Extraction and Reporting Language*, est un langage interprété spécialisé dans la manipulation de texte. Il est très utilisé pour des tâches d'administration de systèmes UNIX et fut pendant longtemps le langage de choix pour créer des sites internet à contenu dynamique (grâce aux scripts CGI).

QuickTime *(p. 16)* : Apple, le créateur de Macintosh, a mis au point il y a quelques années un format de fichier video nommé QuickTime. Très populaire, ce format est devenu un grand standard pour les fichiers video, tant pour les Macintosh que pour les PC. Comme pour d'autres formats de fichiers video tels que AVI de chez Microsoft, QuickTime ne précise pas un *codage* particulier du flux video — c'est le rôle du codec. On peut donc tout à fait avoir un afficheur QuickTime qui n'est pas capable de lire un fichier

video au format QuickTime utilisant un codec qui n'a pas été préalablement installé.

Swing *(p. 31)* : Le langage Java est livré, depuis sa version 1.2 avec une vaste bibliothèque de composants graphiques multi-plate-formes nommée Swing. Elle permet de créer des interfaces graphiques pour des applications Java. Swing repose en grande partie sur des modèles de conception objet très bien étudiés.

Temps-réel *(p. 1)* : Young donne dans [9] la définition suivante d'un système temps-réel : « *Le terme* temps-réel *sert à décrire toute activité de traitement de l'information ou tout système ayant à répondre à des stimuli d'entrée générés extérieurement dans un délai* fini *et* spécifiable. »

Références

[1] François Déchelle, Riccardo Borghesi, Maurizio De Cecco, Enzo Maggi, Butch Rovan, and Norbert Schnell. Jmax : An environment for real-time musical applications. *Computer Music Journal*, 1999.

[2] John L. Hennesby and David A. Patterson. *Architecture des ordinateurs, une approche quantitative*. McGraw-Hill, 1992.

[3] Alfred Raho, Ravi Sethi, and Jeffrey Ullman. *Compilateurs : Principes, techniques et outils*. InterEditions, 1989.

[4] Carl Seleborg. Creating an external package for jmax. *Les logiciels libres de l'Ircam*, 2003. http ://freesoftware.ircam.fr.

[5] Carl Seleborg. Porting packages to jmax 4.0. *Les logiciels libres de l'Ircam*, 2003. http ://freesoftware.ircam.fr.

[6] *Video4Linux Kernel API Reference*, 1999. http ://www.kernel.org/pub/linux/kernel/people/marcelo/linux-2.4/Documentation/video4linux/API.html.

[7] Hugues Vinet and François Delelande. *Intefaces homme-machine et création musicale*. HERMES Science Publications, 1999.

[8] Dave Wilson. The (almost definitive) fourcc guide. *Conexant Systems Inc.*, 2002. http ://www.fourcc.org.

[9] S. J. Young. *Real-Time languages : Design and Development*. Elis Horwood Publishers, 1982.

Index