Gsharp, un éditeur de partitions de musique interactif et personnalisable

Christophe Rhodes* — **Robert Strandh****

* *Department of Computing*
Goldsmiths, University of London
London SE14 6NW
United Kingdom
c.rhodes@gold.ac.uk

** *LaBRI, Université Bordeaux 1*
351, Cours de la libération
33405 Talence Cedex
France
strandh@labri.fr

RÉSUMÉ. *Dans cet article, nous présentons Gsharp, un projet dont le but est la création d'un éditeur de partitions de musique traditionnelles. Le logiciel Gsharp est écrit en Common Lisp et utilise CLIM (Common Lisp Interface Manager) comme bibliothèque pour l'interaction avec l'utilisateur. De nombreux algorithmes et structures de données ont été inventés afin d'assurer un performance acceptable pour de simples interactions comme l'insertion ou la suppression d'une note ou d'un accord.*

ABSTRACT. *In this article, we present Gsharp, a project with the purpose of creating an editor for traditional music scores. Gsharp is written in Common Lisp, and uses the CLIM (Common Lisp Interface Manager) library for interactions with the user. Several new algorithms and data structures were invented in order to ensure acceptable performance for simple interactions such as inserting or deleting a note or a chord.*

MOTS-CLÉS : *édition de partitions, Common Lisp, CLIM, personnalisation, logiciels interactif*

KEYWORDS: *score editing, Common Lisp, CLIM, customization, interactive software*

DN – 11/2008. Documents musicaux, pages 9 à 28

1. Introduction

Gsharp est un projet dont le but est la création d'un éditeur de partitions de musique. Il s'agit d'un logiciel *interactif* dans le sens qu'après chaque action de la part de l'utilisateur, le résultat final de la mise en page est visible. L'un des buts principaux de Gsharp est de permettre une *personnalisation* très poussée par l'utilisateur. Ainsi, un utilisateur avec suffisamment de connaissances peut rajouter sa propre notation, sans intervenir dans le code source de l'application.

Un objectif important de ce projet est de créer une application dont l'utilisation par un compositeur, par un musicien ou par un graveur doit être efficace. Ainsi, les méthodes d'interaction de Gsharp ne sont pas forcement adaptées à un utilisateur occasionnel, mais optimisées pour un usage fréquent.

Étant donné le public visé, un autre objectif important de Gsharp est la possibilité de créer des partitons conformes aux règles de la gravure musicale. Ainsi, le logiciel contient des règles pour déterminer la mise en page, l'espacement des différents caractères, la pente des barres de croches, etc. Ultérieurement, l'utilisateur sera capable de modifier ces règles, voire d'en fournir son propre jeu.

Compte tenu de l'usage par des professionnels de ce logiciel, il est important de pouvoir traiter des partitions de très grande taille. Nous avons conçu des algorithmes des mises en page dont l'efficacité est essentiellement indépendante de la taille de la partition, ce qui permet à Gsharp de traiter des partitions de plusieurs centaines de pages sans dégradation de la performance interactive.

Gsharp est un logiciel libre, sans les moyens d'une entreprise commerciale pour son développement. Entre autre pour cette raison, le langage de programmation Common Lisp a été choisi pour l'implémentation. Ce langage simplifie également la personnalisation du logiciel en permettant le chargement dynamique de code arbitraire à l'exécution pour modifier son comportement.

La section 2 donne l'historique du développement du logiciel Gsharp, ainsi que la justification d'un certain nombre de décisions de conception. La section 3 traite de la méthode principale d'interaction avec le logiciel. Dans la section 4, nous traitons les principaux algorithmes et structures de données. La section 5 traite des difficultés de la présentation graphique d'une partition, en particulier afin d'obtenir une qualité acceptable sur un écran de faible définition. Dans la section 6, nous justifions nos choix du langage de programmation et de la bibliothèque d'interaction pour implémenter Gsharp — des choix qui nous permettent de fournir plusieurs axes de personnalisation, que nous discutons dans la section 7. Finalement, nous concluons dans la section 8 avec les travaux qui restent à faire pour que Gsharp soit directement utilisable pour un public large.

2. Historique

2.1. *Logiciels musicaux*

La mécanisation de la notation musicale est bien ancienne ; en plus du type mobile adapté à la musique, en 1886 on vendait déjà une machine à écrire musicale (la « Columbia Music Typewriter » de Charles Spiro). Les premiers systèmes permettant la production de partitions ont été créés dans les années 1960 ; par exemple, le logiciel MUPLOT opérait en temps différé pour produire à partir de texte simple une partition à une voix par portée, avec bien entendu certaines limites strictes. Des logiciels du même style, avec action à temps differé et production de partitions, ont été developpés de façon continue depuis : par exemple, MusicTeX (une bibliothèque pour le langage de composition de documents TeX), et Lilypond (Nienhuys *et al.*, 2003). Ce dernier à aussi une interface graphique, denommée « denemo », mais le système agit surtout en temps differé.

Il existe cepandant des systèmes qui montrent immediatement une representation graphique de la partition ; les logiciels experimentaux et commerciaux, tels que Mockingbird (Maxwell *et al.*, 1984), Finale et Sibelius ont souvent cette capacité. Nous soulignons dans cette catégorie surtout Igor Engraver (Terenius *et al.*, 2002), car celui-ci est écrit entièrement en Common Lisp (voir section 6).

Il ne faut pas mélanger les logiciels adaptés à l'origine à la production de partitions et ceux qui servent surtout à la production de musique, comme par exemple les séquenceurs MIDI. Les logiciels Cubase et Rosegarden (Cannam *et al.*, 2008) sont plutôt de cette catégorie ; bien qu'ils puissent produire les partitions, celles-ci sont en général de qualité moindre que celles produites par les logiciels de gravure.

Il y a aussi de nombreux logiciels adaptés à l'analyse de musique ou à la composition algorithmique. Une liste complète serait trop importante, mais nous voulons souligner l'existence de plusieurs environnements de ce genre écrit en Lisp, par exemple les langages de programmation visuels Patchwork (Laurson, 1996) et OpenMusic (Agon *et al.*, 1999) de l'IRCAM, et le langage de traitement de signaux musicaux Common Music (Taube, 2004) de Stanford.

2.2. *Développement de Gsharp*

La première version du logiciel était écrite en Tcl/Tk en 1994 par un groupe d'étudiants à l'université Bordeaux 1. Cette version nous a permis une évaluation des méthodes d'interaction, et suite à cette évaluation, une deuxième version, également en Tcl/Tk a été réalisée dans le cadre d'un stage de fin d'études.

Le problème principal de l'utilisation d'un langage de « script » comme Tcl/Tk est que la performance risque de ne pas être satisfaisante. La solution proposée par des défenseurs de ce type de langages est d'écrire les modules importants en utilisant un langage permettant la génération de code natif et efficace comme le C, et de

réserver le langage de « script » pour « coller » ces modules et pour personnaliser l'application. C'est une belle théorie, mais qui marche beaucoup moins bien en pratique, surtout pour des applications comme Gsharp ou la performance est assez importante. L'un des problèmes est la mise au point du programme. Le mélange de plusieurs langages donne un logiciel assez difficile à maîtriser. Un autre problème est la nécessité d'écrire de grandes quantités de code en utilisant un langage d'un niveau relativement bas (comme le C) qui n'est pas très économique en ce qui concerne le temps de développement.

Nous avons donc cherché un autre langage. Ainsi, plusieurs versions de Gsharp ont été développées en Scheme. À l'époque, il y avait très peu de systèmes Scheme libres permettant la génération efficace de code natif, ce qui nous a donné essentiellement les mêmes problèmes que Tcl/Tk avec un mélange entre du code Scheme et du code C. De plus, Scheme nous a posé d'autres problèmes, comme le manque de système de programmation par objets normalisés. Chaque implémentation du langage proposait différents systèmes, nécessitant une dépendance non souhaitée d'une implémentation particulière. Cette dépendance est particulièrement gênante lorsque la survie de l'implémentation du langage n'est pas garantie, ce qui est souvent le cas d'implémentations libres. Un autre problème était l'absence de bibliothèques normalisées pour la programmation d'interfaces graphiques pour le langage Scheme.

À cause de ces nombreux problèmes de choix du langage, nous avons cherché une autre solution. Ainsi, en 1998, nous avons écrit la première version en Common Lisp.

À l'époque, alors que les implémentations commerciales de Common Lisp avaient des bibliothèques très complètes pour la réalisation d'interfaces graphiques, ce n'était pas le cas des implémentations libres. La seule bibliothèque normalisée pour ces implémentations était CLX, une bibliothèque que l'on peut décrire comme la version Lisp de la bibliothèque Xlib écrite en C pour le système de fenêtrage X11. C'est une bibliothèque de très bas niveau permettant la présentation de texte et d'objets graphiques simples, mais ne contenant pas les éléments d'une interface graphique moderne comme des boutons, des menus, etc.

Pourtant, une norme, appelée *Common Lisp Interface Manager* (CLIM) (McKay *et al.*, 1994) existait pour une bibliothèque très sophistiquée permettant la création d'interfaces graphiques pour une grande diversité d'applications. Cette norme avait même plusieurs implémentations, mais toutes commerciales. C'est pourquoi en 2000, nous avons suspendu le développement de Gsharp afin de réaliser une version libre de la norme CLIM. Le résultat est le logiciel McCLIM (Strandh *et al.*, 2002) contenant actuellement environ 100 000 lignes de code Lisp.

Le développement de Gsharp redémarrait graduellement en 2002 avec la conversion de son interface graphique vers CLIM et plus sérieusement en août 2003 avec l'implémentation de plusieurs composants-clés du logiciel actuel.

Depuis 2003, nous avons rajouté un système de rendu des polices de caractères musicaux permettant la publication de partitions sur le web sous la forme de pages HTML. Un projet Google « Summer of Code » nous a permis le rajout d'importation

et d'exportation de fichiers MusicXML, et un grand nombre d'autres améliorations ont également été effectuées.

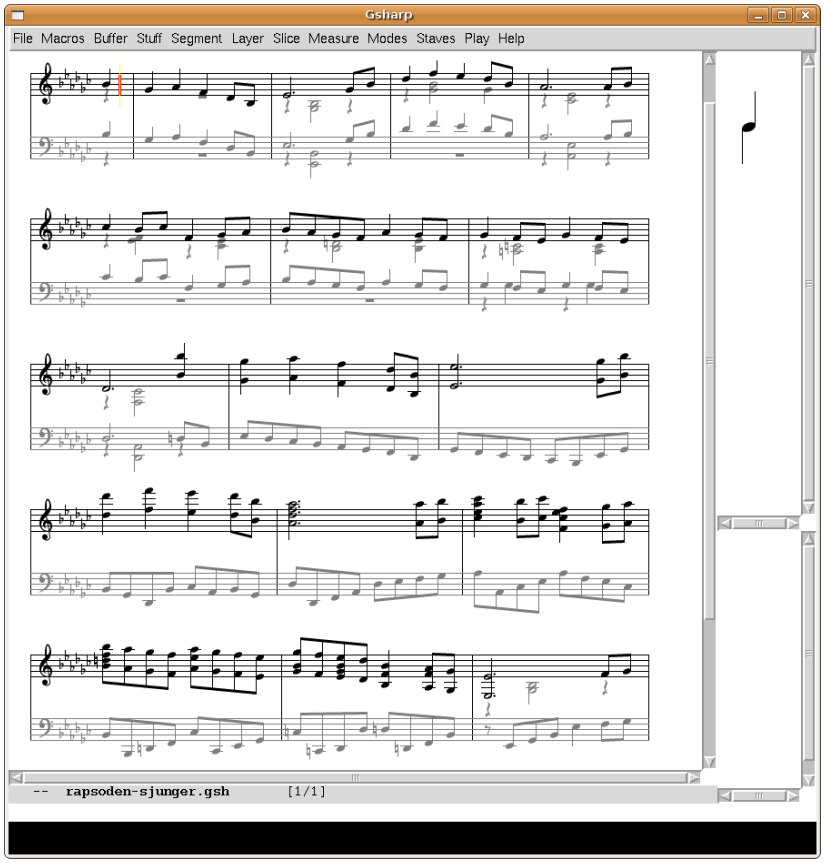

Figure 1. *Capture d'écran d'une fenêtre Gsharp*

Actuellement, Gsharp est très utilisable, mais un grand nombre de notations doit être rajouté afin que le logiciel puisse être utilisé par un public très large. Nous n'envisageons pas de modifications radicales de la structure interne du logiciel, et le rajout de ces notations devrait être possible par des personnes sans connaissances profondes de l'architecture interne du logiciel.

3. Méthode d'interaction

3.1. *Clavier ou souris*

Plusieurs décisions fondamentales concernant la méthode principale d'interaction ont été prises très tôt dans le projet. Une classification utile des méthodes de saisie (Byrd, 1984, chapitre n° 3) divise les systèmes entre ceux considérés comme « temps réel », où l'on joue une voix d'une partition, souvent sur un clavier MIDI et des systèmes « temps non réel », similaires à un logiciel de traitement de texte. Cette même classification divise également les systèmes entre ceux qui traitent la hauteur réel des notes et ceux qui traitent la hauteur de notes comme étant *symbolique*.

Pour Gsharp, nous n'avons pas encore envisagé une méthode d'interaction en temps réel, car cela augmente considérablement la complexité de l'interprétation des rythmes (quantisation, calcul des mesures). Sans interaction en temps réel, un clavier MIDI n'améliore pas considérablement la productivité. Par conséquent, nous avons décidé de réaliser un système qui traite la hauteur des notes comme étant symbolique. Finalement, nous avons décidé, pour des raisons de productivité de l'utilisateur, que l'intéraction se ferait principalement par le biais d'un clavier d'ordinateur ordinaire, plutôt qu'en employant une souris.

On peut croire que l'utilisation d'une souris soit plus judicieuse, car elle permet très facilement à l'utilisateur d'indiquer une position dans la partition. Mais la position indiquée est une position *physique* par rapport à la mise en page de la partition, alors que ce qui est souhaité est une position dans la structure de donnée abstraite qui représente la partition de manière logique plutôt que physique. Le problème ici est qu'une position physique peut correspondre à plusieurs positions logiques ce qui complique l'utilisation d'une souris.

Par exemple, si une portée contient deux voix, une position physique (position verticale et horizontale du pointeur) quelque part dans cette portée est ambiguë, car il est impossible de déterminer la voix souhaitée. Une position logique, par contre, contient la voix souhaitée ainsi que la position actuelle dans cette voix.

Afin que l'utilisation d'un clavier soit pratique, il faut que la position logique soit déterminée autrement que par une indication explicite à chaque interaction, comme c'est le cas d'une souris. La solution traditionnelle de ce problème, et également adoptée par Gsharp, est d'introduire la notion de *curseur*. Un curseur est une position logique dans la structure abstraite représentant la partition. Toute opération est effectuée par rapport à cette position logique, et chaque opération doit indiquer non seulement l'effet sur les objets de la partition, mais également la modification éventuelle de la position du curseur. De plus, il est nécessaire de fournir des opérations dont le seul but est de modifier la position du curseur, sans modifier le contenu de la partition.

Dans la version actuelle de Gsharp, les interactions ressemblent à celles de l'éditeur de texte Emacs (Stallman, 1985). Le déplacement du curseur à l'intérieur d'une voix se fait donc avec les raccourcis `C-f` et `C-b`. Afin de générer une suite d'accord

chacun contenant une seule note, il suffit de taper une lettre pour chaque note correspondant à son nom anglo-saxon : `c`, `d`, `e`, `f`, `g`, `a`, `b`. Afin de rajouter une note à l'accord derrière le curseur, la même note est tapée en majuscule. D'autres opérations, mois fréquentes, nécessitent que l'utilisateur tape le *nom complet* de la commande (précédé de `M-x` comme pour Emacs). C'est notamment le cas pour les commandes dont le but est la modification globale de la partition, comme le rajout d'une portée ou d'une voix. Ces commandes et raccourcis sont entièrement personnalisables : voir la section 7 pour plus de précision.

3.2. *Macros clavier*

Un avantage considérable de l'utilisation d'un clavier plutôt qu'une souris, et c'est en effet l'une des raisons principales de cette décision initiale, est la possibilité d'avoir des *macros clavier* (Stallman, 1985; Strandh *et al.*, 2007). Une macro clavier est une suite de touches enregistrée dont l'exécution peut être répétée autant de fois que nécessaire ; ceci permet à l'utilisateur d'automatiser des tâches répétitives sans programmation nécessaire.

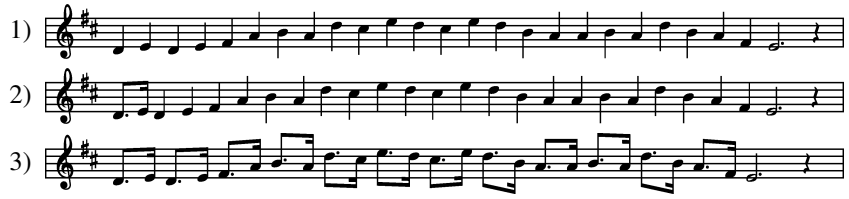

Figure 2. *Exemple d'une tâche rendue plus facile par les macros clavier : voir section 3.2 pour l'explication*

Un exemple d'une tâche considérablement simplifiée grâce aux macros clavier est la saisie d'une mélodie avec un rythme complexe mais régulier, comme l'Humoresque d'Antonín Dvořák. La manière la plus directe de saisir une telle mélodie nécessite une modification de la durée de chaque note, soit une opération supplémentaire pour chaque note saisie. Grâce aux macros clavier, l'utilisateur peut commencer par saisir les notes sans tenir compte de la durée (figure 2, partie 1), puis remettre le curseur au début de la partition et créer un macro clavier qui change les durées des deux notes suivantes avec la suite de touches « `C-x ( C-f ] . C-f [ [ C-x )` » (figure 2, partie 2). Cela accompli, pour changer les durées des vingt-deux notes suivantes il suffit de répéter le macro clavier onze fois, avec « `C-u 1 1 C-x e` » (figure 2, partie 3).

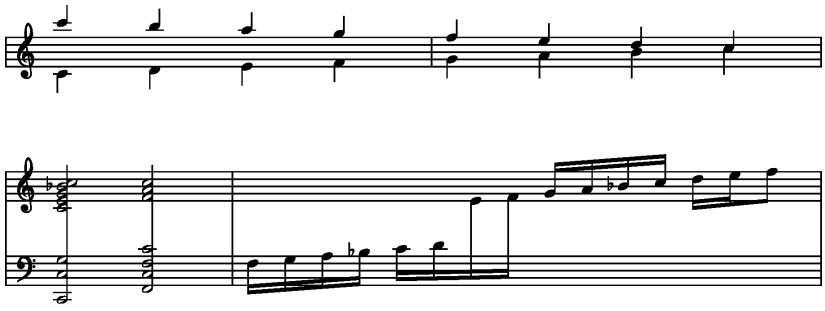

Figure 3. *Exemple de structure non hierarchique : deux voix sur une seule portée (portée du haut), et une seule voix sur plusieurs portées (portées du bas)*

3.3. *Le rythme*

Un aspect de Gsharp qui surprend souvent les nouveaux utilisateurs du logiciel est qu'il permet à une mesure de contenir n'importe quelle suite d'événements musicaux, sans tenir compte de la durée totale de ces événements. Il peut paraître normal pour un logiciel de ce type de vérifier la correspondance entre la durée des mesures de la partition et le mètre du morceau en question. En fait, ce n'est pas une bonne idée. D'abord, une telle vérification serait inutilisable pendant la création de la partition, où l'utilisateur aura besoin de créer de nouvelles mesures et de modifier le contenu de mesures existantes, ce qui crée souvent des situation temporaire où la durée d'une mesure peut ne pas être correcte.

On pourrait alors envisager deux *modes* différents ; un mode *édition* où aucune vérification n'est faite, et un mode *finalisation* où les mesures sont vérifiées. Mais il existe de nombreux exemples de partitions réalisées par des professionnels qui ne respectent pas le mètre du morceau. Il faut donc que le logiciel (comme Nightingale[1]) permette la création de partitions de ce type sans les rejeter.

4. Algorithmes et structures de données

4.1. *La structure de données « buffer »*

La structure logique d'une partition de musique est très compliquée, puisqu'il y a beaucoup de niveaux d'opérations, et ce n'est que rarement qu'un objet graphique représentant un objet musical soit logiquement placé dans une simple hiérarchie. Par exemple, les notes sont souvent (mais pas toujours !) associées à une voix, créant ainsi une structure « horizontale » ou temporelle. En même temps, les notes sont réunies

1. `http://www.ngale.com/`

dans des éléments « verticaux » ou simultanés. Un autre exemple est que les notes de plusieurs voix peuvent figurer sur une seule portée simple, mais les notes d'une seule voix peuvent également figurer sur plusieurs portées, à la fois simultanément et séquentiellement. (voir figure 3).

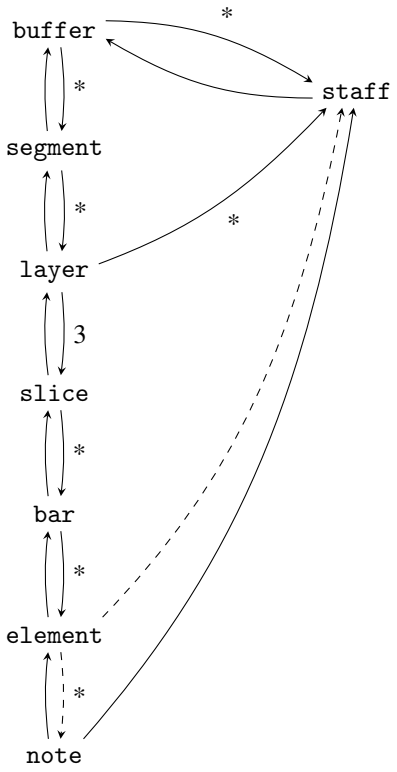

Figure 4. *La structure de données « buffer »*

Il n'y a donc pas de hiérarchie unique capable de décrire les relations que l'on souhaite pouvoir représenter dans une partition de musique, ce qui veut dire que la structure de données naturelle ne serait pas un arbre, mais plutôt un graphe. C'est surtout pour cette raison que les systèmes de sérialisation tels que `**kern` (Huron, 1997) et MusicXML (Good, 2001) ont des difficultés à exprimer même des exemples relativement simples avec des voix non continues, à savoir qui apparaissent et disparaissent dans le temps. MusicXML, par exemple, introduit des marqueurs `<forward>` et `<backward>` pour pouvoir faire référence aux événements hors séquence, tandis que `**kern` est incapable de représenter des partitions contenant des voix de ce type.

C'est pour cela que nous avons choisi une structure de données, visualisée dans la figure 4 avec la possibilité explicite de représenter plusieurs hiérarchies simultanées. Une partition entière est représentée par un objet de type « buffer », qui contient

plusieurs objets de type « segment », censés représenter une section logique. Chaque segment a trois objets de type « slice », dont un pour la majorité du contenu (les deux autres pour des cas speciaux de début et fin de section). Chaque objet de type « slice » contient en ensemble objets de type « layer » ou voix. Ces objets contiennent chacun une séquence d'objets de type « bar ». Ici un objet de type « bar » représente un *point de synchronisation*, plutôt qu'une mesure physique avec une durée fixe (voir section 3.3). Finalement, chaque objet de type « bar » contient une séquence d'objets de type « element » avec une durée temporelle (qui peut être zéro, par exemple pour les changement d'armature).

Les hierarchies multiples se voient dans la figure 4 quand on considère les objets « staff » (portées) et leur association aux « elements » et « notes » : comme précisé au-dessus, les notes d'un accord peuvent être sur plusieurs portées, donc les éléments qui representent les notes ne sont pas sur une portée du tout. Les éléments comme les changements d'armature, par contre, sont par définition associés à une portée unique, et donc mantiennent un lien pour démontrer cette association.

Pour sauvegarder les partitions répresentées en memoire par cette structure de données, nous utilisons une facilité presente dans le langage Common Lisp (voir section 6) qui permet d'ecrire en format textuel une représentation des structures de données arbitraire, y compris les structures cycliques, et puis de reconstruire une strucure isomorphe en lisant le même texte. On ne perd donc aucune information en passant par le texte ; par contre, il nous faut faire un peu d'effort pour assurer la compatibilité des nouvelles versions du logiciel avec les anciens formats ; avant l'émission de la structure de données même, Gsharp écrit un identifiant de version, qui permet eventuellement de construire un objet compatible avec la version courante du logiciel.

4.2. *Complexité des opérations*

Afin qu'un logiciel comme Gsharp soit perçu comme rapide par l'utilisateur, il est essentiel que des interactions simples et fréquentes aient un effet immédiat sur le résultat visible. Voici une liste partielle d'opérations que nous considérons comme « simples » :

– l'insertion ou la suppression d'une note ou d'un accord ;

– la modification de la hauteur d'une note dans un accord existant ;

– la modification d'une altération d'une note existante :

– la modification de la durée d'une note existante ou d'un accord existant ;

– l'insertion ou la suppression d'une barre de mesure ;

– l'attachement ou le détachement d'une suite d'accord par une barre de croche.

Par contre, certaines opérations sont déjà perçues par l'utilisateur comme étant « complexes », et ce dernier s'attend à un certain délai pour que l'effet d'une telle opération soit visible.

Voici une liste partielle d'opérations que nous considérons comme « complexes :

– la modification de la tonalité d'un morceau ;

– la transposition ;

– l'exécution d'une macro clavier agissant sur une grande partie de la partition.

Ces opérations sont complexes, car elles peuvent potentiellement influencer la présentation de la totalité de la partition.

4.3. *Le calcul de la partition*

Conceptuellement, Gsharp doit, après chaque interaction, exécuter les étapes suivantes afin d'obtenir une présentation finale de la partition :

– les mesures de chaque voix doivent être alignées afin d'obtenir la suite de l'ensemble des mesures de la partition entière ;

– les événements musicaux dans chaque mesure doivent être alignés selon leur durées ;

– la place occupée par chaque mesure doit être calculée ;

– en fonction de la place de chaque mesure, la partition doit être découpée en *pages* et souvent en *systèmes* à l'intérieur de la page ;

– la mise en page des pages visibles doit être effectuée ;

– les pages visibles doivent être affichées.

Étant donnée que Gsharp a été conçu pour traiter des partitions de très grande taille (des centaines de pages), il faut clairement éviter la vaste majorité de ces calculs pour des interactions simples, afin d'obtenir une performance raisonnable.

Afin d'éviter les trois premières étapes dans la plupart des cas, nous avons introduit un découpage de la partition en *sections*. Les mesures de chaque section sont alignées séparément. Ainsi, l'insertion ou la suppression d'une barre de mesure n'influence que la section actuelle. Ceci correspond clairement à la façon dont l'utilisateur conçoit une partition, dans le sens qu'une modification locale de ce type ne doit pas influencer une partie de la partition éloignée de plusieurs dizaines de pages. Par contre, nous cherchons toujours une méthode d'interaction permettant l'utilisation commode de ces sections.

Pour le découpage en pages et en systèmes, nous avons inventé un algorithme basé sur la *programmation dynamique* (Cormen *et al.*, 1990, chapitre n° 16). Essentiellement, chaque découpage possible de la partition en page est stocké. En modifiant l'algorithme traditionnel de la programmation dynamique pour le rendre *bi-directionnel*, à chaque itération il suffit de recalculer une petite zone (de la taille d'une page environ) autour du lieu d'édition. De plus, par la nature d'une partition, il est possible d'éviter la zone à considérer pour le calcul optimal du découpage, ce qui nous donne un calcul souvent en temps constant pour des interactions simples.

Après le découpage en pages de la partition, on ne doit traiter que les pages visibles. Ce traitement consiste à calculer la position exacte de chaque caractère musical sur la page. Nous utilisons pour cela un algorithme similaire à celui utilisé par l'éditeur de partitions Lime (Blostein *et al.*, 1991; Blostein *et al.*, 1994). Étant donné que peu de pages sont visibles simultanément, ce calcul est assez rapide bien qu'il puisse paraître complexe.

Finalement, l'affichage d'une page visible est une simple question de parcours des caractères de celle-ci et, pour chacun, d'afficher un caractère à la position calculée pendant l'étape précédente.

5. La qualité de la présentation

Dès le début, nous avons voulu une qualité de la présentation la plus haute possible, à la fois lorsque la présentation est faite sur un écran de relativement faible définition et lorsque la partition est imprimée.

La difficulté principale est d'obtenir une présentation acceptable à l'écran. Pour cela, nous avons d'abord décidé que des symboles ayant essentiellement des lignes droites verticales ou horizontales doivent être présentées avec un nombre entier de pixels. Ceci afin d'éviter des lignes trop floues générées par un système d'anti-crénelage. Puis nous avons voulu maintenir les proportions de chaque symbole musical le plus possible, afin de permettre à l'utilisateur de *zoomer* la partition en maintenant sa forme générale. Finalement, nous avons voulu aligner les côtés de certains symboles habituellement attachés comme la tête et la queue d'une note et la queue de note et le drapeau.

Toutes ces contraintes nécessitent des restrictions assez complexes sur le nombre de pixels et le type de police de caractères à employer. En particulier :

– la largeur d'une tête de note doit être 1,5 fois la distance entre deux lignes de portée. Pour des raisons d'alignement de la tête de note avec la queue de note, cette valeur doit être entière. Cela implique une distance entre deux lignes de portée paire. Étant donnée la taille de certains symboles comme le point, la plus petite distance entre deux lignes de portée est 6 pixels ;

– l'épaisseur d'une ligne de portée doit être un nombre entier de pixels, mais elle peut être paire ou impaire. Cette parité influence les points de contrôle de l'ensemble des caractères musicaux employés ;

– une ligne de portée et une ligne supplémentaire n'ont pas forcement la même épaisseur (la ligne auxiliaire est un peu plus épaisse que la ligne de portée), mais la parité de cette épaisseur doit être la même afin que le même symbole de tête de note puisse être utilisé dans les deux cas. En pratique, cela veut dire que pour des définitions qui sont d'actualité pour un écran, les deux ont la même épaisseur, alors que pour une imprimante de haute définition, il peut y avoir une différence de quelques pixels ;

– il existe d'autres considérations similaires, mais souvent plus simples à gérer. Par exemple, l'épaisseur d'une queue de note est légèrement plus faible que celle d'une ligne de portée, mais cette fois-ci, la parité n'a pas d'importance. Par contre, son épaisseur influence la position horizontale relative entre une tête de note et un drapeau attaché à l'autre bout de la queue.

Afin de répondre à toutes ces contraintes, nous avons dû abandonner les systèmes de rendu et les polices de caractères existants en faveur d'un système propre à Gsharp similaire à Metafont (Knuth, 1986), sauf que les glyphes sont calculés à la volée avec comme paramètre la distance entre deux lignes de portée. Ce système, qui sera élaboré dans la section 5, nous permet de positionner chaque ligne de contrôle très précisément, au prix de la nécessité de créer nos propres polices de caractères musicaux.

L'algorithme de mise en page doit tenir compte de ces contraintes : la position horizontale d'un accord doit toujours être un nombre entier de pixels. Cela influence un nombre d'autres algorithmes comme le calcul de la pente des barres de croches et la longueur des queues de notes. Ces calculs doivent donc être faits dans le bon ordre afin d'éviter des défauts d'alignement des symboles.

Un avantage considérable de notre système de polices de caractères est la possibilité de convertir une partition à plusieurs formats différents assez facilement et sans perte de qualité. Ceci est dû au fait que le système de polices de caractères génère des courbes de Bézier de degré 3. Le rendu de ces courbes peut être fait soit par Gsharp pour les convertir en pixels à l'écran, par une imprimante PostScript pour l'impression de la partition, par un navigateur web avec un plug-in Javascript, où par n'importe quel autre logiciel d'affichage capable de traiter ce type de courbes.

Polices de caractères

Dans cette section, nous expliquons plus en détail notre système de création de caractères musicaux.

Tous les systèmes modernes de création de polices de caractères permettent la modification dynamique de la taille des caractères selon la définition de la cible, que ce soit l'écran d'un ordinateur ou une imprimante.

En ce qui concerne le système TrueType (Apple, 1996) par exemple, chaque glyphe contient deux parties. La première partie est un ensemble de *chemins* où chaque chemin est une suite de *points de contrôle* avec une *courbe d'interpolation* entre les points consécutifs. Pour cette partie, la définition de la cible est considérée comme très élevée. La deuxième partie est un programme dont le but est de déplacer les points de contrôle légèrement afin d'adapter leurs positions à une cible d'une définition particulière. Afin de permettre à une police TrueType d'être chargée dans n'importe quelle application, le programme est écrit grâce à un langage particulier dont le code est interprété par du code dans l'application. Cette technique est appelée « hinting » et elle est brevetée.

Comme cela à été indiqué précédemment, afin d'obtenir une qualité optimale de la présentation d'une partition, nous avons besoin de calculer la forme de chaque caractère musical de manière très exacte. Un logiciel permettant ce type de calcul est Metafont, écrit par Donald Knuth pour la création de polices pour son logiciel TEX. Malheureusement, Metafont a été conçu avant la nécessité de modifier dynamiquement la taille des caractères dans un logiciel interactif. Ainsi, Metafont génère des fichiers *bitmaps* à partir de paramètres arbitraires comme la définition de la cible. L'utilisation de fichiers bitmaps de cette façon n'est compatible ni avec les besoins d'un logiciel interactif de mise en échelle des caractères, ni avec la performance requise.

Pour des raisons différentes, TrueType et Metafont utilisent chacun un langage spécialisé pour la création de chemins. Dans le cas de Metafont, c'est un véritable langage de programmation avec la notion de structures de contrôle, de variables, d'abstractions sous forme de fonctions, procédures, macros, etc.

L'utilisation d'un langage dynamique comme Common Lisp nous permet une autre possibilité, à savoir l'utilisation d'un *langage enchâssé* (Graham, 1993). Essentiellement, il est possible de réutiliser les fonctionnalités de base du langage hôte (ici Common Lisp) et de ne rajouter *que* la partie manquante sous la forme de syntaxe spécialisée pour la création de chemins. Ainsi, nous avons réimplémenté en Common Lisp la partie (relativement modeste) de Metafont permettant la description haut niveau des chemins et dont le résultat est un ensemble de courbes de Bézier. Ces courbes sont alors soit rendues directement par Gsharp à l'écran, soit utilisées pour générer du PostScript destiné à une imprimante, voire pour la création d'un programme HTML permettant ainsi la génération de pages web contenant des partitions avec la même qualité de présentation.

6. Choix du langage et de la bibliothèque d'interaction

6.1. *Le langage de programmation*

Nous avons brièvement mentionné (voir sections 1 et 2) que Gsharp a été implémenté en Common Lisp et que nous utilisons la bibliothèque CLIM pour la partie interaction avec l'utilisateur. Dans cette section, nous traitons plus en profondeur les raisons de ce choix.

Face à une tâche assez impressionnante comme l'implémentation d'un éditeur de partitions de musique, on aurait tendance à choisir un langage dont le code généré par un compilateur typique est le plus rapide possible afin d'éviter des problèmes de performance plus tard. Le choix est souvent un langage comme C++ ou éventuellement C. Afin de permettre la personnalisation du logiciel, il est alors souvent nécessaire de compléter le logiciel en introduisant ce que l'on appelle souvent un *langage de script*, typiquement un langage dynamique comme Python implémenté sous forme d'interpréteur.

Pour le créateur du logiciel, un tel mélange de langages est souvent assez compliqué à mettre au point. Pour une société commerciale avec assez de moyens, cela ne pose pas trop de problème, mais pour une petite équipe de réalisation d'un logiciel libre, ce n'est pas forcement le choix optimal.

Pour l'utilisateur sophistiqué souhaitant personnaliser l'application, avec une telle combinaison de langages, un choix difficile s'impose souvent : soit utiliser le langage script avec un risque de problèmes de performance, soit utiliser le langage de base, avec la nécessité de recompiler l'application pour chaque modification.

Le choix d'un langage comme Common Lisp (Pitman *et al.*, 1994) évite ces problèmes, car le même langage peut être utilisé comme langage de base pour implémenter l'application, et comme langage de script pour la personnaliser. De plus, il existe des implémentations de ce langage, par exemple SBCL (Newman *et al.*, 2000), permettant la génération de code pratiquement aussi rapide que celui généré par un langage comme C ou C++ (Fateman *et al.*, 1995; Verna, 2006).

6.2. *La bibliothèque d'interaction*

Comme nous l'avons déjà évoqué brièvement dans la section 2, nous avons choisi la bibliothèque CLIM (*Common Lisp interface Manager*) pour la partie de Gsharp concernée par l'interaction avec l'utilisateur, et plus particulièrement l'implémentation McCLIM de la norme CLIM. Ce choix nous donne un grand nombre d'avantages par rapport à d'autres bibliothèques similaires.

D'abord, CLIM étant une norme indépendante avec plusieurs implémentations, Gsharp est plus facilement portable à d'autres systèmes Lisp avec une implémentation de CLIM. C'est le cas notamment des deux principales implémentations commerciales de Common Lisp, à savoir Allegro Common Lisp et LispWorks.

Mais l'avantage principal est dans la conception de la bibliothèque CLIM et les services offerts au concepteur de l'application. En fait, CLIM est une bibliothèque très différente des bibliothèques plus traditionnelles du domaine, comme Qt, Tk, GTK, etc. Par la suite, nous allons expliquer brièvement ces différences.

Habituellement, les bibliothèques d'interaction nécessitent que l'application soit structurée autour d'une *boucle d'événements*. Le style de programmation qui en résulte s'appelle *programmation événementielle*, ce qui signifie que l'application consiste en un certain nombre de *traitants* d'événements déclenchés par une action de la part de l'utilisateur comme l'appui sur une touche du clavier, ou l'utilisation d'un bouton de la souris pour activer un élément de l'interface graphique.

L'inconvénient principal de ce style d'interaction est que les traitants sont exécutés dans l'environnement global du logiciel. Toute mémoire d'un contexte quelconque (comme l'événement précédent, ou un contexte censé modifier le résultat de l'action) doit être géré explicitement par l'application.

CLIM contient bien évidemment également une boucle d'événements au plus bas niveau, mais une application typique ne s'en sert pas ou peu directement. Au-delà de cette boucle d'événements, CLIM propose une *boucle de commandes*. Cette boucle infinie exécute les étapes suivantes :

– acquisition d'une *commande* ;

– acquisition des *arguments* de la commande ;

– exécution de la commande avec les arguments.

Ces commandes sont des fonctions Common Lisp augmentées avec l'étiquetage des paramètres par des *types de présentation*. CLIM propose une arborescence de types parallèle à celle de Common Lisp dont le but est d'associer à des éléments graphiques, comme les notes, accords, portés, clés etc. de Gsharp, l'un de ces types et un objet sous-jacent Common Lisp. Le code responsable de l'affichage d'un élément graphique n'indique pas directement les modalités d'interaction avec cet élément, mais uniquement son type de présentation. L'élément devient actif (cliquable) lorsqu'une commande doit acquérir un argument d'un type compatible. Ceci permet une modularité très poussée du code de l'application ce qui augmente également sa maintenabilité et la possibilité de la personnaliser.

En outre, l'existense independente des commandes de CLIM, ainsi que le fait que la bibliothèque McCLIM ait une façon d'agir comme si il y avait des graphismes bien que sans les afficher, nous permet de faire opérer Gsharp en mode « serveur » : cela nous permet de visualiser rapidement les partitions en format MusicXML, ou bien produire une visualisation des accords[2] d'une representation conforme à l'ontologie musicale OMRAS2[3] (Raimond *et al.*, 2007).

7. Personnalisation de Gsharp

Comme nous l'avons brièvement évoqué dans l'introduction, un objectif principal de Gsharp est la capacité d'adapter son fonctionnement selon les souhaits et les besoins d'un utilisateur sophistiqué. Dans cette section, nous élaborons plusieurs types de personnalisation déjà présents ou planifiés de Gsharp.

7.1. *Nouveaux symboles musicaux*

L'algorithme de mise en page de Gsharp ignore la signification exacte des éléments musicaux à mettre en page. À la place, il se sert d'un *protocole* dans le sens de (Keene, 1988), dont le but est de :

– déterminer la *durée de l'élément* afin que des éléments simultanés puissent être alignés verticalement ;

2. voir par exemple `http://rvw.doc.gold.ac.uk/omras2/widgets/chord/C:(3,5)`
3. `http://purl.org/ontology/chord`

– déterminer les *dimensions physiques* de l'élément, afin de calculer la place requises sur la page ;

– dessiner l'élément à sa place finale.

Étant donné ce protocole, un utilisateur qui souhaite rajouter à Gsharp un nouveau type d'objet, peut facilement décrire ce dernier grâce à des *classes* et des *méthodes* pour implémenter le protocole. Par exemple, il est entièrement possible d'envisager la possibilité de rajouter des « sons » à Gsharp, où un son serait représenté par sa durée, par les dimensions d'un rectangle, et par un dessin classique sous la forme de signal temporel. Grâce au langage Common Lisp, ce type d'addition ne nécessite aucune modification au code existant, et peut être entièrement contenu dans un ou plusieurs fichiers propres à l'utilisateur. Le langage permet à la fois la génération efficace de code natif à partir de ces fichiers, et le chargement dynamique du code natif dans Gsharp.

L'organisation interne de Gsharp se sert déjà de ce protocole, dans le sens que les objets de base comme les accords de notes, les paroles, etc., ne sont pas traités de manière particulière. Ils sont simplement écrits pour respecter ce protocole, ce qui donne une structure très modulaire au logiciel.

7.2. *Commandes et raccourcis*

Grâce à la bibliothèque CLIM, la structure d'une commande et éventuellement un raccourci associé sont contenus dans un objet appelé *table de commandes*.

Un utilisateur souhaitant modifier l'association entre raccourci et commande est libre de le faire de manière modulaire dans un fichier privé, sans qu'il soit nécessaire de modifier le code propre de Gsharp. De manière similaire, un utilisateur peut créer ses propres commandes dans une telle table, permettant ainsi de nouvelles opérations non prévues par les développeurs.

De plus, il est possible pour l'utilisateur de créer ses propres tables de commandes, et de personnaliser le contexte où une table est active. Cela permet notamment un ensemble de commandes différent pour différents types d'objets musicaux. Nous utilisons ce mécanisme en interne pour obtenir un jeu de commandes différent sur les paroles et sur les accords de notes.

Encore une fois, grâce au langage Lisp, ces modifications ou rajouts peuvent être effectués sans aucune modification au code source du logiciel, et sans l'aide d'un langage scripte séparé du langage de base utilisé pour implémenter le logiciel.

7.3. *Styles*

Il existe plusieurs *écoles* de gravure musicale. Les différences entre différentes écoles sont souvent mineures, mais les règles appliquées représentent un ensemble

cohérent et il est vivement conseillé de ne pas mélanger différentes règles de différentes écoles.

Un exemple représentatif est la pente des barres de croches. Dans l'école « européenne » la pente permise est beaucoup plus importante que la celle de l'école « américaine ».

Alors que cette idée n'est pas encore implémentée dans Gsharp, nous avons l'intention de paramétrer un grand nombre de calculs de la mise en page par un objet *style*, permettant ainsi à un utilisateur de créer son propre style, ou de modifier certains aspects de l'un des styles fournis.

8. Travaux futur

Il manque actuellement à Gsharp plusieurs sortes d'éléments musicaux ; la voie de développement la plus simple est d'ajouter ce dont on a besoin. Pour certains éléments, toute l'infrastructure codage est en place, mais il manque les glyphes (exemples : chiffrage de la mesure, ornements). Certains autres seront plus compliqués à introduire à Gsharp ; les liaisons d'expression ou les nuances pour des ensembles de notes, mais leur inclusion est bien sûr nécessaire.

Nous voulons ajouter à Gsharp les autres formes de notation musicale ; en plus des portées pour les paroles et la percussion, nous avons déja créé un logiciel pour la tablature de luth (Rhodes *et al.*, 2005a) à partir d'un éditeur de texte (Rhodes *et al.*, 2005b) ; incorporer la tablature, et aussi d'autres notations, les neumes par exemple, serait un objectif à relativement court terme.

Les ornements, notes détachées et autres indications sur les notes suggèrent un mécanisme général pour ajouter les annotations, comme suggéré dans la méthodologie CHARM (Wiggins *et al.*, 1993). Ceci va nous permettre de montrer toutes sortes d'informations, potentiellement extraites des analyses automatiques des enregistrements, sur une partition Gsharp.

La bibliothèque McCLIM permet déjà l'exportation des partitions en format Postscript pour imprimer et HTML/Canvas pour publier sur le web ; le développement de la bibliothèque permettera bientôt l'intéraction sur le web, utilisant les téchniques AJAX pour la communication.

Finalement, nous ésperons ajouter les fonctions auxquelles on est habitué par les éditeurs de texte : par exemple, les commandes pour la recherche et remplacement (Byrd, 2001), eventuellement basées sur un système d'expressions régulières ou sur une autre notation.

Remerciements

C.R. est partiellement financé par le projet OMRAS2 (EPSRC EP/E02274X/1).

9. Bibliographie

Agon C., Assayag G., Laurson M., Rueda C., « Computer Assisted Composition at IRCAM : Patchwork & OpenMusic », *Computer Music Journal*, vol. 23, n° 3, p. 59-72, 1999.

Apple, *The Truetype Reference Manual*, Apple Computer, Inc., Cupertino, California. 1996.

Blostein D., Haken L., « Justification of printed music », *Communications of the ACM*, vol. 34, n° 3, p. 88-99, 1991.

Blostein D., Haken L., « The Lime Music Editor : a Diagram Editor Involving Complex Translations », *Software—Practice and Experience*, vol. 24, n° 3, p. 289-306, 1994.

Byrd D., « Music-Notation Searching and Digital Libraries », *Joint Conference on Digital Libraries*, p. 239-246, 2001.

Byrd D. A., Music Notation by Computer, PhD thesis, Indiana University, 1984.

Cannam C., Bown R., Laurent G., *The Rosegarden Handbook*, 1.7.0 edn. 2008, `http://www.rosegardenmusic.com/doc/en/`.

Cormen T. H., Leiserson C. E., Rivest R. L., *Introduction to Algorithms*, MIT Press, 1990.

Fateman R. J., Broughan K. A., Willcock D. K., Rettig D., « Fast Floating-Point Processing in Common Lisp », *ACM Transactions on Mathematical Software*, vol. 21, n° 1, p. 26-62, 1995.

Good M., « MusicXML : An Internet-Friendly Format for Sheet Music », *XML Conference & Exposition*, 2001.

Graham P., *On Lisp : Advanced Techniques for Common Lisp*, Prentice Hall, 1993.

Huron D., « Humdrum and Kern : Selective feature encoding », *in* E. Selfridge-Field (ed.), *Beyond MIDI : The Handbook of Musical Codes*, MIT Press, p. 375-401, 1997.

Keene S. E., *Object-Oriented Programming in Common Lisp*, Addison-Wesley, 1988.

Knuth D., *The METAFONTbook*, Addison-Wesley, 1986. ISBN 0-201-13444-6.

Laurson M., PATCHWORK : A Visual Programming Language and some Musical Applications, PhD thesis, Sibelius Academy, Helsinki, 1996.

Maxwell III J. T., Ornstein S. M., « Mockingbird : a composer's amanuensis », *Byte*, vol. 9, n° 1, p. 384-401, 1984.

McKay S., York W., « Common Lisp Interface Manager Specification », 1994. `http://bauhh.dyndns.org:8000/clim-spec/index.html`.

Newman W. H. et al., « SBCL User Manual », `http://www.sbcl.org/manual/`, 2000.

Nienhuys H.-W., Nieuwenhuizen J., « LilyPond, a system for automated music engraving », *Proceedings of the XIV Colloquium on Music Informatics*, Firenze, Italy, 2003.

Pitman K., Chapman K. (eds), *Information Technology – Programming Language – Common Lisp*, n° 226–1994 in *INCITS*, ANSI, 1994.

Raimond Y., Abdallah S., Sandler M., Giasson F., « The Music Ontology », *Proc. ISMIR*, Vienna, 2007.

Rhodes C., Lewis D., « An Editor for Lute Tablature », *Computer Music Modeling and Retrieval*, Springer, p. 259-264, 2005a. LNCS 3902.

Rhodes C., Strandh R., Mastenbrook B., « Syntax Analysis in the Climacs Text Editor », *International Lisp Conference Proceedings*, 2005b.

Stallman R. M., *GNU Emacs Manual*, Free Software Foundation, 1985.

Strandh R., Henriksen T., Murray D., Rhodes C., « ESA : A CLIM Library for Writing Emacs-Style Applications », *International Lisp Conference Proceedings*, 2007.

Strandh R., Moore T., « A Free Implementation of CLIM », *International Lisp Conference Proceedings*, 2002.

Taube H. K., *Notes from the Metalevel : Introduction to Algorithmic Music Composition*, Studies on New Music Research, Routledge, 2004.

Terenius P., Doughty B., *Igor Engraver 1.6 Manual*, NoteHeads Musical Expert Systems AB. 2002.

Verna D., « How to Make Lisp Go Faster than C », *IAENG International Journal of Computer Science*, 2006.

Wiggins G., Miranda E., Smaill A., Harris M., « Surveying Musical Representation Systems : A Framework for Evaluation », *Computer Music Journal*, vol. 17, n° 3, p. 31-42, 1993.