Jean BRESSON

Rapport de stage de 3ème année
Mai – Septembre 2003

Représentation et manipulation de données d'analyse sonore pour la composition musicale.

Stage encadré par Carlos AGON

Institut de Recherche et Coordination Acoustique / Musique
Ircam-CNRS UMR 9912

Equipe Représentations Musicales

Sommaire

INTRODUCTION **3**

REMERCIEMENTS **4**

1. PRÉSENTATION **5**
1.1. L'IRCAM 5
1.2. L'ÉQUIPE REPRÉSENTATIONS MUSICALES ET LE LOGICIEL OPENMUSIC 6

2. INTRODUCTION AU PROJET **9**
2.1. CADRE DU PROJET : OMCHROMA 9
2.2. LES OBJECTIFS DU PROJET 10
2.3. LE FORMAT SDIF 11

3. DÉROULEMENT DU STAGE **13**
3.1. PREMIÈRES ÉTAPES 13
3.2. DÉVELOPPEMENT 14

4. UN VISUALISATEUR SDIF EN 3D **15**
4.1. LECTURE D'UN FICHIER SDIF 15
4.2. REPRÉSENTATION DES DONNÉES 17
Structure des données *18*
Représentation graphique *19*
Cas spéciaux *21*
4.3. NAVIGATION 22
Mouvements *22*
Positionnement *22*
Sélection d'un point *22*
Vue des différents champs de la matrice *23*
Sélection d'une fenêtre temporelle *24*
4.4. EDITION 25
Sauvegarde des modifications *25*
4.5. BILAN DE CETTE PREMIÈRE PHASE DE DÉVELOPPEMENT 26

5. UNE APPLICATION PLUS ÉLABORÉE… **27**
5.1. INTERFACE 27
5.2. GESTION DES FENÊTRES ET DE L'EXÉCUTION 29
Structure de l'application *30*
5.3. EDITION DES DONNÉES 31
5.4. PRÉFÉRENCES DE VISUALISATION 34
Couleurs *34*
Points de vue *35*
5.5. VUES SPÉCIALES 36
Coupes 2D *36*
Sonagramme *37*
5.6. REPRÉSENTATION SONORE 38
Un fichier audio attaché *38*
"Ecouter" les données *38*

6. L'INTÉGRATION DANS OPENMUSIC **39**
6.1. UN ÉDITEUR POUR LES OBJETS *SDIFFILE* 39
6.2. UTILISATION DANS OMCHROMA 41
L'analyse par partiels *41*
Un nouveau type SDIF *42*
Visualisation de partiels SDIF *43*

7. CONCLUSION **44**

RÉFÉRENCES : **45**
Annexe 1 : Planning du stage : tableau récapitulatif *46*

Introduction

Ce rapport présente le stage que j'ai effectué à l'IRCAM (Institut de Recherche et de Coordination Acoustique / Musique), au sein de l'équipe de Représentations Musicales, et dans le cadre de ma troisième année d'école d'ingénieur à l'ESSI (Ecole Supérieure en Sciences Informatiques, Université de Nice - Sophia Antipolis), de mai à septembre 2003.

Ce stage à pour objectif l'étude et le développement d'un outil pour la représentation et la manipulation de données d'analyse sonore dans le contexte de la composition musicale assistée par ordinateur. Il s'inscrit dans le cadre du projet Chroma, visant à relier les domaines distincts que sont la composition musicale assistée par ordinateur, où l'on traite et manipule les structures musicales en tant qu'objets informatiques, et celui de l'analyse et de la synthèse sonore, considérant des notions se rapportant plutôt aux techniques de traitement du signal. En outre, il s'agira d'exploiter le format d'échange de données de description sonore SDIF utilisé en analyse, en créant une application permettant d'insérer ce dernier dans un environnement de composition.

Dans une première partie, je présenterai l'environnement et le cadre du stage, pour arriver, dans une deuxième partie à une spécification plus précise du projet. Dans les parties suivantes seront exposées les différentes étapes et réalisations du développement de l'application, jusqu'à sa finalisation et son intégration dans les systèmes auxquels elle est destinée.

Remerciements

Je tiens à remercier tout d'abord Carlos Agon, qui a dirigé et soutenu mon travail, et Gérard Assayag, responsable de l'équipe Représentations Musicales, qui m'ont accueilli et m'ont permis de réaliser ce stage à leurs côtés.

Un grand merci également à Marco Stroppa, qui a permis à ce projet de voir le jour, ainsi qu'à Karim Haddad pour son aide et sa compagnie.

Merci aussi à Diemo Schwartz et aux membres de l'équipe d'analyse/synthèse, à Patrice Tisserand, de l'équipe Logiciels Libres, qui m'ont apporté leurs conseils pendant la réalisation de ce projet.

Enfin je remercie tous ceux qui, à mes côtés, ont fait de ce stage une expérience agréable et enrichissante professionnellement, humainement, musicalement.

1. Présentation

1.1. L'IRCAM

Fondé en 1969 par Pierre Boulez, l'Ircam (Institut de Recherche et Coordination Acoustique / Musique *[1]*) est une institution musicale associée au Centre Georges Pompidou. L'activité de l'Ircam se décline selon trois axes principaux□la recherche et le développement, qui sera le cadre de ce stage et de ce rapport, la création, principalement dans le domaine de la musique contemporaine, et la pédagogie, avec diverses formations et ateliers proposés aux compositeurs, mais également une formation doctorale, et de nombreuses conférences.

L'un des principaux objectifs de l'Ircam est de susciter une interaction entre recherche scientifique, développement technologique et création musicale contemporaine. L'Ircam invite en effet dans ses studios de nombreux compositeurs qui viennent y réaliser des œuvres associant interprètes classiques et nouvelles technologies.

La recherche fondamentale porte sur les apports de l'informatique, de la physique et de l'acoustique à la problématique musicale, et tente de contribuer, par les sciences et techniques, au renouvellement de l'expression musicale, en particulier par la mise au point d'outils logiciels destinés à des applications musicales. Réciproquement, les problèmes spécifiques posés par la création contemporaine donnent lieu à des problématiques scientifiques originales tant théoriques, méthodologiques, qu'appliquées. Ainsi, l'activité de l'Ircam s'inscrit dans le cadre de nombreuses collaborations avec des institutions universitaires ou de recherche en France et à l'étranger, ainsi qu'avec le monde industriel.

En termes de développement, il s'agit d'adapter les modèles et prototypes issus de la recherche sous la forme d'outils et environnements logiciels. Parmi ceux-ci, on peut citer *AudioSculpt* (éditeur de sonagrammes), *OpenMusic* (composition assistée par ordinateur), *Diphone*, *Modalys* (synthèse sonore), *Spat* (spatialisateur temps réel), *jMax* (interaction musicale et multimédia temps réel). Le développement des logiciels comprend en particulier la conception d'interfaces homme-machine adaptées, la mise en œuvre de protocoles de communication entre applications destinée à assurer une cohérence d'utilisation de l'ensemble des outils, et l'intégration permanente de technologies issues d'une industrie informatique en très rapide évolution. Le Forum Ircam *[2]* favorise la diffusion de ces logiciels auprès d'une communauté musicale évaluée, en 2001, à plus de 1500 utilisateurs. Des cessions de licences sont également accordées à des partenaires extérieurs, pour leur utilisation propre ou pour la commercialisation des logiciels.

Les thèmes de recherche et développement s'organisent autour de plusieurs équipes : l'équipe **Acoustique instrumentale**, qui étudie le fonctionnement des instruments de musique en élaborant des modèles acoustiques ; l'équipe **Acoustique des salles**, qui étudie l'effet de la propagation du son dans les lieux d'écoute ; l'équipe **Perception et cognition musicales** étudie les mécanismes intervenant dans l'écoute musicale ; l'équipe **Applications temps réel**, qui met au point des dispositifs informatiques destinés au traitement en temps réel des informations musicales et sonores et, plus largement, multimédia ; l'équipe **Analyse/synthèse** développe des procédés de synthèse et de transformation des sons, à partir de méthodes de traitement de signal ou de synthèse par modélisation physique ; et enfin l'équipe **Représentations musicales**, qui développe des environnements de composition assistée par ordinateur, et dont nous allons développer par la suite l'activité puisque c'est dans celle-ci que se situe le cadre de mon projet.

1.2. L'équipe Représentations Musicales et le logiciel OpenMusic

L'équipe de Représentations Musicales développe des environnements de composition assistée par ordinateur (CAO). Son activité s'appuie fortement sur une collaboration entre chercheurs, compositeurs, musicologues, qui a permis de mettre à jour des outils aujourd'hui très utilisés pour la composition d'œuvres contemporaines, notamment pour l'élaboration de parties instrumentales.

Les travaux sur la Composition Assistée par Ordinateur (CAO) visent à étudier la structure formelle de la musique. Il en résulte des environnements de CAO fondés sur le calcul symbolique utilisant des structures de données classiques (e.g. listes, arbres, graphes) et des algorithmes pour représenter et manipuler des structures concernant un processus de composition.

Du fait de la diversité de modèles esthétiques, techniques, formels qui coexistent chez les compositeurs, on ne peut plus considérer un environnement de CAO comme une application figée offrant une collection finie de procédures de génération et de transformations musicales. Au contraire, l'équipe Représentations Musicales conçoit un tel environnement comme un langage, au sens informatique, à l'aide duquel chaque compositeur pourra constituer son univers personnel. Bien entendu, il ne s'agit pas de fournir un langage informatique traditionnel, dont la maîtrise demande une grande expertise technique, mais un langage aménagé spécialement pour le compositeur. Ceci amène à une réflexion sur les différents modèles de programmation existants, sur les interfaces, de préférence graphiques, intuitives, qui permettent de contrôler cette programmation, et sur les représentations, internes et externes, des structures musicales, qui seront construites et transformées à l'aide de cette programmation.

Les logiciels *PatchWork,* et son actuel successeur *OpenMusic*, développés par l'équipe Représentations Musicales (Gérard Assayag, Carlos Agon), s'inscrivent dans ce travail de formalisation informatique des structures musicales. **OpenMusic** *[3]* est un environnement de programmation visuelle pour la CAO. Ecrit en langage **Lisp/CLOS** (*Common Lisp Object System,* langage orienté objet basé sur *Common Lisp*), et étroitement lié, dans ses principes fondamentaux, à ce dernier langage, il permet de définir de manière purement graphique des algorithmes de construction ou de traitement d'une structure musicale et d'en représenter (ou d'en manipuler) le résultat à l'aide d'éditeurs graphiques interactifs, notamment en notation musicale traditionnelle.

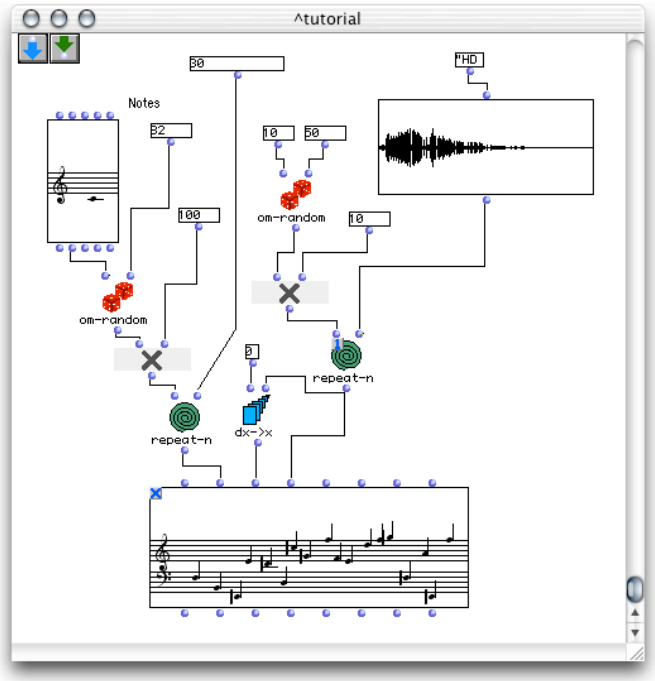


Fig. 1.1 : Exemple de patch OpenMusic

La *Figure 1.1*, représente un *patch* OpenMusic. Le *patch* est la notion de base dans OpenMusic. Il réifie le concept d'algorithme. Les patchs contiennent des *boîtes* (icônes) liées entre elles par des *connections*. Les boîtes constituent des appels fonctionnels, tandis que les connections représentent des compositions fonctionnelles.

Les boîtes sont faites d'une icône et d'un ensemble ordonné d'entrées et sorties. Ils représentent des objets (dans l'exemple de la *figure 1.1* : notes, accords, nombres, listes, sons,□.) Des boîtes de contrôle tels que boucles ou branchements conditionnels sont aussi fournis par le langage visuel.

Les boîtes et les connections dans un *patch* peuvent être vues comme un graphe de compositions fonctionnelles. L'utilisateur peut déclencher une évaluation à n'importe quel point du graphe en cliquant sur une sortie de boîte. L'évaluation d'une boîte nécessite l'évaluation des boîtes connectées à ses entrées, ce qui engendre une chaîne d'évaluations correspondant à l'exécution d'un programme. Le mécanisme d'abstraction d'un patch se fait en ajoutant des boîtes d'entrée et de sortie, représentées par des icônes de flèches (en haut à gauche sur la *figure 1.1*). Un *patch* peut ainsi être inclus dans d'autres *patches*, ou dans une structure qui permettra un ordonnancement séquentiel des différents *patches* (appelée *maquette*).

Les principaux objets de l'environnement illustré sur la *figure 1.1* sont accompagnés d'un éditeur qui permet à l'utilisateur de visualiser et de modifier ses caractéristiques de manière plus naturelle que par la saisie et la lecture de données chiffrées au niveau des entrées-sorties des boîtes (même s'il est toujours possible de procéder de cette façon).

Ainsi, un objet *Chord* (accord) dispose d'un éditeur utilisant les notations "traditionnelles" (*figure 1.2*).

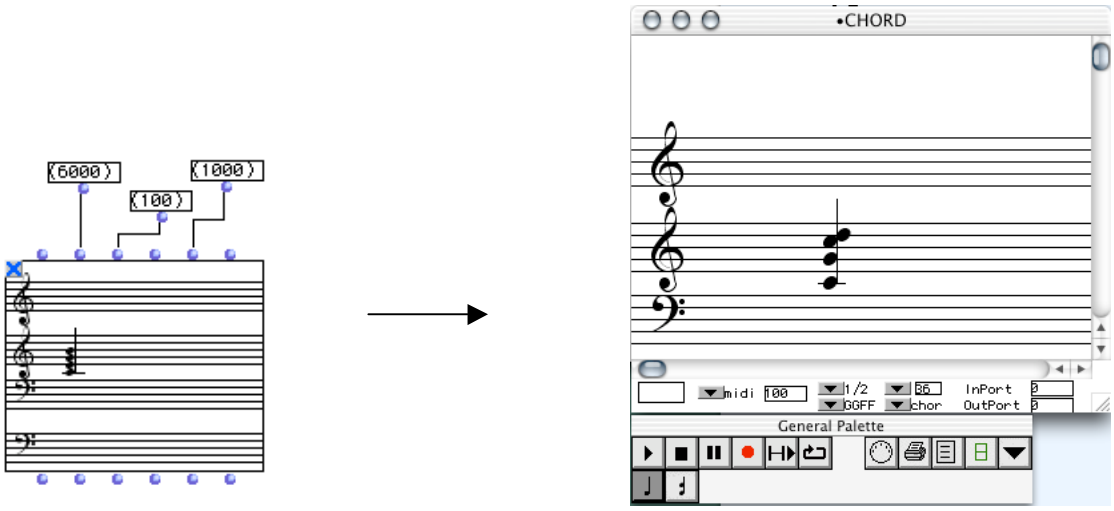


Fig. 1.2 - Un accord ("Chord") et son éditeur graphique

… et de la même manière on retrouve d'autres types d'éditeurs, comme, par exemple (*figure 1.3*), pour les objets *Sound* (son):

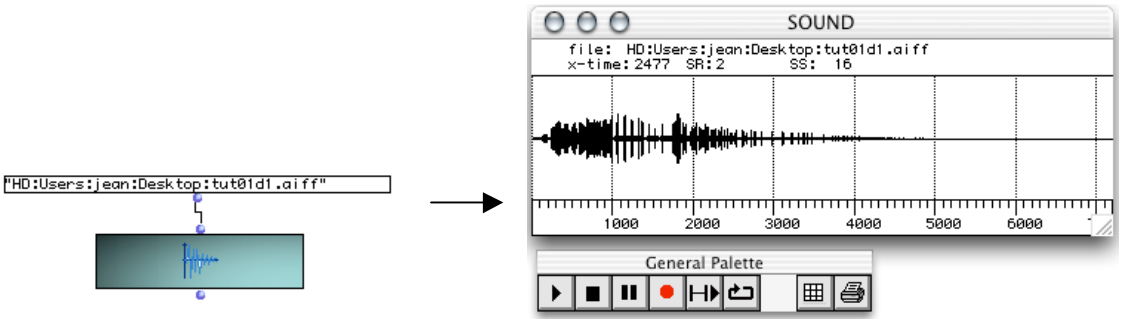


Fig. 1.3 – Un son ("Sound") et son éditeur graphique

2. Introduction au projet

2.1. Cadre du projet : OMChroma

OMChroma *[4]* est un projet basé sur le système *Chroma* développé par le compositeur Marco Stroppa, portant sur l'aide à la composition et le contrôle de la synthèse sonore. La CAO peut en effet être complémentaire à la synthèse des sons dans le sens où les sons créés sont le résultat de processus logiques dépendant de paramètres de synthèse. Le projet Chroma vise donc à effacer la frontière entre le traitement symbolique des structures musicales en CAO et le traitement du signal sonore, faisant de celui-ci un objet musical à part entière et exploitable dans l'environnement de composition.

L'objectif du projet OMChroma est donc d'intégrer le système Chroma dans OpenMusic sous forme d'une librairie, afin d'étendre le champ des possibilités compositionnelles offertes par le logiciel.

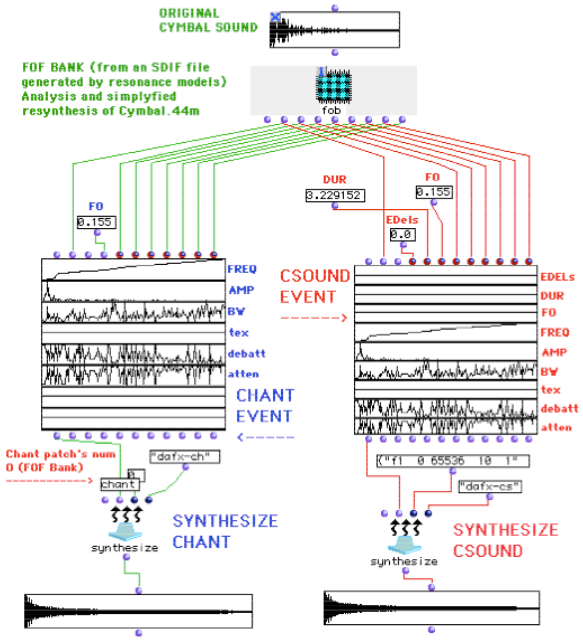


Fig. 2.1. - Patch OMChroma

La *figure 2.1* illustre cette conception de composition. On y utilise le résultat de l'analyse d'un son (en haut) afin de paramétrer la synthèse de nouveaux sons.

Dans l'environnement OpenMusic, on dispose d'une bibliothèque d'objets musicaux (accords, polyphonies, notes, etc…) qui sont des classes, possédant chacune leurs attributs (*slots*) et méthodes propres. Il est dès lors possible, pour chaque utilisateur, de définir des nouvelles classes d'objets en décrivant leurs caractéristiques. A ce titre, OMChroma introduit de nouveaux types d'objets musicaux, liés aux notions d'analyse et de synthèse sonores.

En particulier, OpenMusic permet de manipuler les fichiers de format *SDIF*, dont nous parlerons plus en détail par la suite. SDIF (Sound Description Interchange Format) *[5]* est un format de stockage générique pour les données de description sonores, et donc particulièrement adapté pour l'utilisation dans OMChroma.

2.2. Les objectifs du projet

L'objectif de mon stage sera donc d'intégrer au mieux l'objet *SdifFile* évoqué précédemment (représentant un fichier SDIF) dans l'environnement OpenMusic, principalement avec la réalisation d'un éditeur interactif qui permettra de visualiser et éditer les données contenues dans un fichier SDIF.

Il s'agit ainsi de donner au compositeur, par le biais de ce format d'échange, une vue et un contrôle sur les données de description sonores qu'il possède.

Ces données pourront provenir de programmes d'analyse ou de traitement du son, développés à l'Ircam ou ailleurs, et qui produisent en sortie des fichiers SDIF. C'est le cas par exemple de *Diphone [6]* qui produit au format SDIF les résultats d'analyse par modèles de résonance (*ResAn*), ou additive (*AddAn*). Il pourra donc s'agir d'enveloppes spectrales, de *FOF* (formes d'ondes formantiques), ou de n'importe quel type de données (c'est là un des intérêts du format SDIF), pourvu qu'elles soient ordonnées selon les spécifications SDIF.

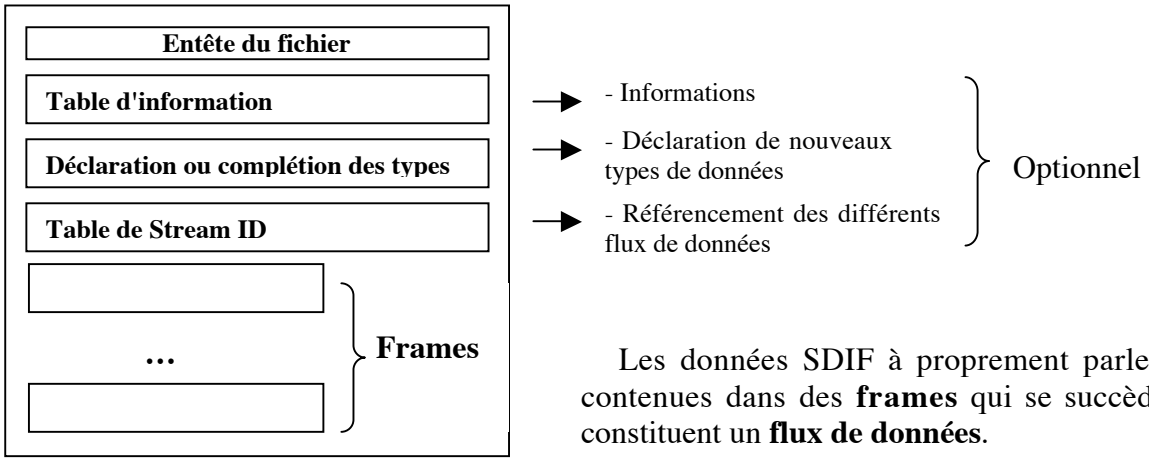
Un des aspects importants de ce travail sera donc de pouvoir traiter toutes les données SDIF, quelles qu'elles soient, sans pré considération sur leur nature. Un autre sera de penser à des systèmes de "navigation", intuitifs et efficaces, dans le flux de données, qui peut être très important et hétérogène, contenu dans un même fichier.

Enfin, l'éditeur qui sera réalisé constituera une application, ou une librairie partagée, indépendante de OpenMusic. Ainsi, elle pourra être réalisée dans un autre langage, différent de Lisp, pourvu qu'il soit possible d'y accéder au travers ce dernier. Cette dernière considération offrira une large étendue de possibilités et de choix de développement, mais introduira également un travail d'interfaçage avec OpenMusic, qui est un des autres intérêts du projet puisqu'il sera possible de l'appliquer par la suite dans d'autres applications.

2.3. Le format SDIF

Le format SDIF (Sound Description Interchange Format) a été développé conjointement par l'IRCAM et le CNMAT (Berkeley, US) afin de disposer d'un standard ouvert et multi plateformes permettant l'échange et le transfert de données de descriptions sonores. J'en ferai, dans cette partie, une rapide description, qui pourra être utile pour la compréhension ultérieure du travail réalisé.

La *figure 2.2* donne la structure générale d'un fichier SDIF :



Les données SDIF à proprement parler sont contenues dans des **frames** qui se succèdent et constituent un **flux de données**.

Fig. 2.2 – Structure d'un fichier SDIF

Chaque *frame* est constituée d'une entête dans laquelle est indiqué son **type**. Le type d'une frame permet de déterminer la nature des données contenues dans celle-ci. Cela permet aux applications utilisant des données SDIF de ne considérer que les frames qui les intéressent ou dont elles connaissent le type. Il existe des types prédéfinis par SDIF, correspondant aux principales descriptions sonores existantes, mais un des objectifs de SDIF est sa flexibilité, et il est ainsi possible pour chacun d'y ajouter ses propres types de données. Pour utiliser une frame d'un type donné, il faut que celui-ci soit défini soit parmi les types prédéfinis par SDIF, soit dans la table de déclaration de types du fichier lui-même.

L'entête d'une *frame* contient également un **temps**, et qui permet de gérer la dimension et l'ordonnancement temporels des données, ainsi que d'un identifiant de **flux** (*streamID*). Ceci permet entre autres de considérer le fichier comme un flux de données, pour des applications en transmission, ou en temps réel (toutefois, dans le cadre de la CAO, qui est le nôtre, on travaillera uniquement sur des fichiers, de taille finie). Du même coup, il est possible de stocker dans un même fichier plusieurs flux de données entrelacés, puisque chacun est identifiable et que chaque frame possède l'information sur son positionnement dans le temps.

Voyons à présent les données contenues dans une *frame* (qui constitue donc un échantillon temporel d'un des flux, ou du flux contenu(s) dans le fichier SDIF):

A son tour, une *frame* contient une ou plusieurs **Matrices**, qui sont la spécification de stockage de base du format SDIF. Chaque matrice possède un type qui doit aussi être prédéfini. Elle contient un certain type de données, organisées en lignes et en colonnes. On dira qu'une matrice représente une *structure*, dont les colonnes sont les **champs (*fields)***, et les lignes sont les **éléments**.

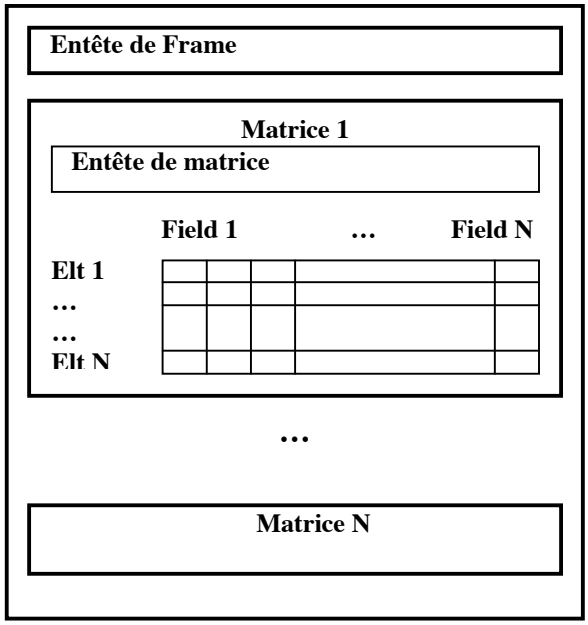


Fig. 2.3 - Schéma d'une frame dans la structuration d'un fichier SDIF

Si on prend pour exemple une enveloppe spectrale, on pourrait avoir parmi les champs : index, fréquence, amplitude; et les différentes lignes constitueraient les éléments de cette enveloppe, ayant chacun un index, une fréquence, et une amplitude:

Index	Fréquence	Amplitude
1	440	0.5
2	480	0.8
…		
n	10000	0.6

L'addition de cette frame avec les autres frames du même flux (aux autres instants échantillonnés) nous permettrait alors de reconstituer l'évolution de cette enveloppe spectrale dans le temps.

C'est donc au travers de ce type d'informations (*figure 2.4*) qu'il nous faudra nous organiser afin de proposer un outil de visualisation cohérent, intuitif et raisonnablement complexe.

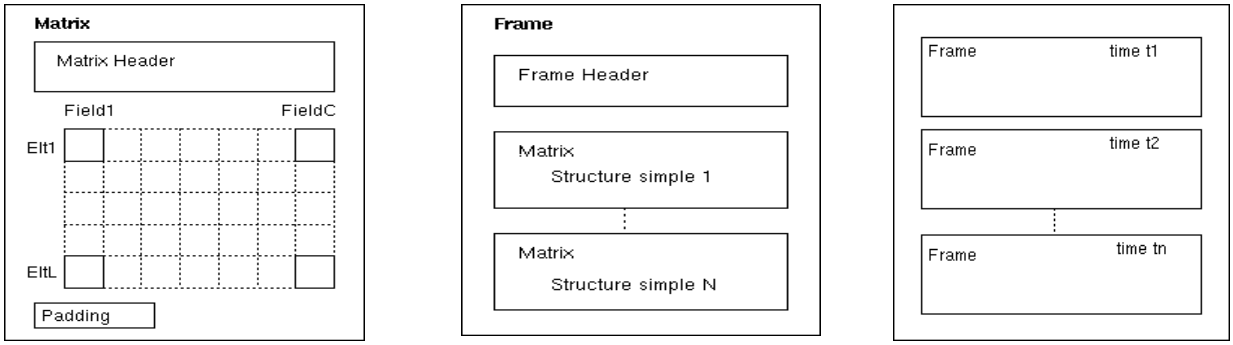


Fig 2.4 – Resumé de la structure SDIF

3. Déroulement du stage

3.1. Premières étapes

Le développement à l'IRCAM, en ce qui concerne la CAO, se fait principalement sur Macintosh (bien que, aujourd'hui, un projet de portage de OpenMusic sur Linux est en cours de réalisation). La période due mon stage a coïncidé avec une période de transition de MacOS9 vers MacOSX, qui apporte des changements radicaux par rapport aux versions précédentes de MacOS, et nécessite des mises à jours dans de nombreux domaines. Mon travail allait donc s'effectuer principalement sur cette plateforme. Toutefois, la portabilité du code a toujours été considérée dans les différents choix qui se sont posés, en particulier la portabilité Linux, puisque sur cette plateforme sont réalisés de nombreux travaux dans le cadre de l'analyse/synthèse, principal domaine d'utilisation du format SDIF.

SDIF se présente sous la forme d'une librairie, écrite en C. Celle-ci devait être la base du développement de mon application. Or n'étant pas encore portée sous MacOSX, j'ai commencé par compiler celle-ci sous forme d'un *framework* (package utilisé dans la nouvelle version de l'OS pour la gestion des librairies), ce qui m'a permis dans un premier temps de me familiariser avec l'environnement de programmation sur MacOSX, puis avec la librairie SDIF elle-même. Cette version de SDIF pour MacOSX est désormais disponible aux côtés des autres versions de la librairie *[7]*.

Les premières spécifications sur le projet et la nature des données qui devaient être manipulées ont rapidement mis en évidence l'adéquation d'une représentation en 3D pour les données d'analyse. En réalité, si l'on considère la structure décrite précédemment, il faudrait même parfois disposer de dimensions supplémentaires. C'est ce qui sera l'un des intérêts de la conception du visualisateur, qui devra, compte tenu des possibilités des dispositifs disponibles, offrir une représentation la plus complète possible.

Nous allions donc avoir besoin d'utiliser une librairie graphique 3D. OpenGL s'est naturellement présenté comme la meilleure solution, installée de série sur MacOSX et disponible sur toutes les plateformes. Cela nous permettra d'utiliser la librairie *GLUT [10]* pour la gestion des fenêtres, cette librairie étant également parfaitement portable et largement diffusée. Le développement en C++ semblait dès lors être la solution la plus adaptée.

3.2. Développement

Ce stage s'est déroulé sur une durée totale de 5 mois, au cours desquels on peut distinguer plusieurs périodes et étapes importantes.

Les premières semaines ont donc été consacrées à l'étude du projet, du logiciel OpenMusic, de la librairie SDIF, de l'environnement MacOSX, afin de déterminer les différentes possibilités qui se présentaient pour mettre en œuvre celui-ci, et de déterminer les lignes de base du développement.

Une première réunion avec des futurs utilisateurs potentiels de l'éditeur SDIF que j'allais réaliser (techniciens, informaticiens, compositeurs) a permis alors de fixer certaines caractéristiques souhaitées pour un tel éditeur et de déterminer les premières priorités. Dès lors, le travail sur la première phase, correspondant à la partie 4 de ce rapport, a commencé, avec dans un premier temps la prise en main et l'utilisation de la librairie SDIF pour la lecture des fichiers, et les premières étapes du développement 3D avec OpenGL.

Sur la fin du deuxième mois, le visualisateur était déjà relativement avancé au niveau de la représentation des données, de la navigation, des lectures-écritures dans les fichiers SDIF. Des tests avec des fichiers de diverses natures et origines ont mis à jour diverses modifications à apporter.

Une fois ce premier prototype stabilisé, la deuxième partie du développement (partie 5 du rapport), a été planifiée en termes d'objectifs, avec la création de l'interface, la gestion des fenêtres, et le développement de fonctions avancées diverses (édition, audio, vues spéciales, etc.)

L'intégration avec OpenMusic s'est faite en parallèle à partir du moment où l'application disposait d'une interface utilisable et d'une stabilité suffisante.

Diverses rencontres, avec Marco Stroppa en particulier, et avec d'autres interlocuteurs, pour les essais du logiciel, ont mis en évidence des cas particuliers à traiter et des fonctionnalités souhaitables (le cas des partiels, éditions en 2D sur les courbes, et autres cas que nous évoquerons au cours du rapport).

Sur les dernières semaines, il s'est agi plus principalement de stabiliser l'application et de rédiger une documentation utilisateur, afin de pouvoir insérer *SDIF-Edit* dans les outils Ircam dans les délais fixés. Parallèlement s'est effectuée la rédaction du présent rapport.

Le tableau donné en *Annexe 1* est un récapitulatif des différentes étapes du stage que je viens de mentionner.

4. Un visualisateur SDIF en 3D

La première étape du développement constitue en quelque sorte la base de l'application. Celle-ci devra permettre la lecture d'un fichier SDIF et la représentation graphique des données, sans prendre en compte pour l'instant, ou dans une moindre mesure, l'interface graphique et les fonctions plus avancées. Dans un souci de clarté, on s'attachera plus ici aux fonctionnalités et idées directrices de l'application qu'à leur implémentation à proprement parler.

4.1. Lecture d'un fichier SDIF

Nous avons vu qu'un même fichier SDIF pouvait contenir une quantité importante de données de diverses natures. Il est donc préférable, dans la mesure du possible, de cibler celles que l'on souhaite visualiser, afin de n'extraire du fichier que celles-ci et éviter une occupation de mémoire trop importante. Il faut donc déterminer une unité minimale de représentation que l'on chargera en mémoire, et dans laquelle on pourra effectuer rapidement diverses opérations sans avoir à accéder au fichier. On considérera la *Matrice* comme cette unité de représentation. En effet, une matrice (ou plutôt un flux de matrices, puisqu'on considère les valeurs d'une matrice d'un certain type au cours du temps), contient un certain nombre d'éléments possédant tous les mêmes champs, éventuellement liés les uns aux autres, et des données du même type, constituant ainsi un ensemble de représentation pertinent.

La librairie SDIF offre de nombreuses fonctions permettant la lecture d'un fichier SDIF (recherches de frames, ou de matrices de types donnés, etc…). Elle permet de réaliser ce type de sélection en lisant séquentiellement le fichier et choisissant, pour chaque *Frame* ou *Matrice* rencontrée, de considérer ou d'ignorer son contenu.

L'idée est donc, au lieu de lire et de charger tout le fichier en mémoire, de le parcourir en relevant les *Frames* et *Matrices* rencontrées, afin d'établir une "carte" du fichier, détaillant les différents flux (*Streams*) qu'il contient, et les matrices contenues dans ces flux. Il sera alors donné à l'utilisateur la possibilité de choisir une des matrices, afin que celle-ci soit chargée en mémoire pour être visualisée.

On va donc créer deux classes *Stream* et *Matrix*, qui seront les représentants respectivement des différents flux et des matrices contenues dans ceux-ci. La classe *Matrix* ne représente donc pas une matrice elle-même, mais la description d'un flux de matrices. Elle ne contient pas les données des matrices, mais des informations telles que le nombre d'éléments, les différents champs, les différentes apparitions de cette matrices dans un flux donné (on rappelle qu'un flux est une succession de Frames possédant le même identifiant *StreamID* et de même type.) Une classe *SdifDescriptor* manipulera les deux précédentes classes afin de mettre en place notre structure. Elle sera par la suite l'interface entre l'application et le fichier SDIF.

On utilise une structure de type *vecteur* pour stocker les objets. Le *SdifDescriptor* contient donc un vecteur de *Stream* qu'il mettra à jour au fur et à mesure qu'il lira le fichier SDIF. De même chaque *Stream* de ce vecteur contient un vecteur de *Matrix*.

La *figure 4.1* présente un diagramme de classes impliquées dans la description préliminaire du fichier SDIF, avec leurs principaux attributs :

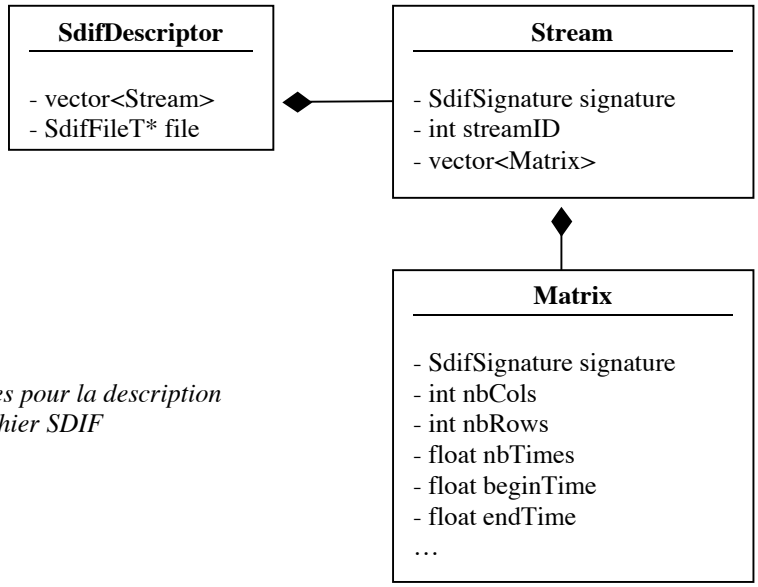


Fig. 4.1 – Classes pour la description d'un fichier SDIF

Au fur et à mesure qu'il rencontre des frames ou des matrices lors de la lecture du fichier, le *SdifDescriptor* crée des nouveaux flux (*Stream*), des matrices, et met à jour leurs attributs s'il vient à les rencontrer par la suite. Ainsi, dans un même flux, à chaque fois que l'on rencontre une matrice, on met à jour le nombre d'occurrences de celle-ci dans le flux, ainsi que les différents instants où elle apparaît.

Dans des cas "classiques", on aurait pu simplement garder les informations de temps au niveau des Frames (classe *Stream*), et garder quelques données simplement au niveau de la matrice (nom, dimensions, etc…), mais des cas particuliers ont mis en évidence la nécessité d'introduire dans la classe *Matrix* certaines informations supplémentaires : une matrice, par exemple peut ne pas être représentée dans toutes les Frames où on l'attend : il faut donc que soit tenu un compte indépendant pour chaque matrice du flux.. On pourra aussi rencontrer le cas où les matrices n'ont pas le même nombre de lignes (éléments) d'une frame sur l'autre, ce qui empêche de stocker simplement une dimension fixe pour la matrice, mais impose de garder la dimension de chaque occurrence de celle-ci dans les différentes *frames*, ce qui compliquera aussi, on le verra par la suite, l'affichage d'un tel type de matrices. Ce type de cas particulier, découlant du caractère fortement flexible du format SDIF, a entraîné quelques restructurations dans la construction de l'application, qui finalement l'ont rendue elle aussi très générale, le but étant d'être en mesure d'accepter tout fichier SDIF normalement constitué (on verra qu'il y a tout de même quelques limitations).

Voici à titre d'exemple les informations qui pourraient résulter de notre lecture "préliminaire" d'un fichier SDIF:

```
Description du fichier "monFichier.sdif":
2 Streams
Stream 1 : 1STR
    1 Matrice:
    Matrice 1 : 1TRC - SinusoidalTrack
           5 champs
           20 frames de 0.0s à 1.8s

Stream 2 : 1PRT
    2 Matrices:
    Matrice 1 : IPAR - Info
           4 champs
           200 frames de 0.0s à 2.0s
    Matrice 2 : 1PAR - Parity
           2 champs
           100 frames de 0.0s à 2.0s
```

On a donc isolé dans cet exemple, parmi les deux flux contenus dans le fichier, trois matrices. On remarque que celles-ci sont caractérisées, tout comme les frames, par une signature (code de 4 caractères) et d'un nom.

On est alors en mesure de proposer à l'utilisateur, à l'aide d'un menu, de sélectionner une de ces matrices. L'étape suivante consiste à charger en mémoire la matrice sélectionnée et la représenter avec le visualisateur.

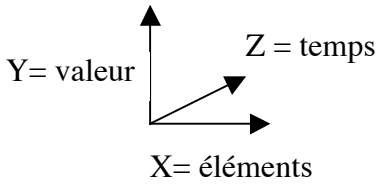
4.2. Représentation des données

Ayant déterminé la matrice que l'on va visualiser, on doit alors relire le fichier, connaissant cette fois la signature de la matrice qui nous intéresse, et lisant les données contenues par celle-ci pour les stocker dans un tableau de données.

Structure des données

La structuration des données sera déterminante dans le fonctionnement de l'application. Elle devra prendre en compte les caractéristiques des données SDIF que l'on a entrevues jusqu'ici, et permettre un accès efficace aux valeurs.

Le représentation graphique se fera sous forme d'une grille tridimensionnelle. Dans une direction (*Z*), on fixera le temps. On placera en abscisse (*X*) les éléments de matrices (dans un premier temps). Enfin en ordonnée (*Y*), on représentera les valeurs pour l'un des champs de la matrice en fonction du temps et des éléments (lignes) de cette matrice.



Un seul champ de la matrice sera donc représenté à la fois. En effet, les différents champs d'une matrices ne se prêtent pas toujours à une vue parallèle, dans le même espace de représentation. Leurs valeurs et ordres de grandeur peuvent être incompatibles, et une représentation simultanée de ceux-ci risque fort de n'avoir que peu d'intérêt dans bien des cas. Il faudra toutefois pouvoir passer d'un champ à l'autre facilement. En réalité, nous verrons par la suite que nous avons quand même la possibilité de visualiser deux des différents champs de la matrice (en en observant un en fonction de l'autre), mais dans le cas le plus simple (et celui où l'on n'aurait qu'un seul champ dans la matrice), on devra simplement représenter (en Y) un champ le long des éléments successifs (lignes) de la matrice.

Voici donc comment nous allons garder les données d'une matrice. Il s'agira d'un tableau de taille T, T étant le nombre de Frames contenant notre matrice. Chaque élément de ce tableau contiendra, en plus des données de la matrice deux informations qui nous permettront de représenter les données : le temps (en secondes) et la dimensions N de la matrice (nombre de lignes). On a vu en effet que dans certains cas particuliers cette dimension pouvait varier d'une frame à l'autre. C'est le cas par exemple si à un instant donnée ou sur un intervalle donné, une analyse plus fine est réalisée sur le son (nous verrons des exemples par la suite). Par contre, on admet que tout le long du flux, les champs de la matrices restent les mêmes (exemple : fréquence, amplitude, etc…). Les données se présentent donc sous la forme d'un tableau bidimensionnel de taille N (nombre d'éléments donné précédemment) x F (nombre de champs de la matrices).

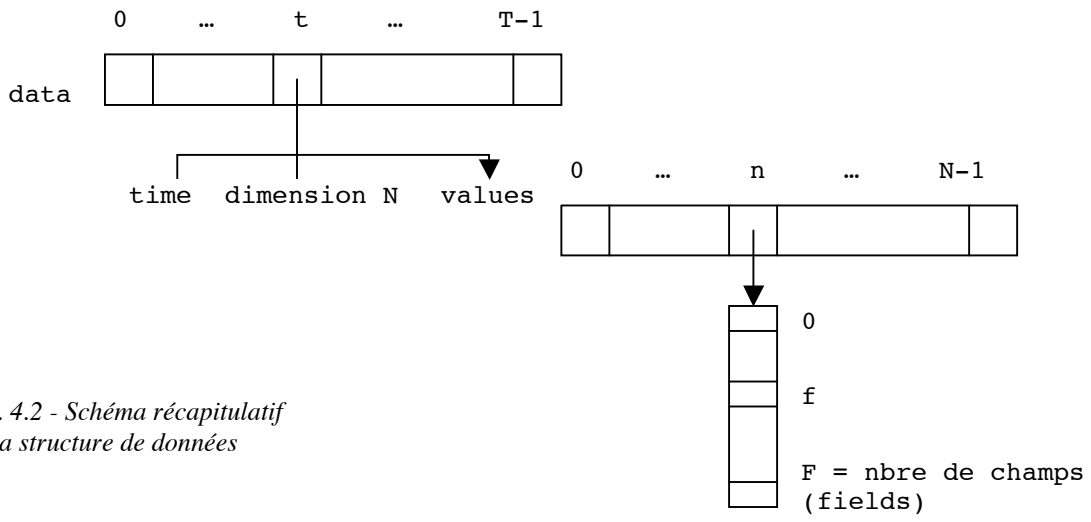


Fig. 4.2 - Schéma récapitulatif de la structure de données

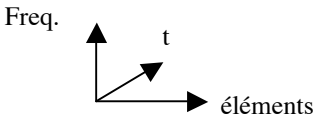
Représentation graphique

On représente donc les données selon trois dimensions. On considère les valeurs des temps en secondes afin de représenter correctement les différences qu'il peut y avoir dans la durée séparant deux frames dans le temps (la fréquence d'échantillonnage n'est pas nécessairement constante), les numéros des éléments (lignes de la matrice), et les valeurs pour un des champs (colonne de la matrice) pour disposer les points dans l'espace.

On effectuera des mises à l'échelle dans chacune des dimensions afin d'offrir une représentation correcte quelle que soit l'échelle de temps (minute, millisecondes, etc…) et la durée, et quelles que soient l'ordre de grandeur des valeurs, et les dimensions des données. Pour cela, on calcule des coefficients d'échelle, qui pourront être modifiés selon la précision désirée ou les besoins de l'utilisateur.

Mettons les choses au clair en observant quelques résultats à ce stade d'avancement. Reprenons l'exemple de l'enveloppe spectrale :

Sur la *figure 4.3*, nous avons une représentation du champ *Fréquence* de la matrice, avec les différents éléments de la matrice horizontalement. A chaque élément de la matrice, et à chaque instant correspond une fréquence.



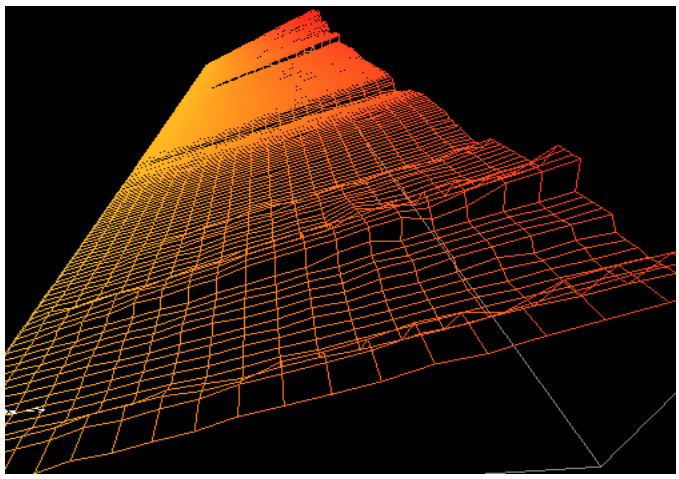
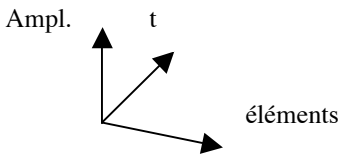


Fig. 4.3 - Représentation des fréquences d'une enveloppe spectrale

On va à présent choisir de visualiser un autre champ de cette même matrice : les amplitudes.

Cette fois on observe les amplitudes correspondant aux différents éléments de la matrice, et aux différents instants.



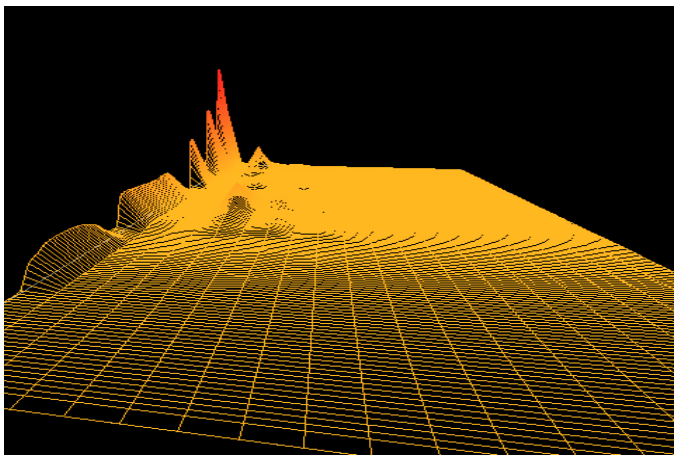


Fig. 4.4 - Représentation des amplitudes d'une enveloppe spectrale

Dans certains cas, comme celui que nous venons de voir, on apprécierait de pouvoir visualiser un champ en fonction d'un autre. Dans ce cas, au lieu d'avoir l'amplitude pour chaque élément et la fréquence pour chaque élément, on aimerait pouvoir voir l'amplitude en fonction de la fréquence, et avoir ainsi une représentation spectrale plus juste. (Ceci est valable lorsque le champ en fonction duquel on veut observer un autre est une fonction monotone, ce qui est le cas pour les fréquences, voir *figure 4.3*).

On va donc reconsidérer la dimension *X* de notre représentation 3D afin d'avoir en abscisse de la représentation non plus le numéro de ligne dans la matrice, mais la valeur d'un autre champ pour cette même ligne, ce que nous permet de faire facilement la structure de données (voir *figure 4.2*). On aura dès lors un champ sélectionné dont on visualise les valeurs, et un champ qui servira de base en abscisse. Ceci nécessitera, comme pour les autres dimensions, des mises à l'échelle en fonction des valeurs du paramètre choisi pour les abscisses, et aura pour effet de "déformer" la grille en fonction de celui-ci.

Voici (*Figure 4.5*) ce que l'on obtient si l'on choisit, avec cette nouvelle approche, de visualiser les amplitudes en fonction des fréquences dans l'exemple de la *figure 4.4*:

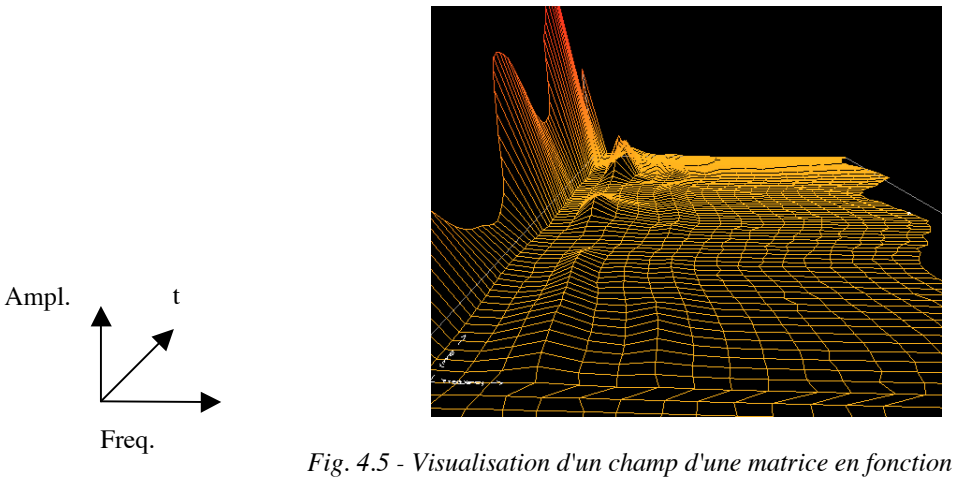


Fig. 4.5 - Visualisation d'un champ d'une matrice en fonction d'un autre

Bien sûr, si le champ choisi pour les abscisses est parfaitement linéaire, la représentation n'est pas modifiée. Inversement, et comme nous l'avons dit, si celui-ci n'est pas monotone, la représentation n'a plus de sens, c'est en partie pourquoi, nous proposerons toujours par défaut de visualiser les données en fonction des éléments de matrice, comme dans la version initiale.

On va donc modifier un tant soit peu les bases de représentations, afin de considérer ce numéro de ligne au même titre qu'un champ de la matrice qui pourra être utilisé comme base des abscisses par défaut, ou quand il n'y aura pas d'autres champ, et avoir ainsi une gestion souple et unique du paramètre choisi pour les abscisses (qu'on appellera par la suite "*x-parameter*"). Sur la *figure 4.2*, on peut voir qu'on a laissé un champ supplémentaire à cet effet dans la structure de stockage des données.

Remarquons au passage qu'une couleur est affectée aux différents points représentés en fonction de leur valeur afin d'améliorer la visibilité. Nous reviendrons par la suite sur des fonctions plus avancées liées aux couleurs (partie 5.4).

Cas spéciaux

Il arrivera souvent que les données contenues dans un fichier SDIF ne se prêtent pas parfaitement au cas "simple" évoqué ci-dessus. Nous avons déjà parlé du cas où la matrice pouvait avoir des dimensions variables selon les frames (moments). Ce cas a été considéré dans la structure des données (voir *figure 4.2*) en prévoyant une dimension à chaque instant, et, par quelques manipulations supplémentaires, il est aussi supporté lors de la visualisation, comme le montre l'exemple sur la *figure 4.6* :

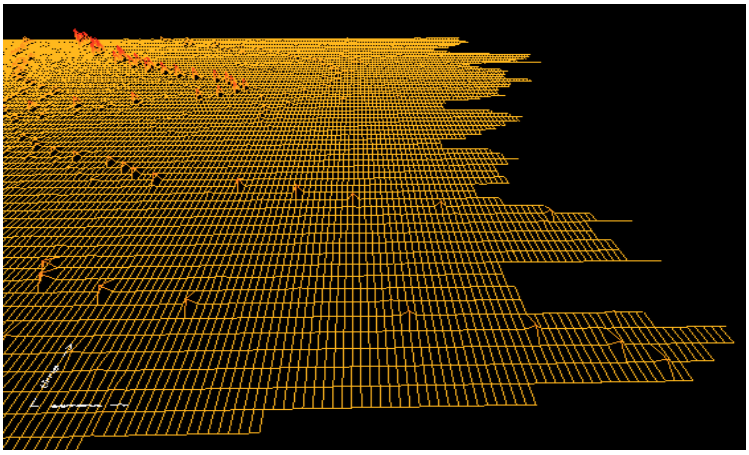


Fig. 4.6 - Une matrice aux dimensions variables…

On rencontrera également des exemples où les matrices auront un seul élément, dans quel cas on représentera une courbe d'évolution dans le temps d'un paramètre (*figure 4.7*), ou le cas symétrique où il n'y a pas d'évolution temporelle, donc un seule *frame* (*figure 4.8*) (voire même le cas où l'on trouve une frame et un élément de matrice, et où la représentation se limite par conséquent à un point unique…)

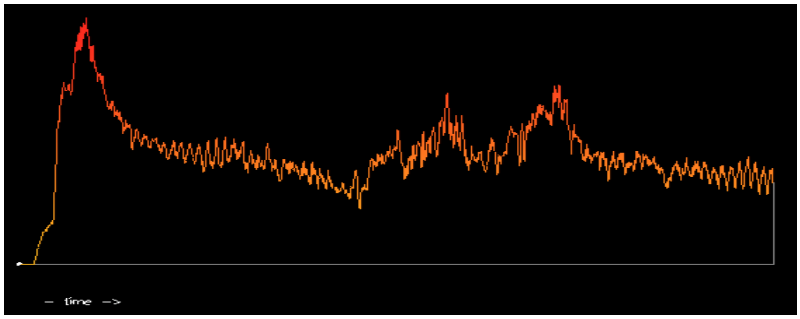


Fig. 4.7 - Evolution d'un paramètre (ici "loudness") dans le temps

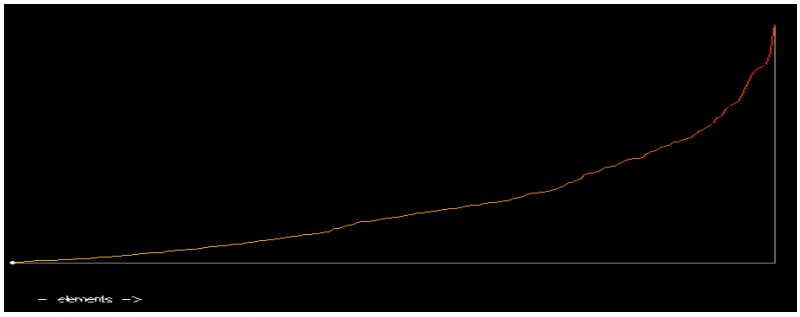


Fig. 4.8 - Champ (ici:"frequency") d'une matrice sans dimension temporelle

4.3. Navigation

Une fois obtenue une représentation graphique des données, nous devons nous préoccuper de la manière d'observer celle-ci, autrement dit du positionnement de la caméra au milieu de ces données.

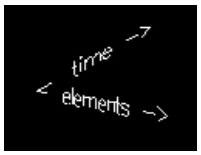
Mouvements

Afin de d'assurer un mouvement cohérent de la caméra au milieu des données 3D, les mouvements de bases sont fixes selon les trois directions du repère. Il est possible de "bouger la tête" avec la souris pour changer le direction d'observation sur les données, mais le déplacement se fera toujours parallèlement aux axes, à l'aide des touches fléchées du clavier.

La vitesse de déplacement est un paramètre qu'il est possible de régler, tout comme, on l'a dit plus haut, les échelles selon les différents axes. Le réglage de ces échelles permet à l'utilisateur d'avoir un vision, tantôt plus précise, tantôt plus globale, des données.

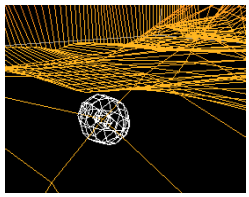
Positionnement

La vue initiale est calculée au départ en fonction de la dimension des matrices et des valeurs, ainsi que les échelles, afin d'avoir une vue générale satisfaisante dès le lancement du visualisateur. Il sera permis à n'importe quel moment de retrouver cette vue initiale (ce qui est appréciable si l'utilisateur se "perd" dans l'espace 3D).



Un petit repère, que l'on a pu observer dans le coin de certaines des vues précédentes, permet également d'avoir connaissance à tout moment de son orientation par rapport au repère initial.

Sélection d'un point



A l'aide de la souris, il est possible d'effectuer une sélection à l'écran. En utilisant des fonctions proposées par OpenGL, on réussit à déterminer, en créant une représentation locale autour d'une position sur l'écran, quel est le point de la grille de données qui a été sélectionné. On peut alors le représenter différemment des autres. On pourra ensuite déplacer ce point à l'aide du clavier.

La présence d'un point sélectionné ouvre alors de nombreuses autres possibilités; nous le retrouverons dans un grand nombre de fonctionnalités présentées dans ce rapport. Tout d'abord, on peut mettre en place un autre type de mouvement, centré sur ce point. Dans un tel mode de déplacement, on se déplace parallèlement aux axes en déplaçant le point sélectionné sur la grille de données (la camera "suit" le point), on se déplace par rotation autour de ce point si l'on va vers les côtés (en gardant celui-ci comme centre de visée), en on va en s'approchant ou s'éloignant du point avec le mouvement avant/arrière. On a ainsi deux "modes" de déplacement, entre lesquels on pourra jouer.

Vue des différents champs de la matrice

Nous avons vu que la représentation simultanée des différents champs était limitée (il est possible tout au plus d'en disposer un en fonction d'un autre). Toutefois, ayant toutes les données en mémoire, il est possible de passer rapidement de l'un à l'autre, et de modifier le paramètre que l'on a choisi de mettre en abscisse de la représentation.

Le passage d'un champ à l'autre permet d'avoir différentes visions de la même matrice selon les différents champs, avec chacun ses réglages et mises à l'échelle propre.

De plus, le fait d'avoir un point sélectionné parmi toute la grille de données va nous permettre d'afficher les valeurs numériques des différents champs pour la position de ce point, réalisant ainsi d'une certaine manière une coupe, vision comparée des valeurs de la matrice selon une nouvelle dimension (*figure 4.9*).

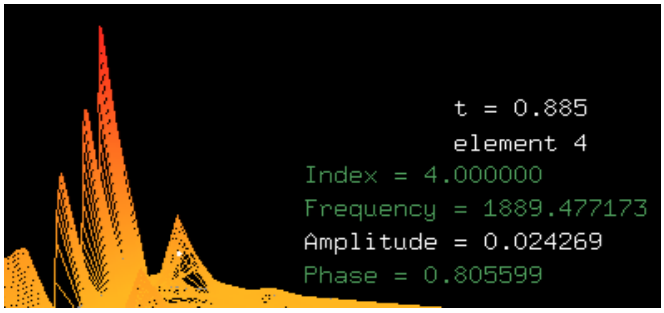


Fig 4.9 - Valeurs des différents champs au point sélectionné

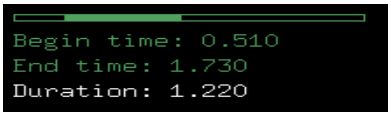
Remarque : Pour l'affichage de texte à l'écran, on utilise une projection orthogonale qui permet d'afficher formes et caractères à l'écran comme si l'on dessinait directement sur le plan de l'écran. De cette manière on peut, en allant chercher dans les structures composant le *SdifFile* de la librairie SDIF, retrouver et afficher les noms des matrices et frames, ainsi que les noms des champs constituant les matrices.

Sélection d'une fenêtre temporelle

Il arrivera parfois qu'un flux SDIF contienne un nombre important de frames (selon la fréquence d'échantillonnage, et la durée de l'extrait sonore décrit). On pourra alors souhaiter, soit pour avoir une vision plus claire, soit pour alléger le coût de l'affichage, ne sélectionner qu'une partie des données.

On introduit pour cela deux valeurs *début* et *fin* qui limiteront la zone de données qui doivent être affichées à l'écran.

Afin de gérer cette sélection de manière légère et intuitive, une petite barre d'aperçu permet d'évaluer la portion visible par rapport à l'ensemble des données.



Le déplacement de la zone sélectionnée (dans un premier temps avec le clavier puis, lorsque l'on mettra au point une interface utilisateur plus élaborée, avec la souris) permet d'observer l'évolution temporelle de cette portion de manière dynamique (voir *figure 4.10*).

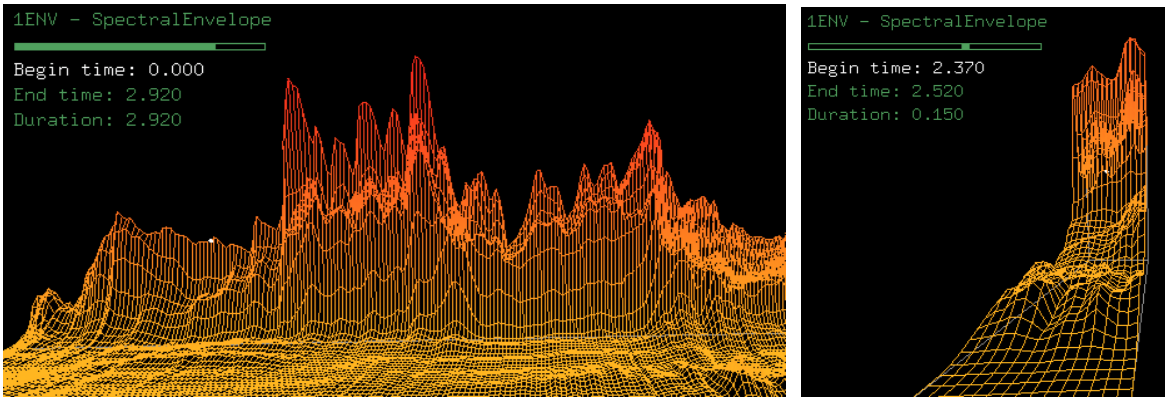


Fig 4.10 – Rétrécissement de la fenêtre temporelle de visualisation d'un fichier SDIF (2.92s à gauche – 0.15s à droite)

4.4. Edition

Le point que nous avons sélectionné précédemment va aussi constituer un premier pas dans l'édition des données. Connaissant sa position, il est facile, avec des touches du clavier, d'augmenter ou de diminuer la valeur correspondante. La difficulté réside dans l'évaluation de la quantité dont on va faire varier cette valeur. Celle-ci sera déterminée à l'aide d'un pas d'édition, lui-même calculé en fonction l'amplitude des données (différence entre les valeurs les plus basses et les valeurs les plus hautes), mais également en fonction inverse du facteur d'échelle appliqué aux valeurs. Ainsi on aura visuellement toujours le même ordre de variation, mais qui correspondra a une variation réelle plus ou moins importante. Ce pas d'édition est également modifiable afin de permettre une édition précise par raffinements successifs.

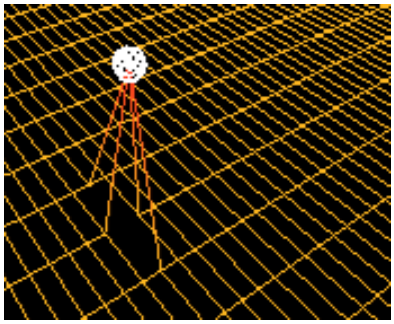


Fig 4.11 - Edition d'un point

Cette édition ponctuelle n'aura en réalité que peu d'utilité dans les cas d'utilisations, mais on mettra par la suite en place un mode d'édition plus avancé (partie 5.3).

Sauvegarde des modifications

Ayant introduit la possibilité d'édition des données, on doit considérer la manière de sauvegarder ces modifications. En effectuant l'édition sur la grille de données, comme nous venons de le voir, la structure du fichier SDIF n'est pas modifiée. Seules varient les valeurs contenues dans les matrices. On pourra donc sauvegarder l'édition en réécrivant nos données dans le fichier, simplement en connaissant les position dans celui-ci des matrices que nous venons d'éditer.

4.5. Bilan de cette première phase de développement

A l'issue de cette première phase, nous disposons d'une application permettant d'ouvrir et de visualiser des archives SDIF. Des exemples, au cours de différents essais, ont mis à jour certaines défaillances et des cas inattendus dans les fichiers à visualiser. Dans la mesure du possible, j'ai pris en compte ces cas et effectué les corrections nécessaires, afin de renforcer la flexibilité de l'application.

La navigation et la visualisation, si elles demandent un petit peu d'adaptation, sont plutôt satisfaisantes, mais à ce stade, la plupart des commandes sont accessibles à partir du clavier seulement, ce qui demande une bonne connaissance préalable des touches. Par la suite, le développement s'orientera sur la mise en place d'une interface homme machine plus élaborée.

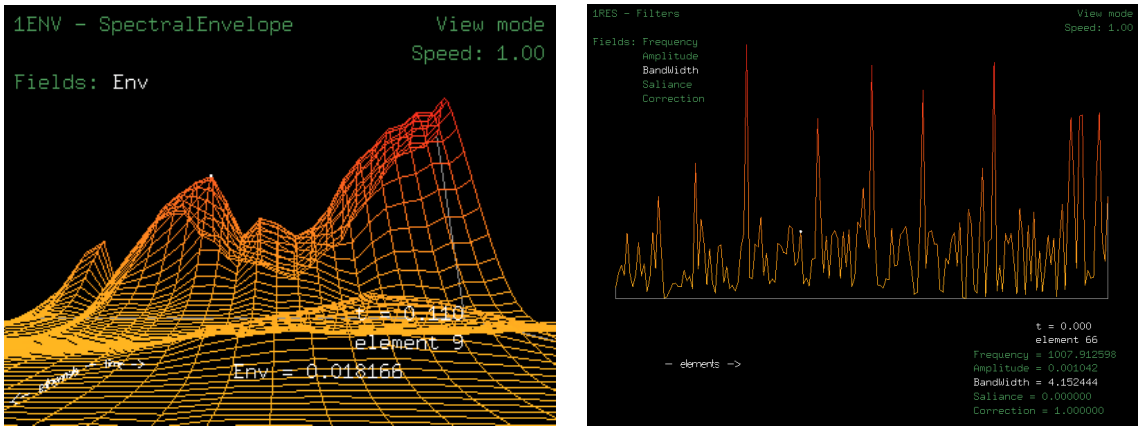
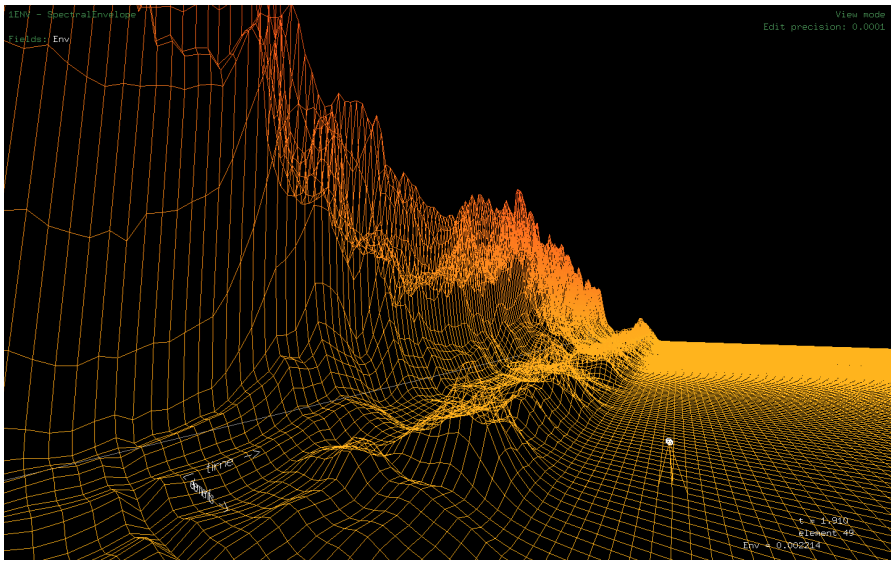


Fig 4.12 – Visualisateur SDIF

Suite à quelques essais et bilans sur ce premier travail, de nouveaux objectifs en termes de fonctionnalités du visualisateur ont également émergé. On développera également dans la partie suivante ces nouvelles possibilités, parmi lesquelles on peut citer l'édition d'une zone de données, que l'on a déjà évoquée, la gestion des préférences de l'utilisateur, de vues spéciales, l'intégration de sons dans le visualisateur.



5. Une application plus élaborée…

5.1. Interface

Au stade où nous en sommes, l'application nécessite l'utilisation d'un nombre croissant de touches pour activer ses différentes fonctionnalités. A un certain moment apparaît la nécessité de disposer d'une interface graphique qui permettra d'afficher plus d'information et d'offrir, par l'intermédiaire de boutons, un accès plus explicite aux différentes fonctions.

Toutefois, les actions de navigation (déplacement avec les flèches de direction, direction du regard avec la souris,□.) resteront ce qu'elle sont, ce qui semble être le plus naturel.

Pour réaliser l'interface graphique, nous allons utiliser le contexte graphique OpenGL dans lequel nous nous trouvons. En effet, nous avons vu que par une simple projection orthogonale, il nous était possible de "dessiner sur l'écran". J'ai donc fait le choix de réaliser moi-même l'interface graphique, qui serait certes moins complexe, tant au niveau des possibilités qu'au niveau graphique, que si j'avais utilisé un toolkit graphique, mais qui suffirait aux besoins de l'application, et présenterait l'avantage d'être simple à utiliser, et de ne nécessiter aucune ressource supplémentaire, laissant intacte la portabilité de l'application.

L'interface se présente sous la forme d'une unique classe *GUIComponent*, qui représente un composant graphique, et qui peut se décliner sous quatre formes, que l'on appellera *bouton*, *label*, *panel*, et *special*. Un *GUIComponent* possède quelques méthodes, en particulier une méthode pour le positionner et donner sa taille, pour lui affecter un texte éventuel, une méthode pour tester l'appartenance d'un point au rectangle formé par ce composant (utilisée pour l'activation, le cas échéant, d'une fonction correspondante au bouton), une méthode *draw()* pour dessiner le composant. C'est au niveau de cette dernière que se trouve la principale différence entre les différents types que nous avons énumérés. Ainsi un *bouton* sera dessiné comme un rectangle avec un texte, un *label* comme un texte uniquement, et un *panel* comme un rectangle. Le composant de type *special* est destiné à des applications particulières et permet de définir la manière dont il sera dessiné. Avec ces quelques éléments, on peut déjà constituer une interface. La composition de l'interface est relativement facile puisqu'il suffit de créer dans la fenêtre des objets *GUIComponent*, qu'on localise et auxquels on affecte les caractéristiques souhaitées. Au moment de dessiner la fenêtre, on passe en revue tous les *GUIComponents* pour les dessiner avec leur méthode *draw* respective.

La *figure 5.1* montre l'apparence du précédent visualisateur remanié pour intégrer l'interface nouvellement conçue :

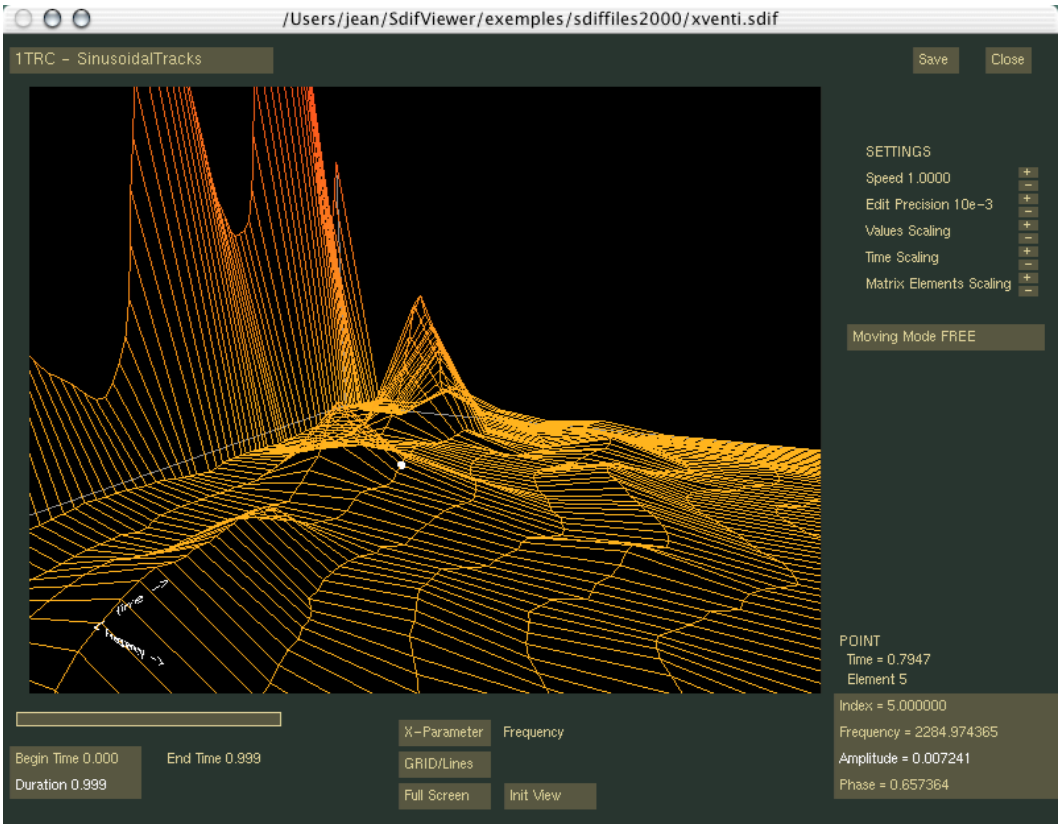


Fig 5.1 – Le visualisateur SDIF et sa nouvelle interface

L'affichage 3D occupe seulement une partie de la fenêtre, le reste étant occupé par des composants graphiques : *panels*, en vert foncé, *boutons* en vert plus clair, *labels* avec les différents textes de l'écran. En cliquant sur les boutons, on active les différentes fonctions anciennement dépendantes du clavier (pour certaines fonctions, on a aussi conservé des raccourcis clavier).

On a un exemple de *GUIComponent* de type *special* dans la petite barre d'aperçu du temps en bas à gauche de la fenêtre de la *figure 5.1*. Celle-ci, on le voit, est dessinée différemment. Son affichage est géré directement au niveau de la fenêtre. De plus, y est inséré un mécanisme de "*drag & drop*", que l'on sera amené à réutiliser par la suite. Pour cela, il faut seulement distinguer les cas "*mouse down*" et "*mouse up*" lors du clic de souris, et tenir à jour une variable permettant de savoir quel a été le dernier composant impliqué qui a été cliqué pour activer ou désactiver le "*drag&drop*". On peut connaître les déplacements effectués sur ce composant avec la souris appuyée et modifier les valeurs (ici les bornes de la fenêtre temporelle) en conséquence.

5.2. Gestion des fenêtres et de l'exécution

La librairie *GLUT*, utilisée pour l'affichage 3D avec OpenGL est aussi, et en grande partie, utilisée pour la gestion de la fenêtre et de l'exécution de l'application. Elle gère une boucle d'événements, qui permet l'actualisation de l'affichage graphique et la récupération des événements clavier, souris, etc…

Par la suite dans le développement, on sera amené, pour ne pas surcharger la fenêtre principale, à créer des petites fenêtres "annexes". Il faut donc penser à gérer ces différentes fenêtres de manière cohérente. Pour cela, nous allons créer une superclasse *Window* dont hériteront toutes les fenêtres de l'application. Il appartiendra à chaque fenêtre de gérer l'affichage, la réponse aux événements, etc…, mais des fonctions plus générales pourront être définies dans cette classe (création, visibilité des fenêtres, etc…).

Mais surtout, cette classe *Window* va permettre de gérer, de manière **statique** la boucle d'événement et l'envoi des évènements aux fenêtres et aux objets concernés. Pour cela on tiendra à jour une table référençant les fenêtres ouvertes par leur identifiant (unique). On pourra ainsi avoir plusieurs visualisateurs, et leurs sous-fenêtres ouverts en même temps, et on aura la possibilité de retrouver la fenêtre active, ou une autre, et de lui envoyer des instructions.

La principale difficulté réside dans la gestion des ouvertures et fermetures de fenêtres. On est pour cela aidé par un identifiant unique attribué par *Glut* à chaque fenêtre lors de sa création. On peut donc créer une hiérarchie dans les fenêtres, une fenêtre en ouvrant une autre, etc.. , tout en gardant le contrôle sur chacune grâce à la table que nous venons de mentionner et qui permet de "remettre la main dessus" au moment voulu.

Toutefois, *Glut* ne propose pas de fonction de fermeture propre à chaque fenêtre (quand on clique sur la "croix" dans le coin supérieur), ni de contrôle sur la boucle d'évènements (celle-ci est bloquante jusqu'à la fermeture de l'application). Ceci est pour nous un problème, et pour plusieurs raisons : tout d'abord parce que pour fermer une fenêtre, sans fermer toute l'application, il nous faut créer un autre bouton "close", et surtout pour l'intégration de l'application dans OpenMusic. En effet, sans contrôle sur la boucle d'évènements, on ne peut pas détruire ou fermer les fenêtres correctement pendant le déroulement de l'application, ce qui entraînera des complications au moment où OpenMusic demandera à notre application de maintenir ouverts plusieurs éditeurs simultanément.

Ces faiblesses de la librairie *Glut* peuvent être remédiées en utilisant des versions alternatives de celle-ci, comme celle proposée par Rob Fletcher *[11],* ajoutant des contrôles non bloquants de la boucle d'événements et une fonction de fermeture des fenêtres. C'est le cas des versions récentes de *Glut* distribuées par *Apple* pour *MacOSX [12]*. En utilisant celle-ci, on arrive à avoir une gestion de l'exécution plus satisfaisante pour notre application.

Structure de l'application

L'application (que l'on appellera *SDIF-Edit*) est lancée à partir d'une première fenêtre (voir *Figure 5.2.1*), qui permet de choisir, parmi les matrices contenues dans le fichier SDIF, celle que l'on souhaite ouvrir avec le visualisateur. Cette fenêtre et celle du visualisateur constituent en principe, au même titre l'une et l'autre, les fenêtres principales, dans le sens où en fermer une équivaut à fermer le visualisateur. Chacune devra donc garder un contrôle sur l'autre afin d'être fermées en même temps, et surtout de pouvoir passer de l'une à l'autre (passer de la fenêtre des matrices au visualisateur lors de le sélection d'une matrice, ou retour à la sélection des matrices depuis le visualisateur).

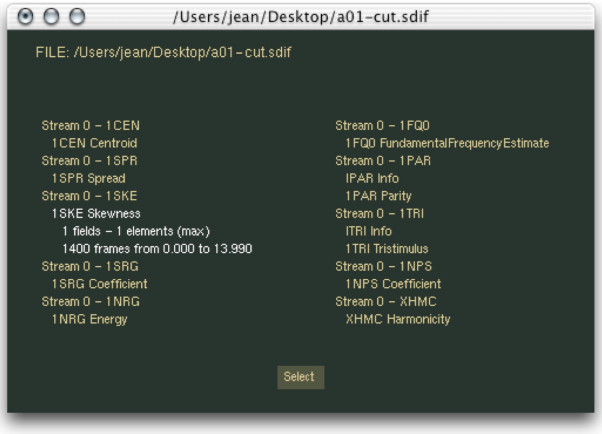


Fig 5.2.1 – Fenêtre de sélection des matrices

Hiérarchiquement, puisque c'est elle que l'on ouvre en premier, la fenêtre des matrices est la vraie fenêtre principale, qui commande l'ouverture du visualisateur avec une matrice donnée. Celui-ci, à son tour, contient et commande de nombreuses petites fenêtres, que nous verrons en détail par la suite, et qui permettront la mise en œuvres de fonctionnalités de visualisation ou d'édition des données spécifiques. La *figure 5.2.2* illustre cette structure dans les fenêtres de l'application.

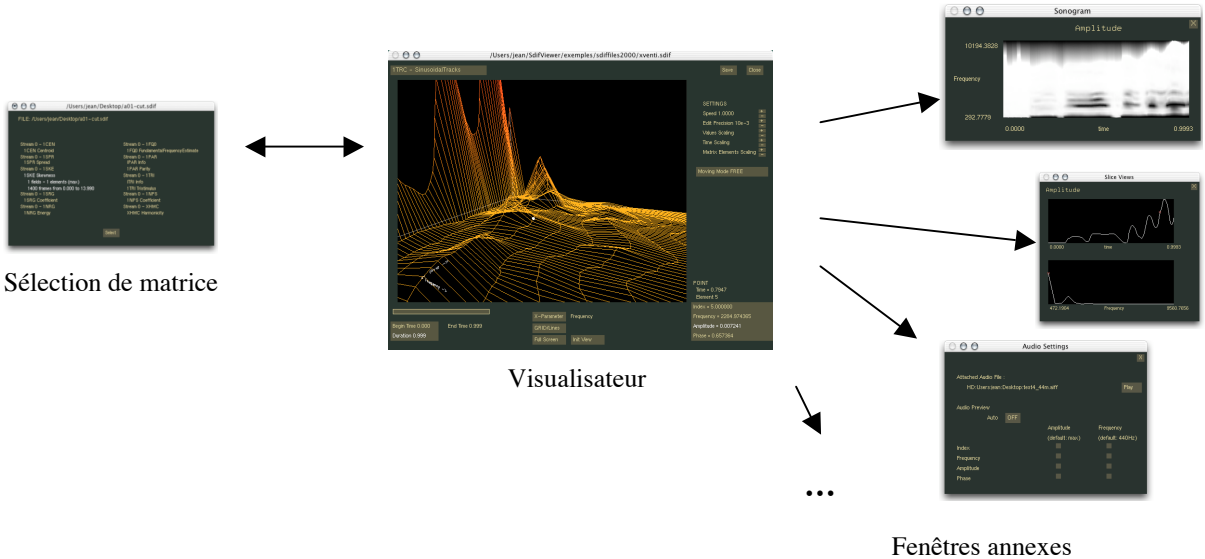


Fig 5.2.2 – Différentes fenêtres de SDIF-Edit

5.3. Edition des données

Pour permettre une édition des données réellement efficace (jusqu'ici, on ne pouvait éditer les valeurs des données que point par point), il est préférable de pouvoir éditer toute une zone à la fois, afin d'élever ou d'adoucir des "pics" sur les données que l'on a et si possible suivant une forme naturelle.

Pour cela, la première étape sera de déterminer une zone d'édition. Celle-ci se situera de part et d'autre du point sélectionné. Moyennant quelques boutons supplémentaires sur l'interface, on peut sans difficulté, en prenant garde aux dépassements de mémoire dans les accès aux données de notre tableau de stockage, permettre à l'utilisateur de déterminer cette zone (*figure 5.3.1*).

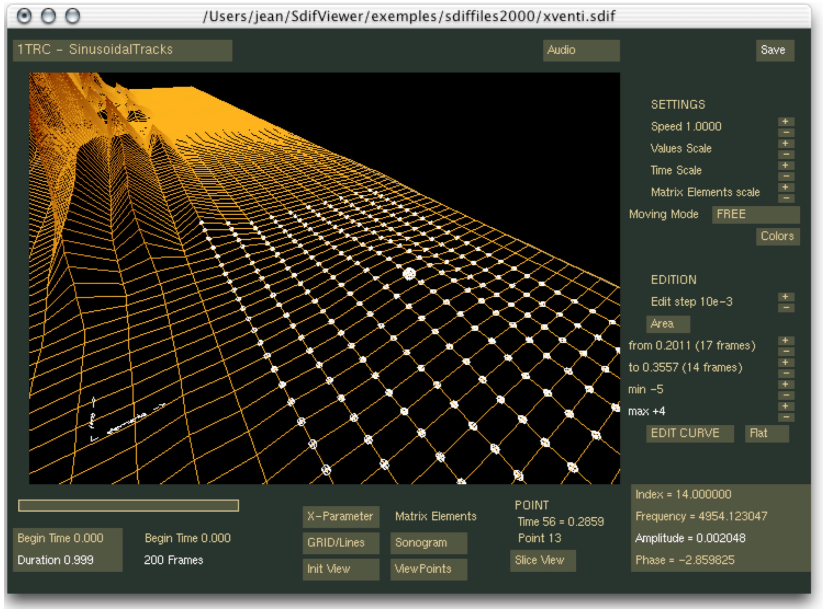


Fig 5.3.1 – Une zone d'édition autour du point sélectionné

Une fois cette zone déterminée, on pourra par exemple aplatir toute la zone, en donnant à tous ses points la valeur du point sélectionné. On a ainsi déjà une fonctionnalité qui permet d'aplatir des zones accidentées sur une grille ou une courbe.

Toutefois, l'objectif principal reste de pouvoir éditer cette zone de sorte à former une "bosse" (ou un creux) d'un profil déterminé. Le principe d'édition que j'ai mis en place consiste à répercuter sur les points qui la constituent les modifications du point central (qui reste toujours le "centre" de cette zone d'édition, même si il ne se trouve pas géométriquement au milieu). On a donc le choix entre plusieurs types d'édition de la surface, selon la manière on on répercute ces modifications, etparmi lesquels on choisira le plus approprié selon le résultat recherché.

Le premier consiste à faire suivre à toute la zone les modifications appliquées au point central. Un autre type d'édition consiste à effectuer ces modifications en chaque point proportionnellement à sa valeur par rapport au point central. Ceci aura pour effet de respecter la forme initiale de la surface éditée, permettant d'amplifier ou d'atténuer celle-ci en gardant les mêmes proportions.

Enfin, un dernier mode, plus complexe, permet d'éditer interactivement le profil de la déformation que l'on va réaliser. Pour ce faire, j'ai utilisé les propriétés des surfaces *splines*. La suite de cette partie sera consacrée à la description de cette méthode d'édition.

Une surface *spline* est une surface continue, définie à partir d'un ensemble limité de points de contrôle. Celle-ci passe par les extrémités et s'étend à l'intérieur de l'espace convexe délimité par ces points de contrôle. L'idée est de permettre à l'utilisateur d'éditer, en positionnant ces points de contrôle, le profil d'une telle surface, qui servira de modèle à la zone que l'on va modifier.

Dans cet objectif, j'ai créé un mini-éditeur de surface spline, sous forme d'une petite fenêtre qui apparaît grâce à un bouton sur l'interface du visualisateur principal. Sur cet éditeur ne sont représentées en réalité que deux courbes, qui sont les coupes transversales de la surface selon nos axes, et à partir desquelles on déduira le reste de la surface.

Observons cet éditeur (*figure 5.3.2*) afin de comprendre comment sont générées les courbes de référence.

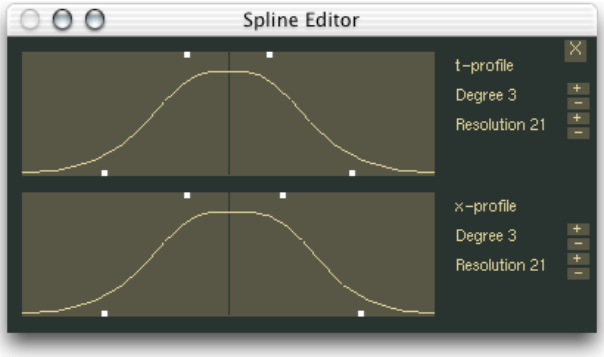


Fig 5.3.2 – Editeur de surface spline

On voit en clair sur les courbes les points qui sont les points de contrôle. Avec quatre points, on arrive à une liberté satisfaisante dans les possibilités d'édition de la courbe. En utilisant le principe de *drag & drop* comme décrit dans la partie 5.1, on offre la possibilité (en imposant quelques contraintes) de déplacer ces huit points sur les courbes. Un algorithme de spline *[13]* calcule au fur et à mesure la surface obtenue et ses deux profils que l'on voit sur l'éditeur.

Notre surface a en réalité un nombre plus important de points de contrôle, comme le montre la *figure 5.3.3*. Chaque courbe en possède 7, dont 3 fixes : deux aux extrémités de hauteurs 0, et un au centre de hauteur 1. Il nous reste donc quatre points "mobiles" par courbe, répartis par deux de part et d'autre du centre, et dont les hauteurs sont également fixes (0 pour les points proches des extrémités et 1 pour ceux qui sont proches du centre). Le positionnement des points de contrôle mobiles est donc uniquement latéral mais c'est, comme nous le verrons, suffisant, et cela limite la complexité du problème. A partir des huit positions que nous avons, on déduit celles des autres points de la surface de contrôle.

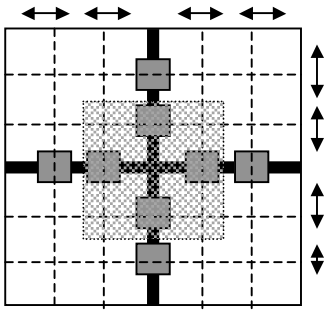


Fig 4.3.3 – Schéma des points de contrôle de la surface

La taille de la surface est fixe (1x1), ainsi que les hauteurs des points (O pour ceux de la périphérie, 1 pour ceux de la zone centrale grisée)

Les carrés gris représentent les points de contrôle gérés par l'éditeur, et les lignes épaisses les deux coupes correspondantes.

On voit que chacun des points gris permet de contrôler la position d'une ligne entière dans la grille que constitue l'ensemble des points de contrôle.

Les autres paramètres qui nous intéresseront dans la génération de la courbe sont le degré des courbes, qui rendra les courbures plus ou moins douces, et la résolution de la surface spline que l'on va générer (à partir de notre "surface de contrôle" de 7x7 points). Ces deux paramètres sont réglables indépendamment dans les deux directions.

En modifiant les positions des points sur l'éditeur, et le degré des courbes, on a déjà une variété intéressante de profils de courbes (voir *figure 5.3.4*).

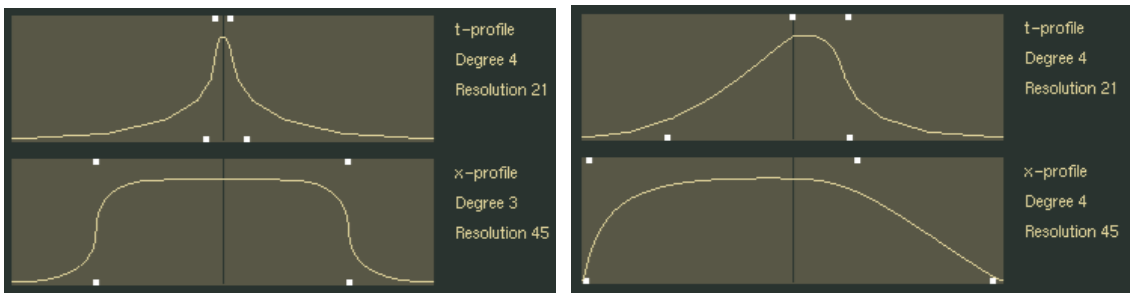


Fig 5.3.4 – Quelques exemples de profils de courbes

Pour appliquer cela à l'édition des points, on va mettre en correspondance la zone d'édition du visualisateur avec la surface spline que l'on a calculée, afin de déterminer à quel point de celle-ci correspond chaque point de la zone. La hauteur de ce point correspondra donc à la variation de valeur à appliquer à ce point pour une variation de 1 sur le point central de la sélection. En effectuant la recherche et ce calcul de rapport pour chaque point de la zone, et ce à chaque modification du point central, on obtient une formation de bosse correspondant aux profils des courbes qui ont été éditées.

La résolution de la courbe spline influera sur la finesse de la recherche, et donc la continuité de la "bosse" élevée. Idéalement, elle doit être supérieure aux dimensions de la zone d'édition, sans quoi apparaîtront des "marches" sur les courbes, mais ceci est évidemment au détriment du coût de calculs (pour la recherche) et donc de la vitesse d'exécution.

Une classe *SplineSurface* est attachée au visualisateur. C'est elle qui gère toute la partie algorithmique qui intervient lors de la manipulation des courbes, le calcul de la surface spline, et de l'édition des données.

La *figure 5.3.5* montre un exemple d'élévation d'une bosse sur une zone plate grâce à la méthode développée ci-dessus. Il est évidemment possible, de la même manière, de faire des creux, ou d'adoucir des reliefs existants.

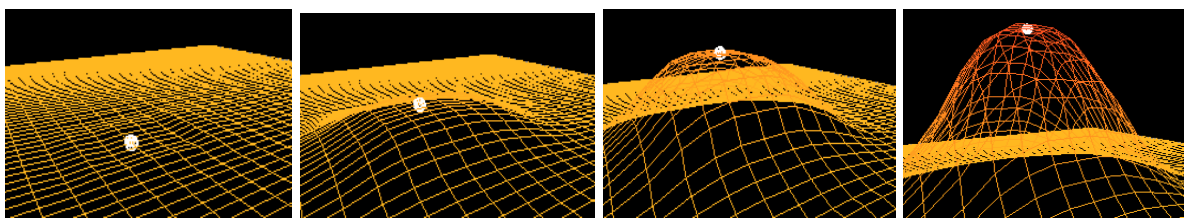


Fig 5.3.5 – Edition d'une zone de données

5.4. Préférences de visualisation

Couleurs

Nous avons vu sur les figures illustrant ce rapport que les points représentant les données sont affichées d'une couleur dépendante de leur valeur. On passe linéairement d'une couleur à une autre en fonction de ces valeurs.

Or, pour des raisons esthétiques, ou de visibilité, on peut souhaiter modifier ces couleurs. C'est pourquoi j'ai ajouté un petit utilitaire à l'application, sous forme d'une nouvelle petite fenêtre, et basé sur le principe d'un *ColorChooser* classique (*figure 5.4.1*).

Celui-ci fonctionne encore une fois avec des *drag&drop*: on choisit une des couleur de la palette pour la faire glisser sur une des cases *min value*, *max value*, ou *background*. On peut alors obtenir différents résultats visuels en combinant ces couleurs (*figure 5.4.2*, par exemple).

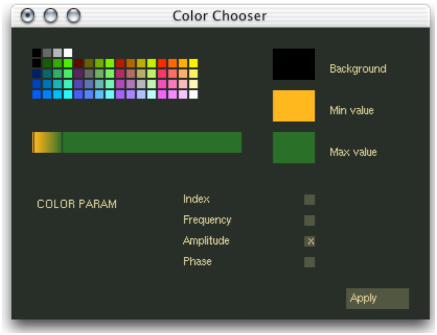


Fig 5.4.1 – Color chooser

Un autre outil intéressant au niveau de la visibilité est la barre située en bas de la fenêtre, et qui permet de modifier l'étendue de la zone de variation de couleurs, afin d'obtenir, si besoin, plus de précision sur une partie du graphique (voir *figure 5.4.2*).

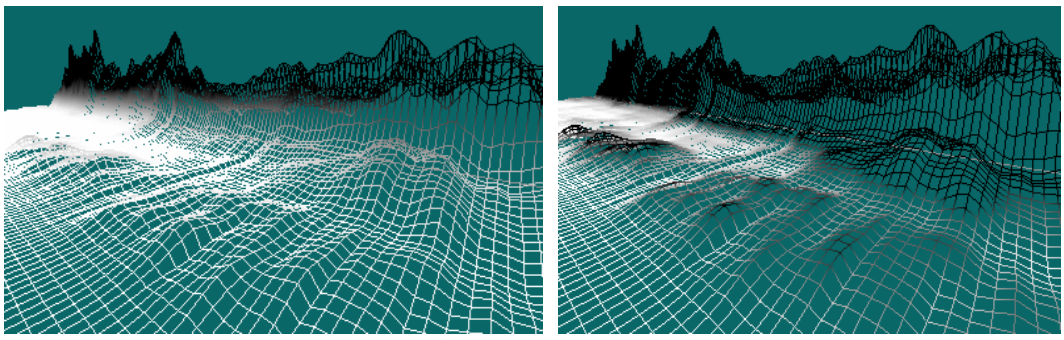


Fig 5.4.2 – Jeu sur les couleurs
Avec l'échelle de couleurs linéaire (image de gauche), on a peu de détails au niveau des petites ondulations. En modifiant l'échelle des couleurs, on arrive à focaliser le précision sur celles-ci (image de droite).

Enfin, cette fenêtre de gestion des couleurs permet de changer le paramètre dont dépend la couleur. Par défaut, celui-ci est logiquement le même que le paramètre sélectionné pour la visualisation (les points les plus hauts sont de la couleur "max", et les plus bas de la couleur "min"), mais on pourra choisir un autre champ pour référence des couleurs, et ajouter ainsi une dimension supplémentaire à la représentation, comme le montre la *figure 5.4.3*.

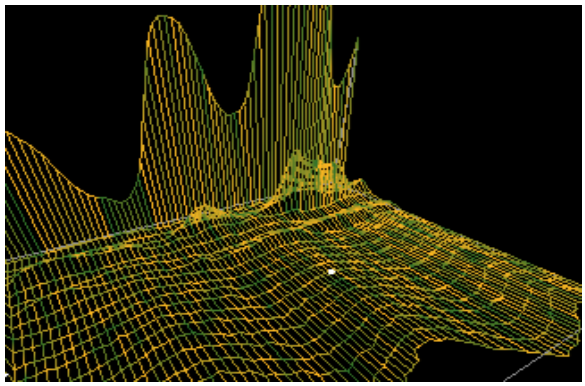


Fig 5.4.3 – Représentation de l'amplitude en fonction du temps et de la fréquence, et phase affectée à la couleur.

Points de vue

En travaillant sur un fichier SDIF, il pourra être appréciable de pouvoir mémoriser certains points de vue sur l'ensemble des données. On gagnera alors du temps en n'ayant pas à se repositionner aux endroits d'intérêt.

Ainsi, une fenêtre (*figure 5.3.4*) permet de sauvegarder à tout moment la position courante, ou de se repositionner sur une position préalablement sauvegardée.

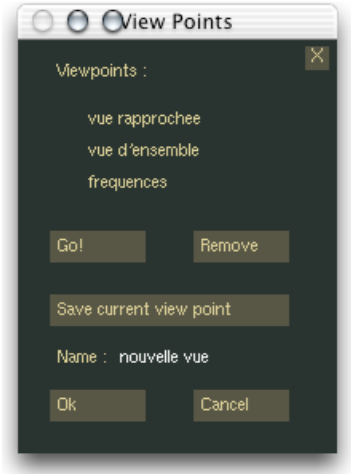


Fig 5.3.4 – Fenêtre de gestion des points de vue

Pour sauvegarder un point de vue, on devra enregistrer un certain nombre de données comme la position tridimensionnelle, l'orientation, les différentes échelles appliquées aux données, etc… Au moment de sauvegarder les modifications du fichier SDIF, on sauve ces points de vue dans un fichier séparé que l'on essaiera de charger lors des prochaines ouvertures du même fichier SDIF. Par la suite, on envisagera de sauvegarder ces données de points de vue préférentiels dans le fichier SDIF lui-même.

5.5. Vues spéciales

Certains types de visualisation sur les données peuvent être, selon les cas et les utilisateurs, plus explicites qu'une vue tridimensionnelle. *SDIF-Edit* propose donc quelques représentations alternatives, parfois plus proches de ce qui se fait généralement dans le domaine de l'informatique musicale.

Coupes 2D

Etant positionné sur un point donné de la grille de valeurs, on peut extraire de celle-ci une coupe selon les deux directions des axes horizontaux. La coupe suivant l'axe du temps nous donnera l'évolution dans le temps du champ visualisé pour un élément de la matrice, et la coupe perpendiculaire nous donne un aperçu des valeurs de toute une colonne de la matrice à un instant donné.

Cela pourra être utile par exemple dans l'utilisation avec OpenMusic puisqu'il est possible avec celui-ci de sélectionner précisément une ligne d'une matrice d'une frame donnée. On pourra donc avoir ici une vue préalable, variable avec les déplacements du point sélectionné, qui permettra de faciliter le choix des données à extraire.

La *figure 5.5.1* montre un exemple de coupes que l'on peut avoir sur un ensemble de données.

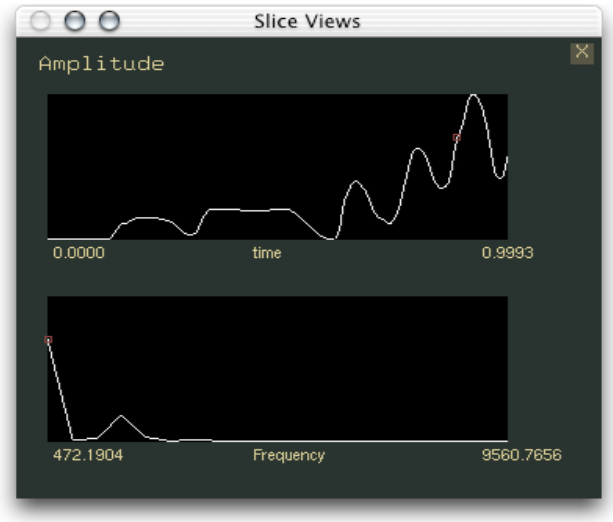


Fig 5.5.1 – Vues en "coupe"

On peut également sur cette fenêtre redessiner les courbes avec la souris. Cette fonctionnalité, si elle n'est pas d'une très bonne précision, permet toutefois de tracer rapidement des profils de courbes ou de corriger des erreurs. Elle se prête plus particulièrement aux cas ou l'on a seulement deux dimensions dans les données.

Sonagramme

Un sonagramme est une représentation graphique du son sur deux dimensions : temps et fréquence. L'intensité de la fréquence pour un instant donnée est signalée par une intensité lumineuse, ou une couleur. C'est une représentation couramment utilisée en analyse sonore.

Pour les applications de notre visualisateur sur des enveloppes spectrales (fréquence, amplitude, temps), on peut donc proposer une vue sous forme de sonagramme. Celle-ci pourra aussi être appliquée pour différents types de paramètre, libre à l'utilisateur de donner un sens aux représentations ainsi obtenues...

On représente donc la grille de données sous forme d'un rectangle dont on ajuste les axes en fonction du temps et du paramètre choisi en abscisse sur le visualisateur (idéalement, les fréquences de l'enveloppe spectrale), et dont le remplissage de couleur dépend des valeurs du champ visualisé (l'amplitude de l'enveloppe spectrale) aux différents points.

La *figure 5.5.2* montre un sonagramme obtenu à partir d'une analyse spectrale.

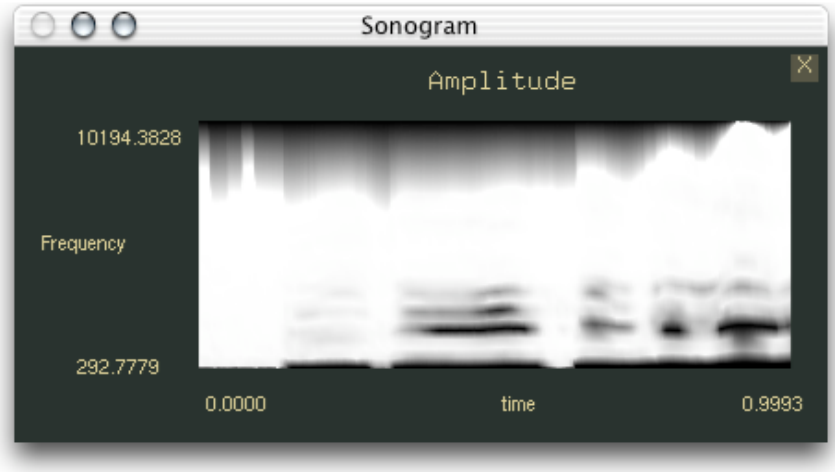


Fig 5.5.2 Sonagramme

Les variations de couleurs sont lissées par interpolations des couleurs aux différents points. De la même manière que sur la représentation tridimensionnelle, on peut modifier les couleurs du sonagramme et surtout la plage de variation afin de mettre en évidence les variations dans certaines zones. C'est même plus particulièrement justifié dans ce cas, où la troisième dimension est représentée uniquement par la couleur.

5.6. Représentation sonore

J'ai également ajouté, pour compléter le visualisateur, des informations sonores qui permettent d'aider à l'appréciation et la localisation des données.

Un fichier audio attaché

Bien souvent, les données SDIF proviennent de l'analyse d'un son. Connaissant les limites temporelles de l'intervalle de cette analyse, on pourrait, en joignant à l'application le fichier son en question, écouter cette portion sonore. Il serait ainsi possible de se repérer temporellement par rapport au son, ce qui dans bien des cas sera plus facile que de se fier aux seules données visuelles dans l'identification de régions d'intérêt.

On propose donc au démarrage d'attacher un fichier audio au visualisateur. Pour lire celui-ci, on utilise la librairie *QuickTime [8]* de *Apple* (cette fonctionnalité audio est donc limitée au seul système MacOSX). En fonction de la fenêtre temporelle actuelle, on lira le fichier audio (au format *AIFF*) d'un instant t1 correspondant au début de celle-ci, à t2, correspondant à la fin.

Comme on commence à en avoir coutume, on ajoute une sous-fenêtre à notre application qui permettra d'accéder aux fonctions audio (*figure 5.6*), elles-mêmes gérées dans un module (une classe) *AudioPlayer* autonome.

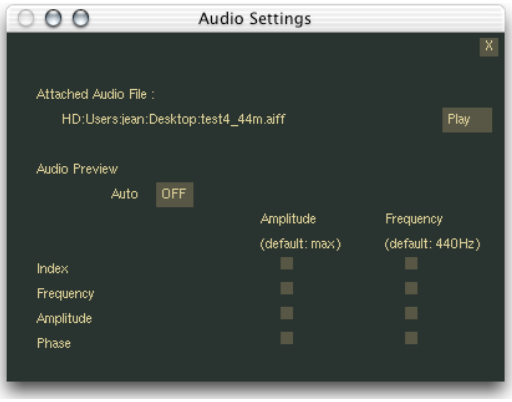


Fig 5.6 – Fonctions audio

"Ecouter" les données

Une fonction audio originale du visualisateur, également gérée dans le module *AudioPlayer,* est la possibilité d'"écouter" les données. En utilisant la librairie de synthèse sonore *STK* (*The Synthesis ToolKit in C++ [9]*), on génère et écoute une fonction sinusoïdale. Par défaut, celle-ci est de fréquence 440Hz, et d'amplitude maximale. Toutefois, on peut, à l'aide du tableau de la fenêtre audio que l'on voit sur la *figure 5.6*, affecter chacun de ces paramètres à un des champs de la matrice SDIF (le plus "logique" étant évidemment d'affecter, si l'on travaille sur ces types de données, le champ de fréquence à la fréquence, et celui des amplitudes à l'amplitude). Ainsi, on pourra faire sonner ce son généré afin d'apprécier auditivement, par fréquences et amplitudes, les caractéristiques des données en un point donné. On pourra aussi décider de faire sonner ce son quelques millisecondes à chaque mouvement du point sur les données (fonction *auto* sur la fenêtre de la *figure 5.6*).

6. L'intégration dans OpenMusic

6.1. Un éditeur pour les objets *SdifFile*

OpenMusic permet de gérer les fichiers SDIF comme des objets. Ainsi, on peut dans l'environnement OpenMusic insérer un fichier SDIF pour en extraire des données. La *figure 6.1* donne un exemple de patch utilisant SDIF.

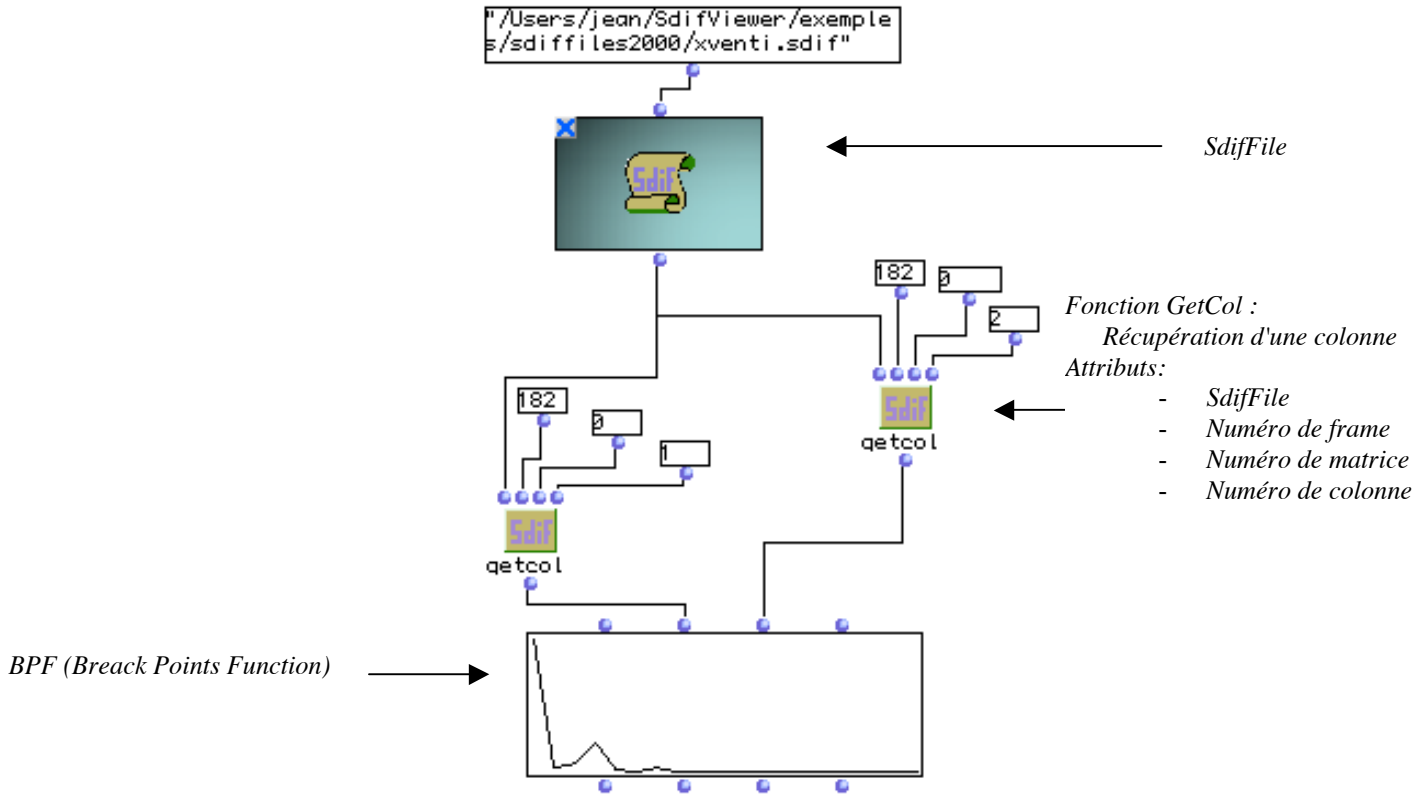


Fig 6.1 – Patch OpenMusic utilisant SDIF
On extrait du fichier SDIF deux colonnes d'une matrice, que l'on redirige sur les entrées (xpoints et ypoints) d'une BPF (breack point function)

L'idée est d'intégrer *SDIF-Edit* afin d'en faire un éditeur de ces fichiers au même titre que les éditeurs que nous avons aperçu dans la partie 1.2 (*figures 1.2* et *1.3*) pour les accords, les sons, etc… Ainsi, pour l'exemple de la *figure 6.1*, on pourra situer visuellement les données du fichier SDIF pour extraire celles qui nous intéressent.

Il faut en particulier que l'éditeur s'ouvre quand on clique sur la boîte *SdifFile*, ou qu'elle revienne en premier plan si elle est déjà ouverte. Il faudra aussi envisager le cas ou plusieurs éditeurs sont ouverts simultanément.

Pour permettre à OpenMusic (ou à *Common Lisp* plus généralement), d'accéder aux fonctions du visualisateur (lancement de l'application, ouverture de fenêtres, etc…), plusieurs solutions étaient possibles. L'une était de compiler l'application comme une librairie partagée (un *framework* sous MacOSX), sur laquelle on laisserait quelques points d'entrée pour ces appels. Une autre possibilité était de garder la forme d'application, qui serait lancée à distance, et qui communiquerait avec OpenMusic. C'est une méthode qui est possible, et qui a déjà été éprouvée, en utilisant les *AppleEvents* de MacOS, qui permettent la communication entre les diverses applications et programmes installés sur le système. C'est pour cette solution que nous avons opté, la première introduisant des complications dues aux deux boucles d'événement (celle de *Common Lisp* et celle de notre application) tournant simultanément, et la difficulté pour l'une ou l'autre des applications de récupérer la main lors de l'utilisation.

Nous avons donc configuré conjointement sur OpenMusic et sur *SDIF-Edit* des ressources correspondant aux divers messages qui allaient passer entre les deux programmes. Dans le cadre de l'éditeur, cela se fait en utilisant la librairie *AppleEvents* et en déclarant les différents événements que l'on s'attend à recevoir. On a ajouté pour cela à l'application une classe *EventManager*, qui, comme son nom l'indique, se charge de déclarer ces différents événements, de les envoyer et de les recevoir pour les rediriger vers les parties concernées (la classe *Window* pour la gestion statique générale de l'application, et le visualisateur, pour l'ouverture de fichiers).

Voici donc le dialogue qui est établi :

- Quand l'utilisateur clique sur une boîte *Sdif*, si *SDIF-Edit* n'est pas déjà en cours d'exécution, l'application est lancée. Un message d'ouverture est envoyé avec l'adresse du fichier à ouvrir.
- Les fois suivantes (si *SDIF-Edit* est déjà en cours d'exécution) seul le message d'ouverture est envoyé. Plusieurs fenêtres peuvent donc être ouvertes simultanément.
- A chaque ouverture, un identifiant unique est renvoyé à OpenMusic afin qu'il puisse tenir à jour une table référençant les différents fichiers ouverts.
- Ainsi, si un fichier est déjà ouvert et que l'on clique dessus, le message ne sera pas d'ouvrir à nouveau ce fichier, mais seulement de faire venir *SDIF-Edit*, et en particulier la fenêtre correspondant au fichier, au premier plan.
- Lors de la fermeture de la fenêtre, le visualisateur envoie un message contenant l'identifiant de la fenêtre à OpenMusic pour que la table des fenêtres soit mise à jour en conséquence.
- Lorsque la dernière fenêtre est fermée, OpenMusic en voie un message de fermeture afin de quitter l'application.

Une fois *SDIF-Edit* stabilisé, son intégration dans OpenMusic constitua déjà une étape importante dans l'achèvement du projet proposé. Comme nous l'avons expliqué en début de ce rapport (partie 1.2), il permet aux boîtes *SdifFile* des *patchs* OpenMusic de disposer d'un éditeur graphique, permettant de contrôler et de modifier leur contenu, au même titre que la plupart des autres objets de l'environnement.

6.2. Utilisation dans OMChroma

L'analyse par partiels

Un des premières applications de SDIF dans le cadre de Chroma est l'utilisation de données d'analyse pour contrôler des algorithmes de synthèse. Une représentation commune de ces données est la représentation en **partiels**. Une analyse par partiel est une analyse spectrale de laquelle on extrait les principaux traits, c'est-à-dire une représentation en intensité et en durée des fréquences dominantes du son analysé. Ce type d'analyse est obtenu en particulier à l'aide du logiciel *AudioSculpt [14]* (voir *figure 6.2.1*).

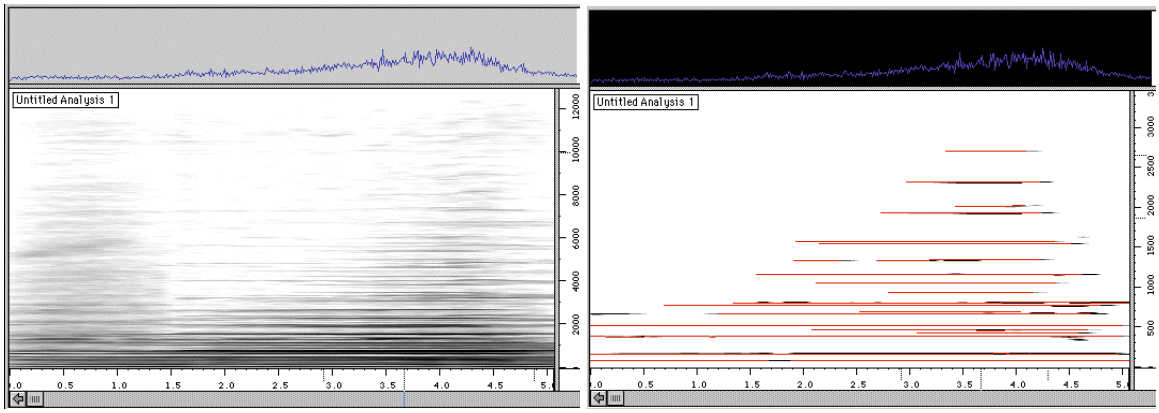


Fig 6.2.1 – Logiciel AudioSculpt: analyse spectrale (image de gauche) et extraction de partiels (image de droite)

Les partiels, que l'on voit en rouge, extraits de l'analyse spectrale sur la *figure 6.2.1*, sont donc définis par deux (voire plusieurs) points, constituant ainsi des segments de droites (s'ils sont constitués de deux points seulement) ou des lignes brisées (s'ils comportent plus de deux points). Chaque point est défini par un temps, une fréquence, une amplitude, et éventuellement d'autres paramètres. En segmentant cette représentation dans le temps, on peut extraire de l'analyse spectrale une représentation simplifiée "*Chord Seq*" (littéralement: séquence d'accords), plus adaptée à la synthèse.

Le système *Chroma* applique sur ces données divers traitements, de filtrage, de seuillage, etc., afin de les utiliser pour le paramétrage de fonctions de synthèse. Celles-ci sont donc considérées dans l'environnement de composition comme des modèles d'objets musicaux. Cependant, à l'heure actuelle, *AudioSculpt* ne permet pas de sauvegarder les résultats de ces analyses au format SDIF. En effet, si une analyse spectrale se prête bien au stockage sous le format SDIF, les partiels, en revanche, constituent un type de données de description sonore délicat puisqu'ils sont en réalité constitués de points ayant chacun un temps. On ne peut donc plus considérer cette analyse comme un flux échantillonné ayant à chaque instant un certain nombre de paramètres et de valeurs, mais on est obligé de considérer chaque partiel comme une entité symbolique indépendante. Or, du fait de la taille des données manipulées, leur stockage au format SDIF constituerait un atout important.

Un nouveau type SDIF

Nous allons donc exploiter la flexibilité du format SDIF pour commencer à y introduire des données non plus uniquement sous la forme traditionnelle, mais se rapprochant d'une conception symbolique des données, plus adaptée à la Composition Musicale Assistée par Ordinateur. Pour cela nous avons défini de nouveaux types SDIF correspondant aux propriétés de l'analyse par partiels.

Nous avons défini type de matrice SDIF, appelé *PartialSet* et de signature *EASM*, représentant un partiel. Les colonnes d'une matrice représentent les différents champs décrits (fréquence, amplitude, etc..), et les lignes représentent les différents points composant le partiel. Chaque frame (on définit un type : *EASF*) contiendra un partiel, c'est-à-dire une matrice *EASM*. La *figure 6.2.2* résume la structure d'une telle matrice.

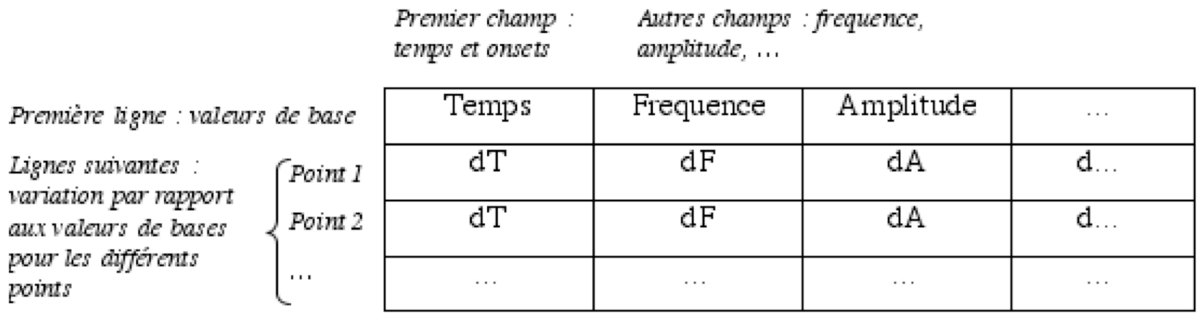


Fig 6.2.2 – Le type de matrice EASM

La particularité de ce type de matrice réside dans l'affectation des premières lignes et colonnes :

- La première ligne indiquant une valeur de référence pour chacun des champs, c'est-à-dire la valeur globale (de la fréquence, etc..) de chaque partiel, et les suivantes les écarts à cette référence pour chacun des points du partiel.
- La première colonne est utilisée pour stocker le temps de chacun des points. L'information temporelle n'est donc plus localisée uniquement au niveau des frames.

Une fois ce type défini, on peut sans grande difficulté convertir les fichiers de sorties des analyses par partiels *AudioSculpt* au format SDIF.

Toutefois, ce nouveau type de matrice, s'il respecte les conventions du format SDIF, fait quelques entailles à son utilisation usuelle. Pour connaître, par exemple, la fréquence en un point, il faudra ajouter la fréquence de référence de la première ligne à l'écart correspondant pour le point en question. De même, si l'on prend le cas de notre visualisateur, celui-ci ne peut pas prévoir a priori que les informations temporelles des points se trouvent dans la première colonne et ne pourrait pas gérer un affichage cohérent des données sachant que l'on a deux dimensions de temps (celui des frames, pour le positionnement temporel des partiels, et celui des temps différés de chacun des points composant le partiel). La représentation de ce type de données s'annonce donc quelque peu problematique.

Visualisation de partiels SDIF

Afin de permettre à notre visualisateur de manipuler les partiels et *chordSeq* via le nouveau type SDIF que nous avons défini, et sachant les particularités que nous avons reconnues à ce dernier, nous allons devoir faire une exception pour ce type de matrice et le représenter de manière alternative, plus adaptée aux données qu'elle contient. Il est évident, même si on a réussi à stocker les partiels sous formes de matrices, qu'une représentation sous la forme d'une grille traditionnelle selon le modèle des types SDIF plus "classiques" serait peu appropriée.

Toutefois, nous avons gardé la même structure (voir partie 4.2 – Représentation des données) pour stocker les données issues des matrices *EASM*, mais sachant, au moment de la représentation et des diverses opérations, que l'on a à faire à un type particulier.

Ainsi la représentation tridimensionnelle devient un ensemble de segments ou de lignes brisées délimitées par les points dont on connaît les valeurs pour les différents champs, et ordonnés et étendus dans le temps selon les informations temporelles et de décalage des différents points. La *figure 6.2.3* illustre par exemple le cas d'un ensemble de partiels "*chord seq*" (constitués de deux points chacun)

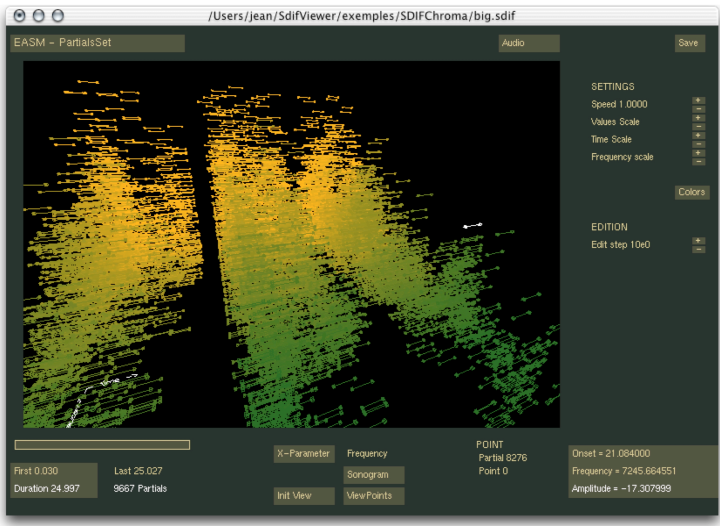


Fig 6.2.3 – Représentation d'un ensemble de partiels stockés dans un fichier SDIF

La plupart des fonctions valables pour les données SDIF classiques s'appliquent aussi aux partiels. On peut éditer des points, visualiser un sonagramme, gérer les déplacements, les couleurs, l'audio.

De plus, le fait d'avoir, dans la première des lignes, les valeurs de référence pour un partiel, va permettre de concevoir la sélection d'un partiel dans son ensemble, et l'on pourra, par modification de cette valeur uniquement, entraîner la modification du partiel entier, puisque toutes les valeurs sont définies par rapport à cette base.

On pourra également utiliser la synthèse (voir partie 5.6) pour écouter un partiel entier en utilisant les données de durée dont on dispose entre deux points successifs, et en générant une suite de sinus avec les durées, fréquences, amplitudes déterminées à partir des valeurs aux différents points.

Nous avons ainsi, au prix de quelques concessions dans le caractère générique du visualisateur, adapté *SDIF-Edit* à un type de données spécifiquement destiné à l'analyse par partiels, celle-ci étant fondamentale dans les techniques de contrôle de synthèse et dans le développement du projet OMChroma.

7. Conclusion

L'éditeur SDIF réalisé est parvenu à un état relativement stable et avancé, même si, idéalement, de nouvelles fonctions pourraient être ajoutées, ou certaines existantes améliorées. La flexibilité de l'application est satisfaisante au vu des tests avec différents fichiers réalisés jusqu'ici. La visualisation 3D, si la quantité de données n'est pas trop importante, est fluide, et les possibilités de visualisation sont assez étendues.

Le fait de disposer d'un tel éditeur dans l'environnement OpenMusic permet à présent d'avoir un réel contrôle sur les données de descriptions sonores utilisées, renforçant ainsi pour ce type d'objet le concept de modèle symbolique dans la composition. L'objet musical constitué par ces données pourra subir des traitements, que l'on sera en mesure d'apprécier visuellement, et qui se répercuteront sur la suite de l'execution. Lui-même pourra être le résultat d'un processus en aval. Il ne s'agit donc plus uniquement d'un son que l'on a analysé, mais d'un véritable modèle faisant partie intégrante de la structure des algorithmes entrant en œuvre dans le processus de composition.

SDIF-Edit a été intégré à OpenMusic, et sera présent à ce titre dans la prochaine version du CD Forum dans lequel sont distribués les logiciels de l'Ircam. A cette occasion, je présenterai en octobre, au séminaire organisé pour la nouvelle distribution, ce nouvel apport à l'environnement de CAO. Cette distribution au forum Ircam jouera un rôle important puisqu'elle devrait permettre d'obtenir des retours et avis d'utilisateurs qui auront pu expérimenter cette application pour leur usage propre.

La compilation de l'application sur plateforme Linux s'est effectuée avec succès, ce qui constitue également une réussite pour le projet.

SDIF-Edit a en effet suscité un certain intérêt dans les équipes de recherches en analyse/synthèse. En effet, à ce jour il n'existe encore aucun éditeur permettant de manipuler les fichiers SDIF. C'est pourquoi l'adaptation Linux de SDIF-Edit ouvre des perspectives d'utilisation dans de nouveaux contextes.

Références :

[1] : www.ircam.fr — Site internet de l'Institut de Recherche et Coordination Acoustique/Musique

[2] : forumnet.ircam.fr — Site du Forum Ircam, chargé de la diffusion des outils développés à l'Ircam

[3] :www.ircam.fr/produits/logiciels/openmusic.html — Le logiciel de Composition Assistée par Ordinateur *OpenMusic*

[4] : **omChroma ; vers une formalisation compositionnelle des processus de synthèse sonore.** (Marco Stroppa, Serge Lemouton, Carlos Agon). Ircam, Centre G. Pompidou, 2000.

[5] : www.ircam.fr/sdif — Spécifications, documentation et téléchargement de la librairie SDIF.

[6] : www.ircam.fr/produits/logiciels/diphone.html — *Diphone Studio*, morphing sonore, éditeur de synthèse.

[7] : www.ircam.fr/anasyn/sdif/download — SDIF library Download Directory

[8] : developer.apple.com/quicktime/ — Librairie audio *QuickTime*

[9] : ccrma-www.stanford.edu/software/stk/ — *The Synthesis ToolKit,* Librairie C++ de synthèse audio

[10]:www.opengl.org/developers/documentation/glut — *OpenGL Utility Toolkit*. Librairie multi plateforme pour l'écriture de programmes avec OpenGL.

[11] : http://www-users.york.ac.uk/~rpf1/glut.html — Version de *GLUT* par Rob Fletcher permettant entre autres le contrôle de la boucle d'événements et de la fermeture des fenêtres.

[12]:http://developer.apple.com/samplecode/Sample_Code/Graphics_3D/glut.htm — Distribution de *Glut* par *Apple* pour *MacOSX* prenant en compte les modifications proposées précédemment.

[13]: http://astronomy.swin.edu.au/~pbourke/surfaces/spline/ — Description et algorithme de calcul d'une surface Spline.

[14] : http://www.ircam.fr/produits/logiciels/audiosculpt.html — Logiciel *AudioSculpt*, éditeur de sonagrammes.

Annexe 1 : Planning du stage : tableau récapitulatif

Ce tableau présente, par semaine, les principales étapes et évènements du déroulement de mon stage. Les parties en *italique* se réfèrent à des travaux mentionnés dans le présent rapport.

5 Mai	12 Mai	19 Mai	26 Mai
Début du stage Prise en main du nouvel environnement et du logiciel OpenMusic	**Définition des objectifs**. Etude de la librairie SDIF et compilation pour MacOSX Essais de développement 3D en Java, C, etc..	Réunion pour déterminer les **besoins d'un éditeur SDIF** **Début de développement** en C++ avec la librairie Glut	*Lecture d'un fichier SDIF et élaboration des structures de stockage de données* Premières visualisations 3D

2 Juin	9 Juin	16 Juin	23 Juin
Différents cas de visualisation 3D *Navigation* *Sélection de points à l'écran et modification des valeurs*	*Mises à l'échelle et gestion des différents types de données* *Ecriture dans le fichier SDIF*	**Premier prototype (v0.1)** *Visualisation d'un paramètre en fonction d'un autre*. *Sélection temporelle* Tests d'interfaçage avec OpenMusic via une librairie partagée	*Remaniement de l'interface graphique* Récupération et **tests** de fichiers SDIF **Présentation de la nouvelle version** a la réunion perspectives développement

1 Juillet	7 Juillet	14 Juillet	21 Juillet	28 Juillet
Edition avec technique de surface spline. *Optimisation de l'interface*	*Gestion des fenêtres multiples*. Bilan avec Marco Stroppa. *Gestion des couleurs* *Vues en coupes*	*Sonagramme* *Partie Audio (player)* *Gestion des points de vue*.	*Synthétiseur audio avec STK*. **Connection OpenMusic** *en utilisant une application et les AppleEvents*.	Rapport de stage

4 Août	11 Août	18 Août	25 Août
Réunion avec Marco Stroppa pour tests Format SDIF pour les partiels *Adaptation aux partiels*	Tests de manipulation des structures NVT SDIF *Stabilisation de l'application et du système de fenêtres*	Doc. utilisateur Rapport de stage	Tests d'utilisation avec **Chroma** Préparation de la soutenance et rapport de stage. Début du portage Linux

1 Septembre	8 Septembre	15 Septembre	22 Septembre
Compilation Linux Retouches d'après avis d'utilisateurs. Préparation d'exemples	Soutenance de stage		