Traitement des messages TuioClient dans Pure Data

Il est possible de traiter directement les messages issus du **TuioClient**, en les filtrant par des objets **route** et en récupérant les données par des objets **unpack**.

La séquence normale de messages est la suivante :

1. Lorsqu'un nouvel objet / contact est appliqué sur la surface tangible, **TuioClient** délivre un message de type « add » (addObject ou addCursor)
2. Puis, lorsque cet objet / contact est déplacé sur la surface, **TuioClient** délivre un ou plusieurs messages « update » (updateObject ou updateCursor)
3. enfin, lorsque cet objet / contact est retiré de la surface, **TuioClient** délivre un message « remove » (removeObject ou removeCursor)

Voyons les informations contenues dans les messages addCursor, updateCursor et removeCursor, ainsi que la manière de les récupérer dans Pure Data :

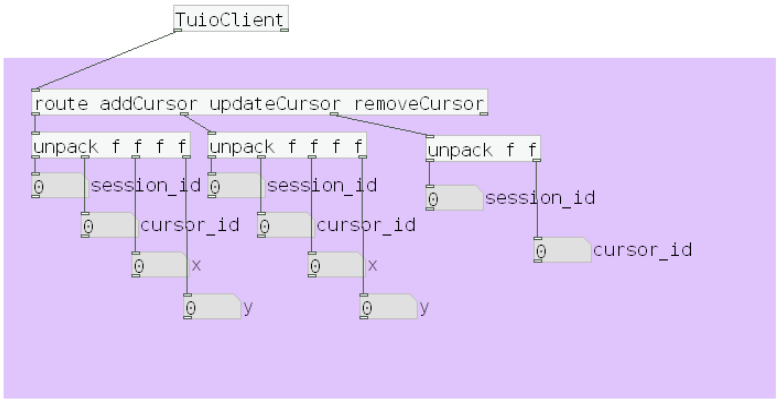

fig. 1 : structure des message xxxCursor

Les messages addObject, updateObject et removeObject diffèrent peu, si ce n'est qu'une information supplémentaire sur la valeur de l'angle (orientation de l'objet) est disponible.

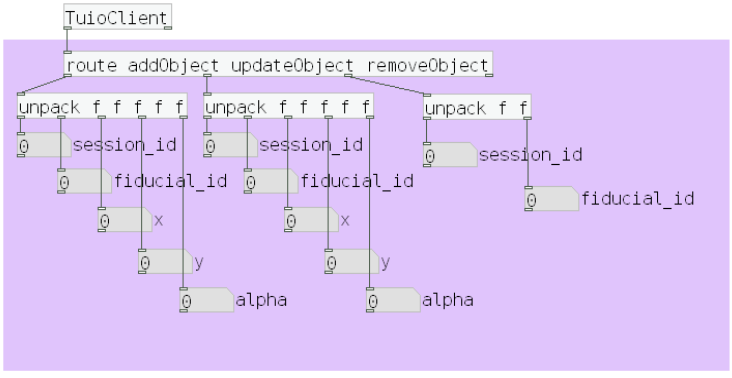

fig. 2 : structure des messages xxxObject

La plupart du temps, le fait de connaître la position (x/y) et l'orientation d'un objet suffisent pour faire déjà pas mal de choses.
Cependant, les messages updateCursor / updateObject fournissent des informations supplémentaires sur la variation dans le temps (dérivées première et seconde) de ces paramètres :

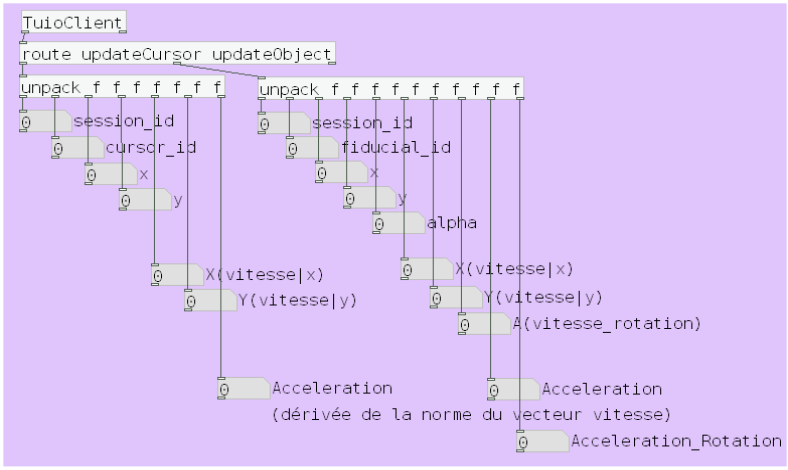

fig. 3 : structure complète des messages updateXXXX

Dans cette première partie, nous allons plutôt nous intéresser aux messages add / update / removeObject, fournis par ReacTiVision. Quant à la gestion des objets add / update / removeCursor, caractéristiques d'une configuration multitouch (telle que CCV), nous l'aborderons plutôt dans la seconde partie de ce tutoriel.

Application n°1

On souhaite commander avec le fiducial no.1 la coloration d'un rectangle défini par la structure GEM ci-dessous.

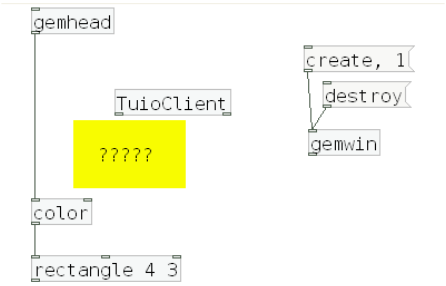

fig. 4 : comment relier les objets TuioClient et color ?

On souhaite réaliser la correspondance suivante entre les coordonnées du fiducial et la coloration :
x -----> saturation
y -----> luminosité
angle -> teinte

L'objet **color** attend des paramètres (rouge vert bleu). on utilisera donc l'objet **hsv2rgb** pour effectuer la conversion entre les triplets (teinte saturation luminosité) et (rouge vert bleu)

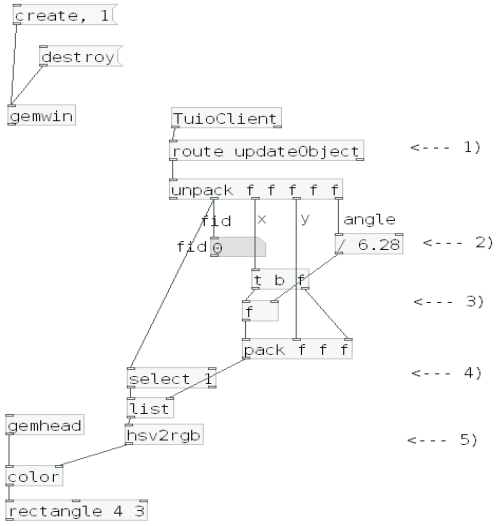

fig. 5 : exemple de mapping (position fiducial ↔ palette de couleurs)

1) on filtre pour ne laisser passer que les messages updateObject
2) conversion angle en valeur entre [0:1]
3) ordonnancement des messages
4) selection du fid==1
5) conversion Teinte / Saturation / luminosité en Rouge/Vert/Bleu

Cette solution fonctionne, mais n'est pas très lisible.
Voici maintenant comment le même problème est traité dans l'environnement fid_abs:

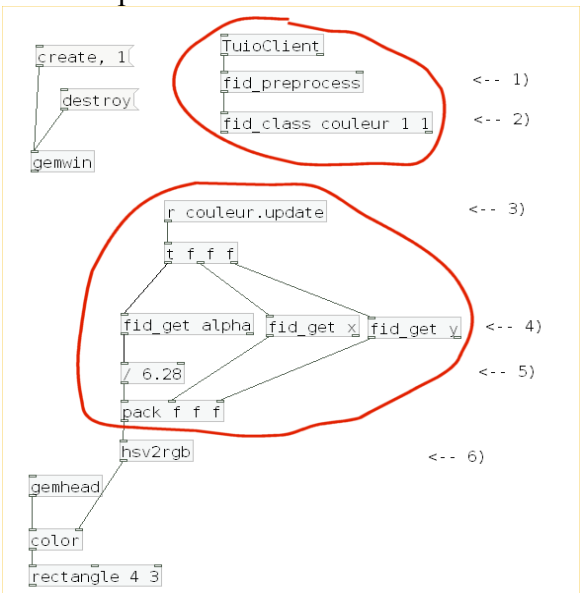

fig. 6 : autre méthode avec l'environnement fid abs, définition de classes d'objets avec fid_class

1) en-tête de prétraitement **fid_preprocess**. directement relié à **TuioClient**
2) avec **fid_class**, on crée une classe 'couleur' pour les événements associés au fiducial#1
3) un événement update de la classe 'couleur' se traduit par un nombre arrivant sur le récepteur **r couleur.update**. Ce nombre est égal à l'identifiant unique (*sid*) du fiducial.
4) À partir de ce *sid*, on récupère les champs 'alpha' , 'x' et 'y', grâce à l'objet **fid_get**.
5) conversion angle (alpha) en valeur entre [0:1]
6) conversion Teinte / Saturation / luminosité en Rouge/Vert/Bleu

Ce patch se sépare donc en deux blocs distincts :

- Dans le premier bloc, constitué des objets **TuioClient** / **fid_preprocess** / **fid_class couleur**,

    - on enregistre les informations fournies par **TuioClient** dans une structure de données (c'est le rôle de **fid_preprocess**)
    - on organise ces informations en définissant des classes, en fonction du numéro de fiducial, par exemple.
      Syntaxe : **fid_class <*classname*> <*fid_min*> <*fid_max*>**
- Dans le deuxième bloc, constitué des objets (**r couleur.update** / **fid_get** …) :
    - on déclenche une action en fonction d'un événement lié à une classe, grâce à un récepteur prédéfini : **r <*class_name*>.<*event_type*>**.
      Ici, il s'agit du récepteur **r couleur.update** mais cela aurait pu être également le récepteur **r couleur.add** ou **r couleur.remove**.
    - on récupère des informations enregistrées dans la structure de stockage, au moyen de l'objet **fid_get <*paramètre*>**.
      Ici, il s'agit de **fid_get alpha**, **fid_get x** et **fid_get y**.

En conclusion :

- Le problème de l'ordonnancement des messages ne se pose plus.
- Le patch est plus lisible, on n'a pas à se soucier de la structure des messages de TuioClient.

Application n°2

Plusieurs fiduciaux sont placés sur la surface tangible.
On voudrait les représenter à l'écran.
Comment procéder ?

La première idée est la suivante : on crée une chaine de rendu GEM, et on modifie les coordonnées avec **translateXYZ**, en fonction des coordonnées du fiducial détecté.

Il faut opérer une conversion d'échelle entre x et y fournis par **fid_get**, et dont les valeurs sont comprises entre 0 et 1, et le système de coordonnées utilisé par GEM, qui change selon le format d'affichage utilisé, comme le montre le tableau suivant:

	TuioClient	GEM (1:1)	GEM(4:3)	GEM(16:10)	GEM(16:9)
Exemples de résolutions possibles		500x500	320x240 640x480 800x600	1280x800	1280x720
x	[0 , 1]	[-4 , 4]	[-5.333 , 5.333]	[-6.4 , 6.4]	[-7.111 , 7.111]
y	[1 , 0]	[-4 , 4]	[-4 , 4]	[-4 , 4]	[-4 , 4]

Remarque :
ces valeurs s'appliquent pour le positionnement par défaut de la caméra virtuelle dans GEM.

Un début de solution est le suivant :

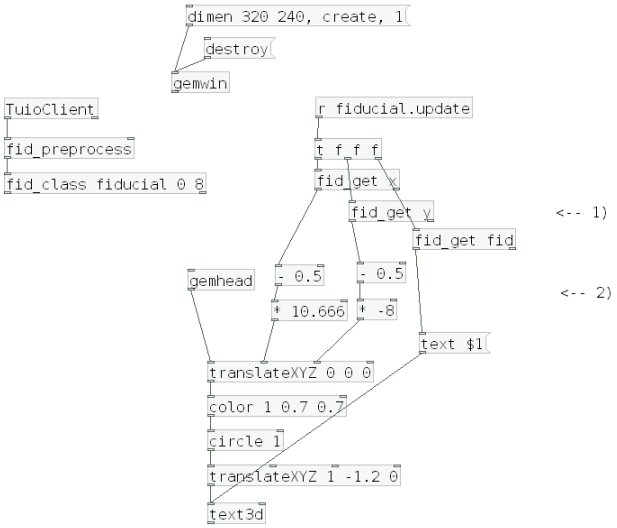

fig. 7 : représentation graphique dans GEM d'un objet fiducial

1) On extrait les coordonnées x et y au moyen de **fid_get**. On récupère également le numéro de fiducial au moyen de **fid_get fid**, à des fins d'affichage.
2) On effectue les conversions d'échelles sus-mentionnées, avec notamment l'inversion du signe pour les ordonnées (y)

Cela fonctionne pour un seul fiducial présent, mais pas avec plusieurs fiduciaux en simultané !
Pour remédier à cela, réfléchissons comment dessiner 'itérativement' plusieurs objets à partir d'une seule chaîne de rendu GEM.

Dans l'exemple suivant, on affiche deux cercles distincts avec une même chaine de rendu:

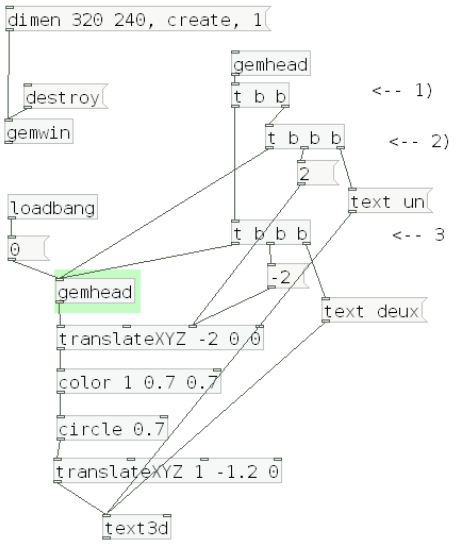

fig. 8 : dessiner deux objets avec une seule chaîne de rendu

1) un premier **gemhead** sert de déclencheur. Il envoie deux bangs à l'objet **gemhead** situé plus bas (souligné en vert), ce qui a pour effet d'activer la chaine de rendu deux fois.
   REMARQUE : le **gemhead** ainsi déclenché 'manuellement' est initialement désactivé par **loadbang** + message '0'.
2) avant d'activer la chaine de rendu, on modifie les paramètres propres à chaque cercle (**translateXYZ**, **text3d**, ..)
3) Ici, on répète deux fois cette procédure, mais on pourrait la répéter davantage, avec des paramètres différents. Pour bien faire, il faudrait répéter cette procédure autant de fois qu'il y a d'objets à représenter à l'écran. C'est ce que nous allons voir maintenant.

Nous utiliserons une nouvelle fois l'environnement fid_abs pour définir une classe d'objet.

Rappelons que nous souhaitons déclencher **gemhead** autant de fois qu'il y a de fiduciaux à afficher, c'est à dire *pour chaque instance* présente de la classe (ici, la classe baptisée 'fiducial')

L'objet **fid_get_instances** renvoie toutes les instances existantes d'une classe.
L'objet **fid_count_instances** compte le nombre d'instances disponibles.

Le patch suivant permet d'étudier le fonctionnement de ces 2 objets :

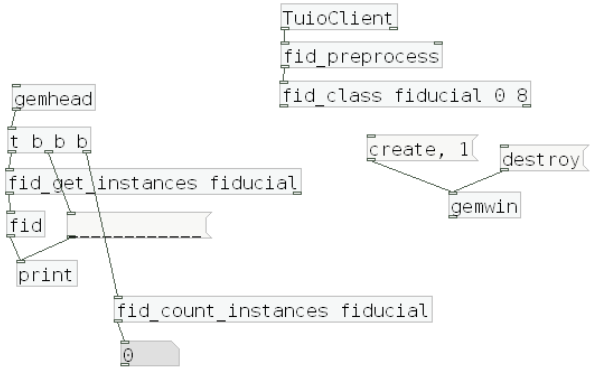

fig. 9 : les objets fid_get_instances et fid_count_instances

Revenons au patch représenté *fig. 08.*

On y insère **fid_get_instances** pour provoquer 'en boucle' l'action de dessiner chaque fiducial avec la même chaine de rendu GEM.
L'objet [**fid_get_instances fiducial**] serait l'équivalent d'un objet 'for each' dans un langage script.

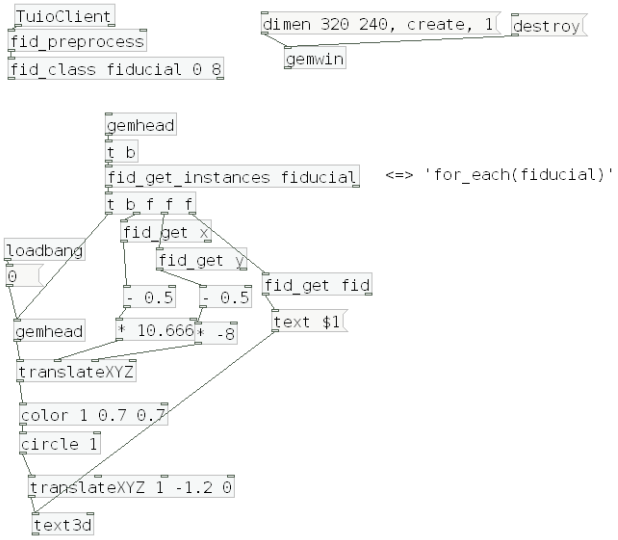

fig. 10 : une chaîne de rendu utilisée de manière itérative

pour chaque fiducial :
-on extrait le fid (**fid_get fid**)
-on extrait / convertit les coordonnées x et y (**fid_get x / y**)
-on déclenche le rendu en envoyant un bang à **gemhead**.

Simplification du patch

Il est possible d'indiquer à l'environnement fid_abs dans quel format d'affichage on travaille. Pour cela, on ajoute l'argument GEM=*x:y* à l'objet **fid_preprocess**, où *x:y* peut être 1:1; 4:3, 1024:768, 16:10, etc...

On peut alors substituer à **fid_get x** et **fid_get y** l'objet **fid_get coords**, qui contient un doublet de coordonnées converties dans l'échelle GEM (*fig 11*), ce qui simplifie le patch

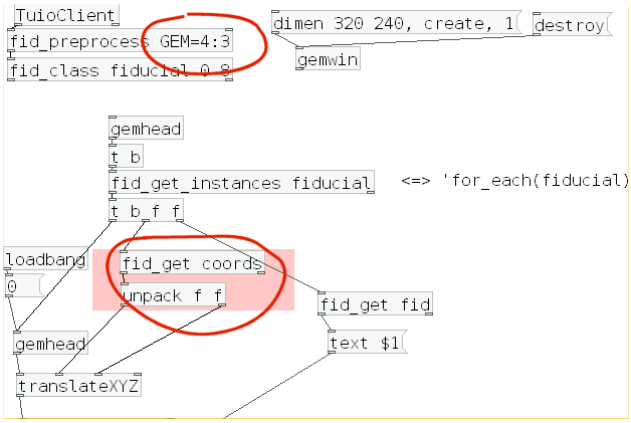

fig. 11 : fid_get coords permet de convertir les coordonnées x y dans le système GEM

On peut encore davantage réduire la chaîne de rendu en faisant appel à l'objet **fid_gui** (*fig. 12*) :

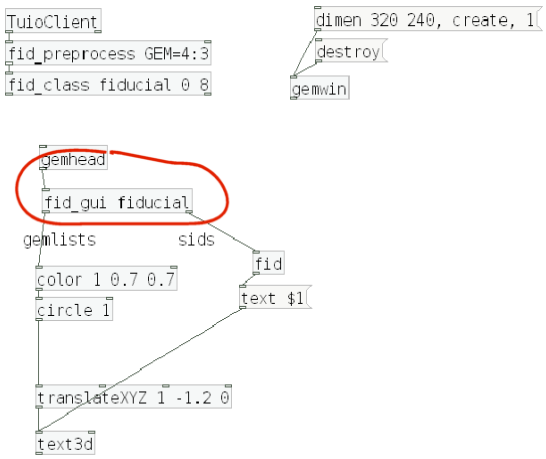

fig. 12 : l'objet fid_gui est une sorte de 'macro' permettant de simplifier le processus précédent

Chaque fois qu'il est activé par un bang (ou n'importe quel message), **fid_gui** envoie le sid (outlet droit) puis la gemlist (outlet gauche), et ce, pour chacune des instances de classe présentes.

Il devient alors très rapide de représenter des classes d'objets différentes à l'écran, par des chaines de rendu distinctes :

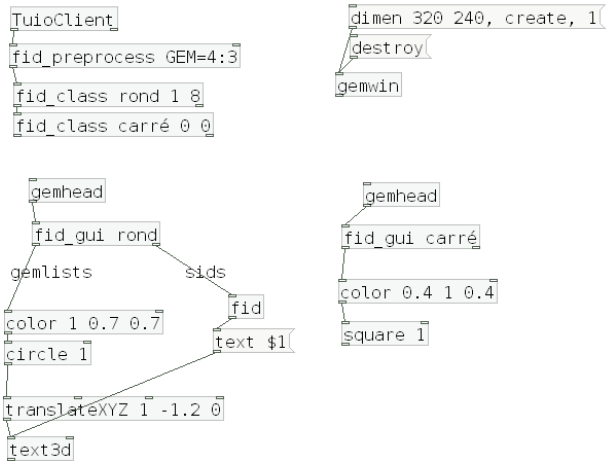

fig. 13 : deux chaines de rendu pour deux classes d'objets différentes

Remarque : On ne peut pas utiliser **fid_gui** pour afficher plusieurs vidéos différentes en simultané, car l'objet **pix_film** ne gère qu'un fichier à la fois, et ne peut pas être invoqué de manière itérative, comme le fait **fid_gui**. Dans ce cas, il faut alors utiliser autant de modules gemhead qu'il y a de films à lire, et la structure du patch est semblable à celle du lecteur de sample que nous allons étudier ci-après. En revanche, **pix_multiimage** devrait fonctionner avec **fid_gui** (non testé).

Nous venons de voir comment générer un effet visuel à partir d'événements Tuio, nous allons maintenant générer du son.

Exemple : un lecteur de samples

Nous voulons créer un lecteur de samples sonores, déclenché par la détection d'un fiducial par la caméra.

Le fonctionnement minimal est constitué de trois actions :

- charger le fichier-son en mémoire
- démarrer la lecture du sample
- arrêter la lecture du sample

Les deux premières actions sont déclenchées quand on pose le fiducial sur la surface tangible, la troisième quand on retire le même fiducial, ce qui correspond respectivement aux messages *add* et *remove*.

Pour lire un son dans un fichier, nous utiliserons l'objet **readsf~** de Pure Data.
Le patch correspondant est le suivant :

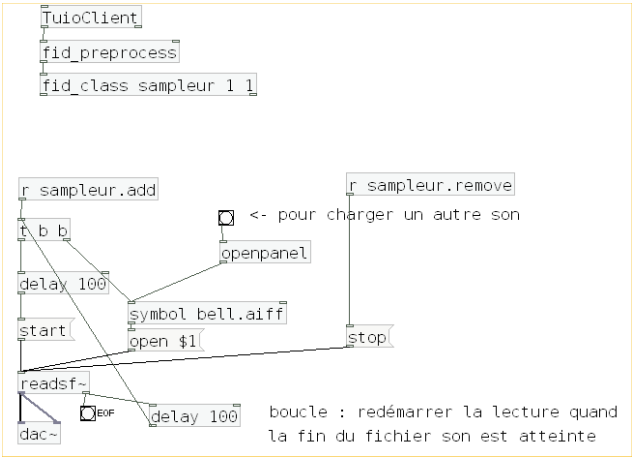

fig. 14 : déclenchement basique d'un sample avec un fiducial

Ce patch est très basique, on voudrait maintenant pouvoir ajuster le volume et la balance en fonction de la position du fiducial.

Pour cela, on effectue un mapping simple entre volume et position y d'une part, balance et position x d'autre part. On récupère x et y avec les objets fid_get.

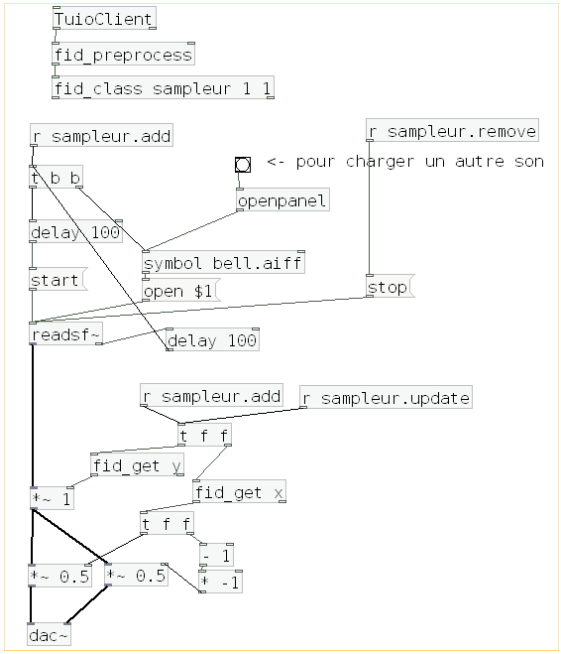

fig. 15 : gestion du volume et de la balance dans l'exemple précédent

On notera que le volume et la balance sont gérés pour des événements *add* mais aussi *update*, ce qui permet de modifier leurs valeurs quand on déplace le fiducial.

Remarque : le rendu sonore est moyen, les variations de volume sont hachées. C'est normal. Pour bien faire, il faudrait piloter les objets ***~** par des générateurs de rampe (**line~**) pour bien lisser les commandes de variation de volume et de balance.

Nous voudrions maintenant un fonctionnement polyphonique, c'est à dire pouvoir lire plusieurs samples simultanément.

Cela implique de créer plusieurs unités de lecture **readsf~**.

Préalablement, et dans l'optique de faciliter la lecture ultérieure du patch, nous allons placer l'unité de lecture dans un sous-patch, et gérer un peu différemment ses commandes, de façon à rendre l'architecture la plus modulaire possible. Ce qui veut dire que cette unité de lecture devra être commandée par des messages play, stop, volume, pan (balance), en aval et indépendamment de la chaine de traitement tuio.

Dans ce sous-patch (**pd lecteur-sample**), on n'utilise qu'une seule entrée (**inlet commandes**), et on aiguille les différents messages grâce à l'objet **route** :

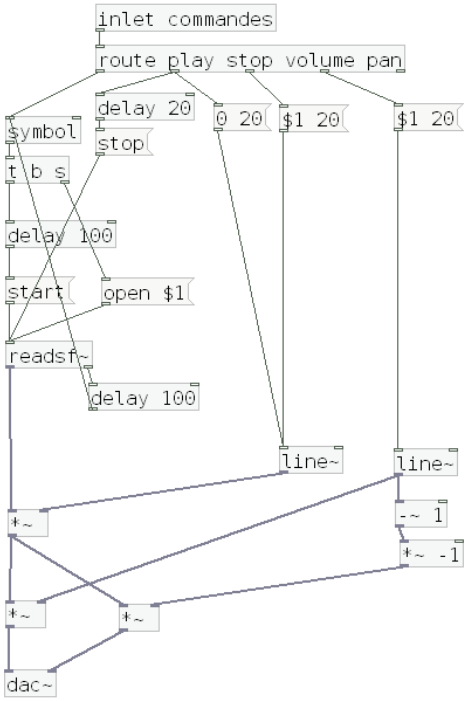

fig. 16 : réécriture dans un sous-patch du module de lecture de sample

Le patch principal, qui gère les messages tuio, prend l'aspect suivant :

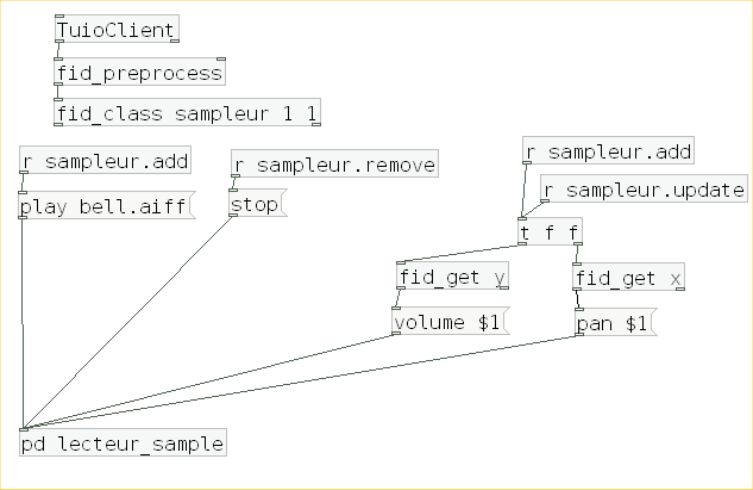

fig. 17 : le patch principal correspondant au sous-patch précédent

Réécrivons maintenant ce patch de manière à pouvoir jouer plusieurs samples simultanément.

L'idée est d'associer un numéro d'ordre à chaque unité de lecture, et de faire correspondre ces numéro avec ceux des fiduciaux.

Pour cela, on modifie le patch précédent de la manière suivante :

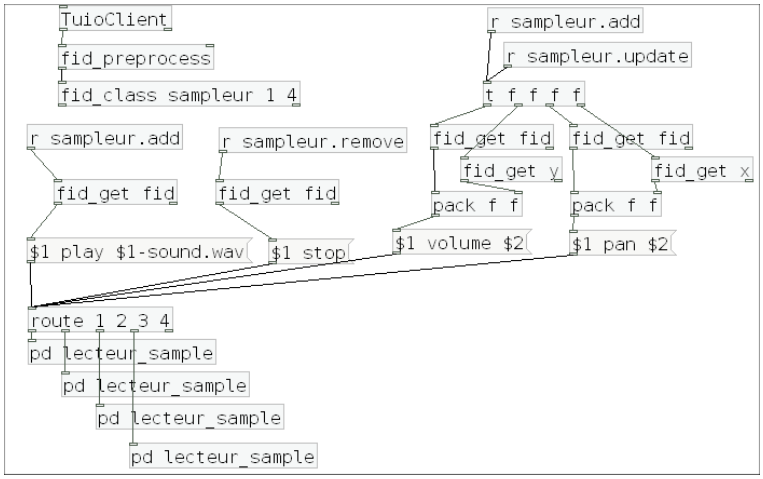

fig. 18 : Version polyphonique

L'objet **route** assure l'acheminement des messages vers l'unité concernée, en fonction de la valeur du nombre qui figure maintenant en « préfixe » de chaque message.

A noter que l'on spécifie maintenant un nom de fichier différent en fonction du fid.
Nous pouvons jouer à présent 4 sons différents, mais nous ne pouvons pas jouer simultanément plusieurs fois le même sample (par exemple en ayant 2 fiduciaux identiques sur la table). En effet, comme le routage est défini par le fid, les messages générés à partir de ces fiduciaux iraient tous vers la même unité de lecteur_sample !

Il faut donc imaginer un autre mécanisme pour l'affectation dynamique des unités lecteur_sample.

Dans les objets natifs de pure data, **poly** permet l'affectation dynamique de messages MIDI de type note on / note off vers des ressources données, en fonction du nombre de voies spécifié.

Fid_Abs contient un objet analogue, baptisé **fid_multiplex**, qui permet l'affectation dynamique des messages vers des unités de ressources disponibles, en fonction d'un identifiant unique qui doit figurer en en-tête de chaque message.

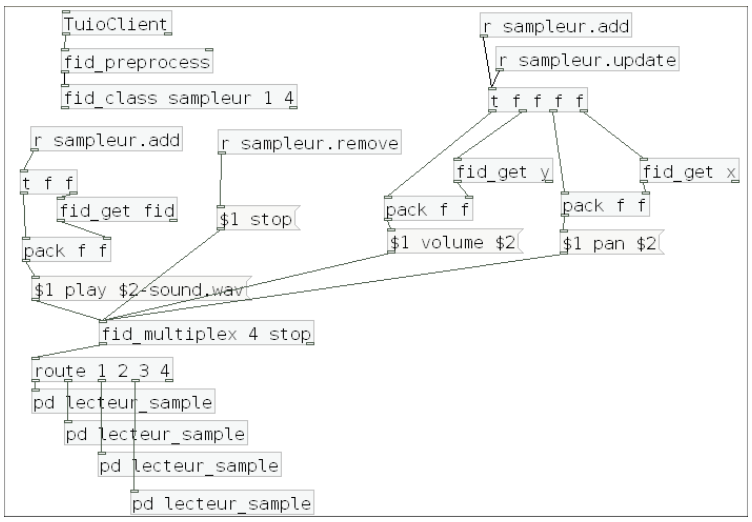

fig. 19 : Mise en oeuvre de l'objet fid_multiplex

fid_multiplex transmet tous les messages qui le traversent, en substituant au sid qui figure en en-tête un numéro de ressource.

Le premier argument de fid_multiplex est le nombre d'unités de ressources disponibles en parallèle. Le second argument, optionnel, spécifie le message/mot-clé utilisé pour libérer les ressources. Par défaut, ce mot-clé est 'remove'.

Résumé

- Nous avons étudié la structure des messages générés par TuioClient : addCursor, updateCursor, removeCursor, addObject, updateObject, removeObject
- En utilisant l'environnement fid_abs, nous n'avons plus besoin de nous souvenir de la structure interne de ces messages
- Nous avons utilisé les objets suivants de Fid_Abs
  - **TuioClient** est raccordé à **fid_preprocess**
  - **fid_preprocess** accepte comme argument optionnel le format de la fenêtre GEM : GEM=1:1 / GEM=4:3 / GEM=1024:768 / etc....
  - **fid_class** définit des classes en fonction du numéro de fiducial
  - **r <*class_name*>.add**, **r <*class_name*>.update**, **r <*class_name*>.remove** déclenchent la gestion événementielle
  - **fid_get <*parametre*>** permet de récupérer les infos : **fid_get x** , **fid_get y**, **fid_get alpha**, **fid_get coords**, **fid_get fid**, etc...
  - **fid_get_instances <*classname*>** énumère toutes les instances d'une classe en renvoyant leurs sids respectifs
  - **fid_count_instances <classname>** compte toutes les instances d'une classe.
  - **fid_gui <*classname*>** simplifie le code pour la représentation graphique des instances-membres d'une classe
  - **fid_multiplex <*nb_voies*>** simplifie l'affectation dynamique des ressources de traitement, dans le cas d'une utilisation multivoies / polyphonique