Quelques petits objets...

Denis Lorrain, SONVS, CNSM de Lyon, 3 Quai Chauveau, C.P. 120, 69266 Lyon Cedex 09, FRANCE
dlo@ubaye.cnsm-lyon.fr

Résumé : *Environnement multi-tâches sous LISP, permettant de réaliser simultanément plusieurs processus musicaux en temps-réel.*

1. Introduction

On peut utiliser diverses formes LISP, permettant de gérer des boucles répétitives, afin d'atteindre l'objectif que nous venons de résumer. Nous nous sommes arrêtés à une solution mettant en œuvre des classes d'objets basés sur le mécanisme d'appels récursifs à une fonction temporaire locale, temporisés par recours à un noyau logiciel multi-tâches et temps-réel.

Il s'agit ensuite d'insérer, dans le corps d'une telle boucle récursive, les évaluations visant à faire évoluer un processus, d'étape en étape, dans le temps. Ce processus peut consister lui-même en une séquence de formes LISP quelconques ou, à titre d'exemples musicaux bien caractérisés, en la création algorithmique d'une ligne mélodique, au sens large, ou en l'exécution d'une séquence d'actions pré-déterminées.

2. Implémentation

2.1. Bases

Le travail présenté ici est implémenté en *Common LISP* [Steele 1990], plus précisément Macintosh Common LISP, version 2.0 [MCL2.0 1992]. La programmation par objets est basée sur le *Common LISP Object System* [Bobrow *et al*. 1990]. Son fonctionnement s'appuie sur le *Common LISP Compositional Environment* (*CLCE* [Letz *et al*. 1992]), qui est une prolongation de Common LISP, proposant, outre un environnement pour la composition musicale, des outils spécifiques pour la gestion d'événements MIDI et du temps-réel. Afin de réaliser cette fonctionnalité, CLCE repose lui-même sur *MidiShare* [MidiShare 1994] : "un noyau logiciel multi-tâches et temps réel, spécialement conçu pour le développement d'applications MIDI. [... Ce noyau] gère une horloge interne [qui] sert à dater [...] tous les événements reçus, ainsi qu'à spécifier les dates d'événements à transmettre." Sur cette base, CLCE propose un ensemble de primitives permettant de gérer chronologiquement, non seulement l'émission et la réception d'événements MIDI, mais beaucoup plus généralement, et surtout pour ce qui nous concerne, l'évaluation de toute forme LISP [Letz *et al.* 1992 : 94-96, 104, 106].

2.2. Principe

Les objets que nous présentons sont ainsi arrimés au temps-réel par des appels différés de procédures faisant évoluer progressivement un processus, d'étape en étape, dans la durée.

Dans une réalisation en temps-réel, on retrouve donc un environnement tout-à-fait semblable dans son principe à *FORMES*, par exemple [Cointe et Rodet 1984], qui permet à des processus, incarnés par des classes d'objets, d'évoluer dans le temps simultanément, de manière tout aussi bien indépendante qu'interdépendante selon les besoins.

En dehors des méthodes de base gérant la "naissance" et la "mort" d'instances de ces classes d'objets, une méthode principale consiste en l'activation, en l'"éveil" d'un objet, avec sa complémentaire d'arrêt, de mise en "sommeil".

En voici les structures de principe :

```
(defmethod eveille ((obj ...))
  (reinitialize-instance obj :eveil t)
  (let (...
        (deltat (slot-value obj 'deltat))
        ...
        (cmpt   0                       ))
  ...
  (labels ((interne ()
             (cond
               ((slot-value obj 'eveil)
                 ... < <  corps  > > ...
                 (after (eval deltat) (progn (incf cmpt)
                                             (interne))))
               (t
                 ...))))
    (interne))))

(defmethod dors ((obj ...))
  (reinitialize-instance obj :eveil nil))
```

On voit, particulièrement, la primitive CLCE *after* qui permet de différer, de temporiser la récursion à la fonction temporaire locale *interne* selon la durée d'un délai *deltat* que la méthode est en mesure de calculer elle-même. Après ce temps donné d'attente, la procédure sera réactivée afin de re-calculer, dans sa partie *corps*, la prochaine étape du processus qu'elle réalise, et ainsi de suite, tant que la valeur de la variable d'instance *eveil* sera *t*.

3. Objets

Quelques classes d'objets et variantes de méthodes ont été réalisées sur cette base.

3.1. zombie

La classe *zombie0* est la plus générale : elle permet l'insertion, dans le *corps* de l'objet, d'évaluations de formes LISP quelconques, aussi bien "normales" qu'appartenant à CLCE et interagissant donc éventuellement avec des entrées-sorties MIDI.

```
(defmethod eveille ((obj zombie0))
  (let ((nom    (slot-value obj 'nom   ))
        (quid   (slot-value obj 'quid  ))
        (deltat (slot-value obj 'deltat))
        (cmpt   0                       ))
  (when quid
    (reinitialize-instance obj :eveil t)
    (format *debug-io* "~&~A > début de la boucle...~%" nom)
    (labels ((interne ()
               (cond
                ((slot-value obj 'eveil)
                 (eval quid)
                 (after (eval deltat) (progn (incf cmpt)
                                             (interne))))
                (t
                 (format *debug-io* "~&~A >... fin, ~D actions.~%"
                         nom cmpt)))))
      (interne)))))
```

On remarque au passage que des équivalents des variables d'instance *first-time* et *last-time* des objets FORMES ne sont pas présents ici ; leur insertion est cependant très aisée, et a d'ailleurs été réalisée dans certaines variantes

D'autres classes *zombie* ont également été conçues spécialement pour effectuer des processus d'interpolation. L'une, *zombie2*, agit de manière comparable à celle de l'objet *randi* des environnements de type *Music V* etc. (*Csound*, par exemple [Vercoe 1993]), mais avec, de surcroît, la possibilité de faire calculer les points d'inflexion successifs du processus au moyen de formes LISP *quelconques*. Certaines autres classes d'objets réalisent des parcours temps-réel déterminés à l'avance ; l'une en fonction de valeurs de départ, d'arrivée et de durée données (équivalente à *linseg* de Csound), une autre suivant le contour d'une fonction de temps librement définie à l'avance dans un tableau (de manière analogue à l'objet *oscil1i* du même environnement).

Tout est envisageable ici, et les exemples décrits ne sont évidemment pas limitatifs.

3.2. melos

La classe *melos0* est specialisée dans la réalisation de processus mélodiques. Son corps consiste donc en l'envoi d'une *note* à MidiShare, que celui-ci transmettra à l'interface MIDI sous forme de messages normalisés *key-on* et *key-off* appropriés.

```
(defmethod eveille ((obj melos0))
  (reinitialize-instance obj :eveil t)
  (let ((nom    (slot-value obj 'nom   ))
        (pitch  (slot-value obj 'pitch ))
        (vel    (slot-value obj 'vel   ))
        (deltat (slot-value obj 'deltat))
        (dur    (slot-value obj 'dur   ))
        (pgm    (slot-value obj 'pgm   ))
        (chan   (slot-value obj 'chan  ))
        (cmpt   0                       ))
  (when (midip pgm)
    (midi-send-im clce (prog-change :pgm  (eval pgm)
                                    :chan (eval chan))))
  (format *debug-io* "~&~A > début de la boucle...~%" nom)
  (labels ((interne ()
             (cond
               ((slot-value obj 'eveil)
                 (midi-send-im clce (note :pitch (eval pitch)
                                          :vel   (eval vel)
                                          :dur   (eval dur)
                                          :chan  (eval chan)))
                 (after (eval deltat) (progn (incf cmpt)
                                             (interne))))
               (t
                 (format *debug-io* "~&~A >... fin, ~D notes.~%"
                         nom cmpt)))))
    (interne))))
```

Cette classe d'objets est donc conçue de manière à engendrer elle-même ses propres données mélodiques (hauteurs, vélocités, intervalles de temps, durées, etc.).

Sur la base du même "moteur", la classe *melos1*, elle, agit plutôt comme un *séquenceur* en puisant ses données mélodiques au fur et à mesure dans des listes fournies à l'avance.

Toutes les combinaisons de ces deux approches sont bien entendu possibles.

4. Exemples

Dans la majorité des exemples suivants, nous faisons intervenir une librairie de fonctions stochastiques [Lorrain 1980] et chaotiques (littérature pléthorique, travaux personnels et [Bidlack 1992]).

Sans tenter de justifier, si besoin était, ni même de commenter ici le recours à de telles fonctions mathématiques, on admettra simplement que ces outils permettent de *modeler* ou *mouler*, de *former* ou *profiler* des processus musicaux. Leur utilisation permet de créer des textures, des *objets musicaux* plus ou moins typés, d'une manière semblable à celle d'un architecte dessinant des contours de masses, à l'intérieur desquels des matériaux de qualité déterminée seront coulés ultérieurement.

Ces résultats sont qualifiés d'objets *musicaux*, et non *sonores* au sens schaefferien, car les illustrations que nous en présentons ici demeurent entièrement attachées au concept traditionnel de *note* et à tout ce qui s'en suit. Bien que ce fait soit sans doute facilité par la norme MIDI, soulignons tout-de-même qu'elle n'en est aucunement la cause réelle.

D'autre part, bien que nous ayons indiqué que nos processus sont essentiellement capables d'un comportement *rythmique*, en calculant eux-mêmes leurs intervalles successifs d'attente temps-réel, nous ferons aussi entendre, paradoxalement, des exemples basés sur des *ostinati* rythmiques monotones ; mais nous le ferons pour mettre en évidence d'autres aspects plus pertinents des possibilités de modelage de la matière musicale. Pour la même raison, certains des exemples ne mettront en œuvre qu'un seul processus à la fois.

4.1. Mélodie grise

Ce processus mélodique est "aléatoire" au sens le plus banal : on peut le qualifier de *gris*. Les hauteurs (entre valeurs MIDI 53 et 84), les vélocités (entre 10 et 120) et les fluctuations rythmiques (intervalles entre notes allant de 70 à 170 msec.) sont tous régis par une *distribution continue uniforme* faisant part égale à toutes probabilités. Cet exemple pourrait aussi bien être qualifié de *blanc*, puisque cette distribution correspond au spectre du bruit blanc.

Voici les formes LISP permettant de créer l'objet réalisant ce processus, de l'activer, et finalement de l'interrompre :

```
(defparameter grise (make-instance 'melos0
                                   :nom    'grise
                                   :pitch  '(+ 53 (random 32))
                                   :vel    '(+ 10 (random 110))
                                   :dur     99
                                   :deltat '(+ 70 (random 100))))
(eveille grise)
...
(dors grise)
```

4.2. Trémolo varié accentué

Pour illustrer la possibilité de modeler des processus beaucoup plus typés, voici une sorte de trémolo varié autour d'une note principale. Les excursions de part et d'autre de la note centrale 69 obéissent à une *première distribution de Laplace* (ou exponentielle bilatérale, avec un coefficient de dispersion 0.65). Les vélocités, variées en fonction d'une *distribution Beta* (coefficients 0.125 et 0.25, pour des vélocités entre 20 et 120), créent un rythme stochastique d'accents très marqués sur le fond monotone tendu par la vitesse constante du trémolo (5 notes par sec.).

La forme suivante crée l'objet réalisant ce processus :

```
(defparameter tst1
          (make-instance 'melos0
                 :nom    'tst1
                 :pitch  '(round (plapla 69 0.65))
                 :vel    '(+ 20 (round (* 100 (beta 0.125 0.25))))
```

```
                    :dur      200
                    :deltat   90))
```

où *plapla* et *beta* sont deux fonctions stochastiques synthétisant les distributions citées.

4.3. Trémolo à dispersion variable

Deux processus simultanés produisent ici une variante du trémolo précédent. Encore à vitesse constante, mais cette fois sans accents marqués (*distribution continue uniforme* des vélocités entre 60 et 120), on joue ici, autour de la note centrale, sur la dispersion variable des hauteurs. Celles-ci sont gérées par une *distribution de Gauss* dont l'écart type est modifié graduellement par un second processus d'interpolation linéaire lancé simultanément (allant de 0 à 6, et retournant à 0).

4.4. "Improvisation"

Cet exemple est plus élaboré : sa description exhaustive n'est pas possible ici, et la brièveté des extraits présentés ne rendra pas justice à l'ensemble des processus en action, qui ne s'affirment clairement que dans une certaine durée. Les hauteurs et les rythmes sont gouvernés par un "moteur" *chaotique* (*fonction logistique* de May-Feigenbaum). Le coefficient permettant de piloter cette fonction vers des cycles déterminés ou vers des plages de comportement aléatoire, varie de temps à autre. Les accents proviennent encore de l'utilisation d'une *distribution Beta* des vélocités.

Plusieurs autres processus simultanés agissent en outre sur le comportement du processus principal qui configure ainsi le résultat final audible, en faisant varier, à des intervalles de temps indépendants, des paramètres de phrasé, d'intensité globale, d'accentuation, de registre principal, etc.

4.5. Séquences cycliques

Les classes d'objets exécutant des séquences pré-déterminées puisent leurs données dans des listes. Toutes les fonctionalités d'un séquenceur sont bien entendu possibles ainsi ; mais il est surtout très aisé d'exploiter des *listes circulaires* de données. Cet exemple met en évidence de telles listes, de longueurs intentionnellement différentes (12 hauteurs, 7 vélocités et 5 durées, avec intervalles de temps réguliers).

Un second exemple reprend, simultanément à ce que l'on vient d'entendre, un autre processus utilisant des variantes des mêmes données circulaires (ainsi qu'un cycle de 9 intervalles de temps variés), avec un autre timbre et dans un registre plus aigu.

5. Conclusion

5.1. Remarque lispienne

Il est bien entendu nécessaire de pouvoir transmettre des informations entre différentes variables d'une même instance d'objet, et entre instances elles-mêmes. La première vertu serait normalement atteinte par la création de *variables locales temporaires* à l'intérieur des instances des classes, c'est-à-dire définies par l'utilisateur lui-même lors de la création de chacune des instances individuellement.

De par la portée *lexicale* des variables en Common LISP (cf. "*Scope and Extent*" [Steele 1990 : 42 *sq.*]), et faute de maîtriser une belle solution pour contourner cette difficulté, nous avons recours à des variables globales pour arriver à ces deux fins.

5.2. Ouverture

La dernière version de Common LISP sur Macintosh [MCL3.0 1995] inclut des possibilités multi-tâches (*multiprocessing*). Le portage de CLCE vers MCL3.0 étant actuellement en cours à GRAME, il est trop tôt pour analyser l'influence que cette nouvelle version pourrait avoir sur ce que nous venons d'exposer. De prime abord, sa mise en œuvre, même sur les Power Macintosh les plus puissants, s'avère d'une lenteur très décevante par rapport à ce qu'on serait en droit d'attendre de ces nouvelles machines.

Les limites de la précision chronologique, et de la puissance de calcul actuellement disponibles en réalité pour cet environnement, ne sont malheureusement pas très reculées. On espère par contre, en comptant sur de nouvelles versions de LISP et sur divers progrès technologiques, atteindre des performances comparables

à celles de *MAX* [Puckette 1986 ; MAX 1995] telles qu'on les a connues depuis plusieurs années sur les Macintosh à processeurs de la série 68000.

Ce dernier environnement possède beaucoup d'avantages, dont la modularité, la fiabilité et la rapidité ne sont pas des moindres. On admettra cependant aussi que la programmation LISP, qui a fait ses preuves par ailleurs, n'est pas le pire terrain pour faire prendre racine à des processus musicaux...

Références

[Bidlack 1992] Bidlack, Rick, "Chaotic Systems as Simple (but Complex) Compositional Algorithms", *Computer Music Journal*, 16(3):33-47.

[Bobrow *et al*. 1990] Bobrow, Daniel G., Demichiel, Linda G., Gabriel, Richard P., Keene, Sonya E., Kiczales, Gregor, et Moon, David A., "Common LISP Object System" *in* [Steele 1990 : 770 *sq*.].

[Cointe et Rodet 1984] Cointe, Pierre, et Rodet, Xavier, "FORMES: An Object and Time Oriented System for Music Composition and Synthesis", *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, Austin.

[Letz *et al*. 1992] Letz, S., Merlier, B., et Orlarey, Y., *CLCE, version 4b*, Lyon, GRAME.

[Lorrain 1980] *Une panoplie de canons stochastiques*, Paris, Centre Georges Pompidou (Rapport IRCAM 30/80). Aussi disponible : "A Panoply of Stochastic 'Cannons'", *Computer Music Journal*, 4(1):53-81, et *in* Curtis Roads, éd., *The Music Machine: Selected Readings from Computer Music Journal*, Cambridge, MIT Press : 351-379.

[MAX 1995] *MAX Reference*, Palo Alto, Opcode Systems.

[MCL2.0 1992] *Macintosh Common LISP*, Apple Computer.

[MCL3.0 1995] *Macintosh Common LISP*, Apple Computer.

[MidiShare 1994] *MidiShare Developer Documentation, version 1.68*, Lyon, GRAME.

[Puckette 1986] Puckette, Miller, développements originaux de MAX à l'IRCAM, Paris, entre 1986 et 1989.

[Steele 1990] Steele, Guy L. Jr., *Common LISP: The Language*, Digital Press.

[Vercoe 1993] Vercoe, Barry, *CSOUND: A Manual for the Audio Processing Programs with Tutorials*, Cambridge, MIT, Media Lab.