UNE NOUVELLE APPROCHE DES OBJETS GRAPHIQUES ET INTERFACES UTILISATEURS DANS PURE DATA

Pierre Guillot
CICM - EA1572,
Université
Paris 8, MSH Paris Nord,
Labex Arts H2H
Guillotpierre6@gmail.com

RÉSUMÉ

Cet article présente le CICM Wrapper, une interface de programmation facilitant la création d'objets graphiques et d'interfaces utilisateurs pour le logiciel Pure Data ainsi que la bibliothèque d'objets qui en découle. Nous revenons sur les problèmes de programmation liés aux mises en œuvre d'interfaces graphiques utilisateurs rencontrés dans Pure Data afin de préciser les choix qui ont conduit à la création d'une interface de programmation ainsi que les solutions que nous proposons. Enfin nous présentons en détail les objets de la bibliothèque et leurs nouvelles fonctionnalités pour mettre en évidence les améliorations offertes par notre approche.

1. INTRODUCTION

Depuis 2012, le CICM[1] a développé une bibliothèque de mise en espace du son grâce aux techniques ambisoniques dans le cadre des projets du Labex Arts H2H de l'Université Paris 8[2]. Ces projets, orientés vers une approche artistique, ont pour objectif d'explorer les possibilités musicales offertes par ces techniques et de rendre accessible aux musiciens et compositeurs les concepts acoustiques et mathématiques sous-jacents pour faire émerger de nouvelles applications musicales. La bibliothèque HOA[3], qui en résulte, offre un ensemble de classes C++ desquelles découlent des mises en œuvre pour des logiciels d'édition musicale et de synthèse sonore largement usités par les musiciens, notamment sous forme d'objets pour les logiciels Max[4] et Pure Data[5] [1][2].

L'un des éléments essentiels de ces mises en œuvre a été la création d'interfaces graphiques utilisateurs permettant de répondre aux enjeux didactiques de la bibliothèque et de faciliter la prise en main des opérations dans le domaine des harmoniques circulaires[6]. Grâce à l'objet hoa.scope~, par exemple, il est possible de visualiser graphiquement les harmoniques circulaires et ainsi de faciliter la compréhension du modèle ambisonique (Figure 1). Ces interfaces sont relativement complexes à élaborer de par la grande variété des interactions proposées à laquelle se rajoutent des représentations qu'il n'est aisément possible de réaliser qu'en utilisant des systèmes de calques[7] et des opérations comme des rotations et translations matricielles. En résumé, la création de ces interfaces nécessite que les plateformes logicielles auxquelles elles sont destinées disposent au sein de leurs kits de développement logiciel de nombreux outils dédiés à la mise en œuvre d'interfaces graphiques utilisateurs.

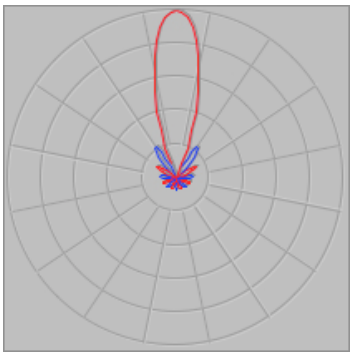

Figure 1. L'objet hoa.scope~ représente le champ sonore sous la forme d'une somme pondérée d'harmoniques circulaire.

Alors que la mise en œuvre des interfaces pour le logiciel Max a été relativement aisée à réaliser, grâce à un SDK[8] très bien adapté aux besoins de la bibliothèque HOA - tant pour la mise en place de traitements pour le multicanal que pour la création d'interfaces graphiques, de nombreuses complications sont apparues lors des mises en œuvre pour le logiciel Pure Data. Ces difficultés nécessitent de repenser la structure et le fonctionnement général des objets graphiques et des objets de traitement du signal dans Pure Data [3]. Notons néanmoins qu'il était déjà possible d'utiliser certains traitements de la bibliothèque HOA dans Pure Data grâce à la version FAUST qui permet de compiler des

[1] Centre de recherche Informatique et Création Musicale.

[2] Les projets sont "*La spatialisation du son pour le musicien, par le musicien*" et "*Des interfaces pour la mise en espace du son*" et "*HOA, 3D*".

[3] http://www.mshparisnord.fr/hoalibrary/.

[4] http://cycling74.com/.

[5] http://msp.ucsd.edu/software.html/.

[6] Les harmoniques circulaires sont des fonctions périodiques utilisées en ambisonie pour représenter l'espace.

[7] Système de calques que nous pouvons trouver dans les logiciels de dessin.

[8] *Software development kit* ou kit de développement logiciel.

traitements pour de multiples plateformes logicielles, cependant cette version est plus adaptée à une utilisation fixe qu'aux expérimentations[9] que nous souhaitons réaliser avec la version HOA spécifique à Pure Data. Afin de répondre à ces contraintes et de faciliter la mise en œuvre des objets graphiques, nous avons réalisé une bibliothèque en C et Tcl/Tk[10] sous la forme d'une API[11] : le CICM Wrapper[12]. Cet ensemble de codes permet de proposer de nouvelles fonctionnalités et de faciliter la mise en œuvre d'objets. Loin d'être restreinte à notre contexte d'interfaces pour la spatialisation ambisonique et à nos problématiques initiales, ce travail s'est ouvert à de nombreuses utilisations et a permis notamment de réaliser un ensemble d'interfaces graphiques dans Pure Data au fort potentiel ergonomique.

Dans cet article, nous revenons, dans un premier temps, sur les problèmes que nous avons rencontrés lors de la mise en œuvre d'objets dans Pure Data et présentons par la suite les solutions que nous proposons via le CICM Wrapper. Il s'agit de permettre aux utilisateurs de comprendre les modifications opérées par notre approche et les changements par rapports aux précédentes interfaces graphiques. Nous espérons ainsi offrir aux développeurs la possibilité de prendre en main cette API, afin qu'ils puissent offrir de nouveaux outils qui nous l'espérons répondront aux besoins de la communauté d'utilisateurs de Pure Data [4].

2. LES DIFFICULTES DE PROGRAMMATION D'OBJETS EXTERNES

Lors de la mise en œuvre des interfaces graphiques utilisateurs dans Pure Data, nous avons rencontré de nombreuses difficultés. Il nous est impossible d'énumérer l'ensemble des fonctionnalités et approches qui selon nous posent problème; de ce fait nous présentons celles qui nous semblent les plus contraignantes.

Tcl et Tk sont dans Pure Data le langage et les outils qui gèrent les interfaces graphiques et les interactions avec l'utilisateur par envoi d'instructions sous forme de scripts. Ainsi, ces scripts doivent être réécrits dans chaque objet pour chaque nouvelle instruction, comme le dessin d'un rectangle ou la récupération de la position du curseur de la souris. Étant un langage interprété, cette pratique est très sujette aux erreurs de programmation car celles-ci ne sont pas révélées lors de la compilation des objets. De plus, la syntaxe expansive peut très rapidement encombrer les codes et rendre difficile leur lecture. Bien qu'il nous paraisse préférable de ne pas se servir de ce langage au sein du code d'un objet, son utilisation semble inévitable, du moins dans une première approche.

L'un des éléments gênants de ce langage dans notre contexte est le système de *tag*. Il est, en effet, possible d'attribuer des noms de référence aux éléments dessinés permettant par la suite de pouvoir les modifier ou les supprimer, opérations nécessaires dans toute mise en œuvre. Ainsi, cette propriété du langage se rapproche des systèmes de calques, que nous avons suggéré précédemment. Cependant, la formulation n'est ici pas des plus adaptées. Il est souvent nécessaire de définir des noms spécifiques à certains éléments, des noms généraux à des groupes d'éléments et un nom global à l'ensemble des éléments d'un même objet ce qui complexifie le code. Notons encore qu'étant donné que ces noms ont un champ d'action pour l'ensemble d'un *canvas*, il faut que les noms contiennent un élément unique relatif à chaque instance d'un objet afin que les changements ne s'appliquent qu'à un objet spécifique et non à l'ensemble des objets d'une même classe qui se trouvent dans ce *canvas*.

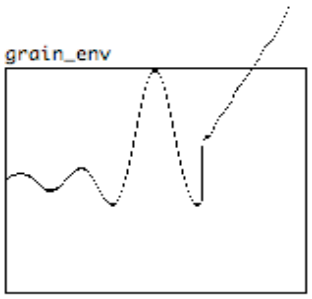

Figure 2. Exemple concret où le dessin peut sortir des limites d'un objet dans Pure Data avec l'objet array.

Une autre contrainte rencontrée réside dans la particularité de Pure Data de devoir dessiner directement dans le *canvas*. Bien que le programmeur définisse des limites à son objet (permettant notamment d'être notifié lorsque l'utilisateur clique ou se déplace dans l'objet), il lui est possible de dessiner en dehors de celles-ci (Figure 2). Ce problème n'est que minime pour des interfaces graphiques simples tels qu'un objet bang ou un objet toggle mais dès lors que l'interface se complexifie, il peut devenir nécessaire de tronquer les figures dessinées dans les limites de l'objet.

Dans l'objet hoa.map, par exemple, il est possible de déplacer des cercles, représentant des sources sonores, au delà des bordures de l'objet parce que l'interface ne représente qu'une partie focalisée de l'espace d'interaction. Dans un tel cas, il est nécessaire dans le code de vérifier si les cercles sont dans les limites ou non de la boîte afin de les dessiner ou de les effacer (Figure 3). Cette opération se complexifie lorsque le cercle chevauche les limites où l'utilisateur est en attente d'un

[9] Le langage FAUST ne permet pas de mettre en œuvre des objets avec un nombre de canaux dynamiques ou des interfaces graphiques, il est ainsi peu adapté à une phase d'expérimentation dans ce contexte de l'ambisonie bien qu'offrant des avantages certains tant sur le plan du déploiement multi-plateforme que sur celui des optimisations.

[10] Tcl (Tool Command Language) est un langage de programmation et Tk est une bibliothèque logicielle pour la création d'interfaces graphiques, http://www.tcl.tk/.

[11] *Application programming interface* ou interface de programmation

[12] Le CICM Wrapper est disponible sur le répertoire Git du CICM, https://github.com/CICM/CicmWrapper.

cercle partiellement caché ; or cette opération est complexe à mettre en œuvre, nécessite de nombreux calculs et amène immanquablement un manque de clarté dans le code.

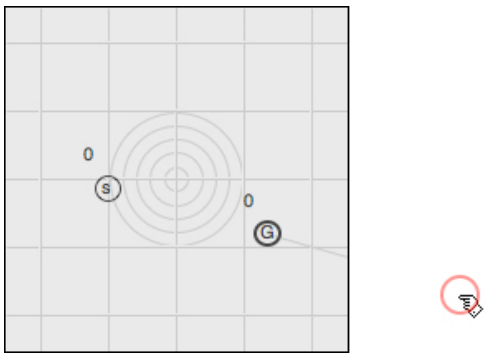

Figure 3. L'objet hoa.map où une source sonore, représentée par le cercle rouge, est déplacée en dehors des limites de l'objet.

Enfin une dernière caractéristique dérangeante, liée au dessin dans Pure Data, est le fait que le programmeur doive lui même dessiner le contour de sa boîte ainsi que les entrées et les sorties alors que les *canvas* définissent à quel endroit l'utilisateur doit cliquer afin de créer des lignes et de raccorder les objets. En somme, il revient au programmeur de redéfinir cette formule, qui répond en grande partie à de fins réajustements, permettant de calculer l'emplacement des entrées et sorties afin de les dessiner aux endroits qu'il convient. Opération redondante qui, là encore, complexifie le code.

Sur le plan des interactions, l'API de Pure Data est encore ici relativement restreignant. Les *canvas* sont rattachés via Tcl/Tk à un certain nombre d'événements desquels ils reçoivent des informations, comme les coordonnées de la souris lors de son déplacement ou les touches du clavier activées, qu'ils renvoient par la suite aux objets lorsque ceux-ci possèdent les méthodes adéquates. L'architecture peut sembler tout à fait valable mais le problème réside dans le manque de variété des événements auxquels un objet peut être rattaché. Ceux-ci sont seulement au nombre de deux pour les événements liés à la souris, son déplacement et le clic de la souris potentiellement associés à deux touches : *Alt* et *Shift*. Dans l'objet hoa.map, le comportement de l'objet diffère non seulement selon la combinaison des touches mais offre, de plus, la possibilité de faire apparaître un menu contextuel lorsque l'utilisateur effectue un clic droit sur l'interface ; or cet événement est exclusivement rattaché au menu contextuel natif de Pure Data (permettant d'accéder à l'aide ou aux propriétés de l'objet) ainsi, il est impossible d'en être notifié et de modifier ce comportement. Aussi, il nous a très rapidement semblé que la variété des événements auxquels nous avons accès dans Pure Data est trop limitée en comparaison de l'ensemble des événements usuellement offerts dans les interfaces utilisateurs [5].

La création d'une fenêtre de propriétés nous a semblé aussi être un problème récurrent. Il est, en effet, nécessaire de réaliser une nouvelle interface en Tcl/Tk pour chaque objet affichant les différentes valeurs des propriétés telles que la taille de l'objet, la police ou encore les couleurs. A cela, il faut ajouter une série de méthodes permettant de recevoir et envoyer ces valeurs afin de communiquer entre l'objet et la fenêtre de propriétés. De plus, les fenêtres de propriétés ne diffèrent d'un objet à un autre que par leur nombre, mais les types de variables (données numériques ou données textuelles) et leurs représentation (zone de texte, sélecteur de couleur, boîte nombre, etc.) se retrouvent d'un objet à un autre. Ainsi, cette opération est, encore ici, redondante et ne fait que diminuer la clarté du code. Il nous aurait donc semblé préférable de pouvoir générer automatiquement cette fenêtre de propriété, sans avoir nécessairement à réécrire l'ensemble du script, et d'accéder et modifier ces valeurs exclusivement à partir du code source C de l'objet. Notons encore que ces propriétés sont, dans les usages, des variables que l'utilisateur souhaite sauvegarder et retrouver à la même valeur à l'ouverture d'un patch. Là encore, il revient au programmeur de réaliser la sauvegarde des propriétés à l'enregistrement du patch et leur récupération et initialisation à l'ouverture. En somme, le programmeur peut, ici aussi, souhaiter faire appel à des fonctions usuelles, tel que proposées dans le système d'attributs présent dans le SDK du logiciel Max.

De manière plus générale, un ensemble de fonctions ou d'outils usuels, pouvant grandement faciliter la création d'objets dans Pure Data, nous semblait manquer [4]. Le présent contexte ne permet pas, comme nous l'avons dit, de tous les énumérer mais nous nous intéressons plus particulièrement à deux d'entre eux. D'une part la gestion des fonctions du traitement du signal qui devient complexe à mettre en œuvre dès lors que l'on souhaite un nombre d'entrées et de sorties dynamique ou lorsque les vecteurs d'entrées et sorties doivent être différents (afin de réaliser des traitements du signal *out-of-place*) et d'autre part, un système de type *proxy* pour les entrées permettant, par exemple, d'avoir plusieurs entrées pouvant recevoir tous types de données mais offrant la possibilité d'obtenir l'index de l'entrée par laquelle chaque message arrive.

3. RESOLUTION DES PROBLEMES

Notre travail a été orienté par certaines nécessités. Cette bibliothèque se doit de fonctionner sur l'ensemble des systèmes d'exploitations auxquels est destiné Pure Data notamment Linux, Mac OS et Windows, c'est pourquoi notre choix s'est très rapidement porté sur l'utilisation seule des dépendances internes à Pure Data. Ainsi, il nous a paru préférable, malgré les problèmes rencontrés, de proposer une version utilisant uniquement Tcl/Tk. Il s'agissait aussi de ne pas limiter les possibilités de programmation donc de laisser ouverte l'utilisation des fonctions natives et aussi de pouvoir être

facilement intégré à Pure Data ou Pure Data Extended. Enfin, nous avons tenté de nous rapprocher au maximum de l'API du logiciel Max afin de faciliter par la suite le passage d'un objet Max au logiciel Pure Data et inversement.

La première approche employée utilisait la structure d'objet graphique aujourd'hui généralement répandue et qui se trouve dans les objets graphiques natifs de Pure Data et de tenter par la suite de contourner et résoudre les problèmes. Cependant, cette approche est une impasse, il est, en effet, impossible d'être notifié de toutes les interactions réalisées sur l'objet et rogner les dessins dans les objets [6] reste très compliqué à réaliser. De plus, cette approche engendre le développement d'un code source très complexe, rempli d'exceptions impliquant de nombreux problèmes, notamment sur la potentielle réutilisation et relecture du code.

La solution retenue est apparue en étudiant les codes sources des objets tels que entry ou popup[13]. Bien que leurs mises en œuvre ne semblent pas optimales au premier abord - de nombreuses difficultés compliquent l'utilisation de ces objets - l'approche qui consiste à créer des *widgets* via Tcl/Tk dans le patch, comme un menu contextuel ou champ de saisie textuel, nous a semblé pouvoir offrir une solution à nos problèmes notamment par le fait qu'il est possible d'être rattaché à l'ensemble des événements utilisateurs sans passer par le *canvas*. Notre solution consiste donc à créer un *widget* de type fenêtre dans lequel nous créons une nouvelle instance d'un *canvas*, cet outil peut alors être vu comme une fenêtre au contour invisible possédant son propre *canvas* et incrusté dans le *canvas* initial. Ce procédé offre deux principaux avantages, d'une part, il est possible de rattacher le *canvas* créé à l'ensemble des événements proposés dans Tcl/Tk afin d'en être notifié et d'autre part, de ne plus se soucier de dessiner hors des limites étant donné que la fenêtre dans laquelle nous nous plaçons tronque automatiquement le dessin à l'intérieur de ses limites. Néanmoins cette approche implique que le *canvas* initial ne soit plus notifié des événements de utilisateur lorsque le focus est sur notre objet. Ainsi cela amène l'inconvénient de devoir notifier en retour le *canvas* initial d'un grand nombre d'événements comme la sélection de notre objet, son déplacement, le raccordement de ses entrées ou sorties, méthodes que nous avons dû implémenter.

Afin de généraliser cette approche et de pouvoir accéder facilement à ce système dans de multiples mises en œuvre, nous avons fait le choix de développer une API (Figure 4). Cette interface de programmation comprend ainsi un certain nombre de structures et de fonctions sur lesquelles nous revenons brièvement afin d'en comprendre non pas le fonctionnement, car le présent document ne pourrait suffire, mais du moins son utilisation.

[13]L'objet *entry* et *popup* font parti de la bibliothèque *flatgui* développé par Ben Bogart pour Pure Data Extented.

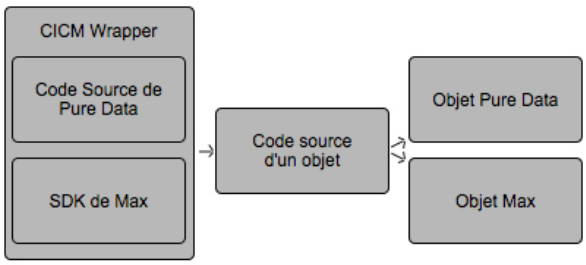

Figure 4. Représentation du CICM Wrapper dans le processus de création d'un objet Pure Data ou Max.

Il a fallu étendre la structure de *classe* d'objet native de Pure Data, structure définissant un type d'objet, afin d'offrir les outils nécessaires à nos attentes. A l'instanciation d'une nouvelle classe d'objet, qui se réalise de manière très similaire à ce que nous pouvons trouver dans l'API de Max, nous définissons d'une part si l'objet est de type graphique et/ou traitement du signal afin d'initialiser la structure des objets selon leurs fonctionnalité et d'autre part les méthodes de l'objet, nous permettant alors d'initialiser un certains nombre de fonctions enveloppant le fonctionnement interne de l'objet pour ne laisser transparaître au programmeur que ce qui nous semble nécessaire.

```
CLASS_ATTR_RGBA                 (c, "bgcolor", 0, t_bang, f_color_background);
CLASS_ATTR_LABEL                (c, "bgcolor", 0, "Background Color");
CLASS_ATTR_ORDER                (c, "bgcolor", 0, "1");
CLASS_ATTR_DEFAULT_SAVE_PAINT   (c, "bgcolor", 0, "0.75 0.75 0.75 1.");
CLASS_ATTR_STYLE                (c, "bgcolor", 0, "color");
```

Figure 5 Portion du code de l'objet c.bang présentant la syntaxe du CICM Wrapper, similaire à celle utilisée dans MAX, pour créer des attributs.

Il est aussi possible au programmeur de créer des attributs pour cette classe qui seront dès lors sauvegardés et initialisés de façon automatique à la création et à la sauvegarde de chaque instance de l'objet. La création de ces attributs se fait via un système de macro-définitions[14], ici aussi très similaires à celles proposées dans l'API de Max permettant de définir un style aux attributs (texte, nombre, menu, couleur, etc.), des valeurs par défaut, une étiquette de présentation, et d'autres caractéristiques qui apparaîtront dans la fenêtre de propriété (Figure 5).

Le fonctionnement interne de l'objet est, quant à lui, relativement opaque au programmeur mais les méthodes à mettre en place sont sensiblement plus simples à réaliser. Les deux points les plus problématiques dans la création d'un objet, le dessin et les interactions, sont nettement plus simples à mettre en place. Le dessin dans un objet se fait en créant une méthode répondant au message *paint*. Dans cette méthode, il est possible de créer des calques répondant à un nom et dans ces calques il est possible de dessiner toutes formes de figures. Afin de créer des dessins, nous offrons des fonctions qui

[14]Système permettant de remplacer un indicateur, sous forme de texte, par un texte.

s'occupent de mettre en forme et d'envoyer le script Tcl/Tk et qui gèrent notamment le système de *tag* évitant ainsi toute erreur de syntaxe ; notons néanmoins qu'il est toujours possible de passer outre ce système si besoin[15]. En plus des fonctions de dessin usuelles telles que la création de forme géométriques simples ou l'écriture de texte, nous offrons un système de matrices permettant de réaliser des rotations et des translations. Il est aussi possible de définir les couleurs à utiliser dans les formats RGB, HSL ou en hexadécimal, offrant plus de flexibilité. Cette méthode de dessin est appelée soit de manière opaque à chaque fois qu'il est nécessaire de redessiner l'objet, lorsque l'objet bouge ou qu'un attribut lié au graphisme est changé par exemple, soit de manière transparente dans le code de l'objet. Il est ainsi possible au programmeur, selon le contexte, d'invalider un calque spécifique ; seulement ce calque-là sera alors redessiné à l'appel de la méthode *paint* afin d'alléger le processus (Figure 6). Enfin le dessin des bordures, des entrées et sorties est réalisé de façon automatique.

```
// Fonction qui dessine l'arrière plan de l'objet
void draw_background(t_bang *x, t_object *view, t_rect *rect)
{
    // Le rayon du cercle inscrit de l'objet
    float radius = rect->width * 0.5;

    // Creation d'un nouveau calque défini par le symbol
       "background_layer"
    t_elayer *g = ebox_start_layer((t_ebox *)x,
       gensym("background_layer"), rect->width, rect->height);
    // t_jgraphics *g = jbox_start_layer((t_object *)x, view,
       gensym("background_layer"), rect->width, rect->height);

    // Si le calque est nouveau ou a été invalidé, il est possible de
       le redéfinir
    if (g)
    {
        // Definition de la couleur utilisée
        egraphics_set_color_rgba(g, &x->f_color_background);
        // jgraphics_set_source_jrgba(g, &x->f_color_background);

        // Création d'un cercle
        egraphics_arc(g, radius, radius, radius * 0.9, 0, EPD_2PI);
        // jgraphics_arc(g, radius, radius, radius * 0.9, 0, EPD_2PI);

        // Ajout du cercle dans le calque en remplissant
        egraphics_fill(g);
        // jgraphics_fill(g);

        ebox_end_layer((t_ebox*)x, gensym("background_layer"));
        // jbox_end_layer((t_jbox*)x, view,
           gensym("background_layer"));
    }

    // Dessin du calque
    ebox_paint_layer((t_ebox *)x, gensym("background_layer"), 0., 0.);
    // jbox_paint_layer((t_jbox *)x, view,
       gensym("background_layer"), 0., 0.);
}

// Fonction appelée au click de la souris
void mouse_down(t_bang *x, t_object *view, t_pt pt, long modifiers)
{
    // Invalidation du calque de l'arrière plan de l'objet
    ebox_invalidate_layer((t_ebox *)x, gensym("background_layer"));
    //jbox_invalidate_layer((t_object *)x, NULL,
       gensym("background_layer"));

    // Notification à l'objet de redessiner
    ebox_redraw((t_ebox *)x);
    // jbox_redraw((t_jbox *)x);

    outlet_bang(x->f_out);
}
```

Figure 6. Portion du code de l'objet c.bang présentant la syntaxe utilisée dans le CICM Wrapper pour dessiner et gérer le système de calques. Les fonctions commentées utilisent la syntaxe de Max qu'il est aussi possible d'utiliser.

[15]Nous invitons le lecteur à regarder le code de l'objet c.blackboard afin d'en avoir un exemple.

Les interactions de l'utilisateur sont à présent très variées et font appel à plusieurs méthodes selon les actions : entrer dans l'objet, sortir de l'objet, survoler l'objet, cliquer, « traîner », relâcher, utiliser la molette et double cliquer, presser une touche clavier quelconque, presser une touche clavier « spécifique » (Tab, Supprimer, Espace, etc.). Ces méthodes relatives aux événements de la souris transmettent la position de la souris relative à l'objet mais aussi les clefs Shift, Maj, Ctrl et Atl. En somme, il est à présent possible de réaliser un objet au fonctionnement élaboré et aux dessins complexes sans pour autant rencontrer de grandes difficultés de programmation.

Notons encore que les objets offrent à présent un système de *proxy* pour les entrées très facile à mettre en œuvre ainsi qu'un nouveau formatage des fonctions *DSP* et des fonctions *perfom* similaires à celles offertes dans l'API de Max. A cela se rajoutent de nombreuses petites améliorations dont, entre autres, des méthodes facilitant la lecture et l'écriture de fichiers externes, la mise en place d'un système de préréglages, une fonction renvoyant la position globale de la souris ou relative au *canvas* ou encore un système de notifications[16].

4. LA MISE EN ŒUVRE : PRESENTATION DES OBJETS

Notre bibliothèque, le CICM Wrapper a, par la suite, permis de réaliser un certain nombre d'interfaces graphiques fonctionnant tout aussi bien sur Pure Data Vanilla que sur Pure Data Extended pour les systèmes d'exploitation Mac Os, Linux et Windows et ayant pour but d'améliorer l'ergonomie générale en proposant de nouvelles interactions et en offrant des représentations plus raffinées[17]. Cette série d'objets reprend les outils standards déjà présents soit dans Pure Data, Vanilla et Extented, soit dans Max. Avant d'énumérer l'ensemble des fonctionnalités offertes, nous pouvons mettre en évidence certaines caractéristiques générales à l'ensemble des ces outils.

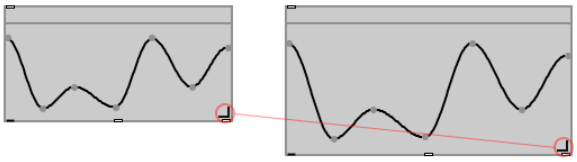

Figure 7. Redimensionnement de l'objet c.breakpoints en déplaçant le coin en bas à droite.

Les objets graphiques possèdent tous la particularité de pouvoir être redimensionnés en tirant les bordures bas ou droite ou le coin bas-droite (Figure 7). Notons que selon les objets, la hauteur et la largeur peuvent être

[16]Fonctions auxquelles le programmeur pourra se référer dans la documentation et les exemples disponibles avec la distribution du projet.

[17]La bibliothèque est disponible sur le répertoire Git du projet CICM Wrapper, https://github.com/CICM/CicmWrapper/releases.

fixées arbitrairement ou dépendre de la taille de la police tel que définie dans l'objet c.number. Afin d'offrir des patchs plus lisibles, les entrées et sorties des objets ne sont visibles qu'en mode édition, nous avons aussi fait le choix de leurs attribuer les couleurs syntaxiques par défaut de Pure Data Extented permettant de facilement repérer le type de messages qu'ils peuvent recevoir ou envoyer.

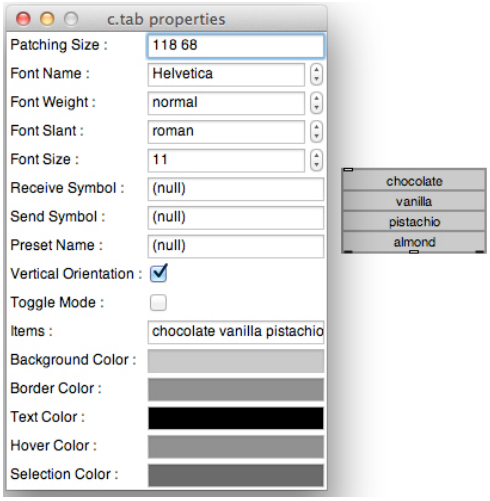

Figure 8. Fenêtre de la fenêtre de propriété de l'objet c.tab.

Tous les objets possèdent des propriétés communes (ou attributs) : la taille de l'objet (hauteur et largeur), le nom, le poids, l'inclinaison et la taille de la police utilisée par l'objet, la couleur d'arrière plan et la couleur de la bordure, auxquelles viennent se rajouter les propriétés spécifiques de chaque objet. Enfin, ils possèdent aussi un symbole d'envoi et un symbole de réception, de manière analogue aux interfaces natives de Pure Data (Figure 8). Ces propriétés peuvent, bien sûr, être modifiées de manière dynamique par des messages envoyés à l'objet et sont aussi accessibles via la fenêtre de propriétés qui facilite leur édition en offrant un champ de saisie textuel ou numérique, avec un système permettant d'incrémenter ou décrémenter des valeurs numériques ou l'index d'un menu, un bouton de type interrupteur ou encore un sélecteur de couleur selon le type de valeurs attendues. Notons enfin que l'ensemble des caractéristiques des propriétés des objets peut être affiché dans la console en envoyant le message "attrprint" à l'objet, permettant de connaître les messages attendus par l'objet pour définir ses propriétés.

Figure 9. Représentation de l'interaction permettant d'incrémenter la valeur de l'objet c.number.

Certains objets sont donc très similaires à leurs homologues natifs de Pure Data. C'est le cas des objets c.bang et c.toggle, qui hormis les améliorations générales à l'ensemble des objets, offrent des fonctionnements identiques au bang et toggle. D'autres proposent de très légères améliorations comme les objets c.slider qui deviennent verticaux ou horizontaux selon que la hauteur soit plus grande que la largeur, l'objet c.number qui permet d'incrémenter des valeurs numériques avec une précision qui dépend de la position de la souris lorsque l'utilisateur clique sur l'objet (Figure 9), l'objet c.number~ qui consiste en une combinaison de l'objet boite nombre et d'un snapshot~, l'objet c.meter~ qui, de manière identique, peut être comparé à une combinaison de l'objet vu et de l'objet pvu~ ou encore l'objet c.radio qui offre en plus, par rapport à l'objet radio, un mode de type *check-list*. D'autres interfaces sont des adaptations de certaines bibliothèques présentes dans Pure Data Extended mais qui, suite à des mises à jours ou du fait de la complexité de leur création, sont difficiles à utiliser, possèdent certains dysfonctionnements ou n'offrent pas certaines fonctionnalités essentielles. C'est le cas des objets c.scope~, un oscilloscope uni ou bidimensionnel, l'objet c.colorpanel, un sélecteur de couleur sous forme de matrice à taille variable, l'objet c.blackbloard, un tableau permettant de dessiner à la souris et par script, d'écrire du texte et d'afficher des images au format GIF, l'objet c.knob, un bouton rotatif avec un mode circulaire et un mode sans fin ou encore l'objet c.menu, qui permet d'afficher un menu déroulant. Enfin certains objets possèdent seulement leurs équivalents dans le logiciel Max et nous semblent des plus pratiques : c.function qui permet de créer une fonction par suite de points avec différents modes d'interpolations, c.incdec qui permet d'incrémenter ou décrémenter une boite nombre sans *stack overflow*, l'objet c.plane qui permet de déplacer un point sur un plan bidimensionnel, l'objet c.tab qui offre une séries de boutons à texte, l'objet c.gain~ qui permet de contrôler le volume sonore avec une glissière à curseur, l'objet c.rslider qui permet de définir une plage grâce à une glissière à double curseur et enfin l'objet c.spectrum~ qui permet d'afficher le spectre d'un signal sonore. Notons néanmoins que l'ensemble de ces objets a été réalisé afin de répondre à des attentes personnelles, ainsi leur fonctionnement peut légèrement varier vis à vis de leurs homologues natifs de Pure Data ou de Max.

Enfin, nous offrons un objet c.preset. L'ensemble des objets possédant une méthode *preset*, possède un attribut permettant de leurs attribuer un nom qui sert de référence à l'objet afin d'enregistrer l'état actuel de l'objet et revenir à cet état. Ce système est, en somme, très similaire à celui proposé dans le logiciel Max à la différence notable qu'il est possible de réaliser des interpolations. Ce système vise, en outre, à être complété par une série d'interfaces permettant d'éditer les préréglages de manière plus simple tel que le système de *pattrstorage* de Max mais, ici aussi, ayant accès à l'ensemble des méthodes et fonctionnalités des objets, nous pouvons envisager un système différent répondant plus à nos attentes.

5. CONCLUSION

Lors de la mise en œuvre de la bibliothèque HOA pour Pure data, nous avons réalisé une API afin de répondre aux nombreux problèmes rencontrés concernant la création d'objet graphiques et de traitements multicanal. Cette bibliothèque peut néanmoins être utilisée dans un large contexte et ouvre de nouvelles perspectives quant à l'ergonomie du logiciel. Elle possède de plus l'avantage, d'être libre, gratuite et de n'avoir aucune dépendance autre que Pure Data. Cela permet, en outre, d'être totalement indépendant des différentes distributions du logiciel.

Ainsi, nous pouvons envisager que son utilisation perdure avec les évolutions du logiciel et soit poursuivie au sein de la communauté. Nous espérons également voir émerger de nouvelles mises en œuvre conçues en dehors notre équipe de recherche, car suite à nos premières publications des objets, nous avons réalisé qu'il existe une forte demande d'outils stables et ergonomiques pour Pure Data.

Nous pouvons aussi envisager de nombreuses améliorations afin de faciliter la mise en œuvre d'objets. Nous pensons notamment mettre en place un système permettant de s'attacher aux notifications d'autres objets et une méthode de commentaires sur les entrées et sorties. Nous envisageons aussi la possibilité de choisir d'autres API graphiques selon les besoins. Nous pensons notamment à Juce[18] qui permettrait d'avoir un rendu plus homogène entre les systèmes d'exploitation et d'avoir accès à des méthodes tels que le *Drag & Drop* de fichiers ou le rendu tridimensionnel via OpenGl.

Enfin, suite aux nombreux retours qui nous parvenus des utilisateurs et programmeurs, nous envisageons avec les créateurs et les responsables la possibilité d'intégrer la bibliothèque à la distribution de Pure Data Extended et de PD-L2ork, dans laquelle nous pourrions tirer partie de l'utilisation des SVG.

6. REFERENCES

[1] Colafrancesco, J., Guillot P., Paris E., Sedes A., Bonardi 1. « La bibliothèque HOA, bilan et perspectives », *Actes des Journées d'informatique musicale*, St-Denis, France, p. 188-197, 2013.

[2] Guillot P., Paris E., Deneu M. « La bibliothèque de spatialisation HOA pour MaxMSP, Pure Data, VST, Faust, … », *Revue Francophone d'*Informatique *Musicale*, St-Denis, France, 2013.

[3] Zmölnig J. M. « How to Write an External for Pure Data », Institute of Electronic Music and Acoustics, Graz, Autriche, http://pdstatic.iem.at/externals-HOWTO/.

[4] Miller P. « Pure Data : Another Integrated Computer Music Environment », *Proceedings Second Intercollege Computer Music Concerts*, Tachikawa, Japon, 1996.

[5] Todoroff T. « Control of Digital Audio Effects », *DAFX – Digital Audio Effects*, J. Wiley & Son, 2002.

[6] Foley, J. D., Paris E., Van Dam A., Feiner S. K., Hughes J ; F. « Computer Graphics : Principles and Practice in C, 2nd Edition ». *Addison-Wesley*, 1995.

[18] http://www.juce.com/.