51-Jitter Java

Ce tutoriel suppose que le lecteur est familier avec le processus de programmation des classes *mxj* Java. Les détails de cette procédure sont contenus dans le document *Writing Max Externals* in Java.

Dans ce tutoriel, nous allons décrire comment les classes *mxj* peuvent opérer directement sur les cellules d'une matrice d'entrée. Nous verrons également comment créer des classes dans le langage de programmation Java qui chargent en interne des objets Jitter et définissent et exécutent un réseau de traitement. Enfin, nous apprendrons à concevoir des éléments d'interface utilisateur, accélérés par le hardware, en attachant un "écoutant" à un contexte de dessin et en interrogeant une fenêtre pour les événements de souris.

L'API Java de Jitter s'articule autour de la classe JitterObject, qui permet aux classes *mxj* d'instancier des objets Jitter. La ligne de code suivante crée un objet Jitter *jit.op*.

```
JitterObject jo = new JitterObject("jit.op") ;
```

Nous pouvons envoyer des messages aux objets dans notre code Java en utilisant les méthodes call() ou send() de JitterObject. Par exemple, nous pouvons définir l'opérateur dans l'instance de *jit.op* ci-dessus comme suit :

```
jo.send("op", "+") ;
```

Nous pourrions également définir cet attribut à l'aide de la méthode setAttr() :

```
jo.setAttr("op", "+") ;
```

La classe *JitterMatrix* prolonge JitterObject - après tout, *jit.matrix* est un Jitter Object. Puisque *jit.matrix* est un objet tellement commun, il est pratique d'avoir des méthodes pratiques dédiées à sa création et sa suppression. Par exemple, la ligne de code ci-dessous crée une nouvelle *JitterMatrix* à 4 plans de type **char** et de dimensions 320 par 240.

```
JitterMatrix jm = new JitterMatrix(4, "char", 320, 240) ;
```

Nous pouvons également créer un objet JitterMatrix en spécifiant le nom de la matrice :

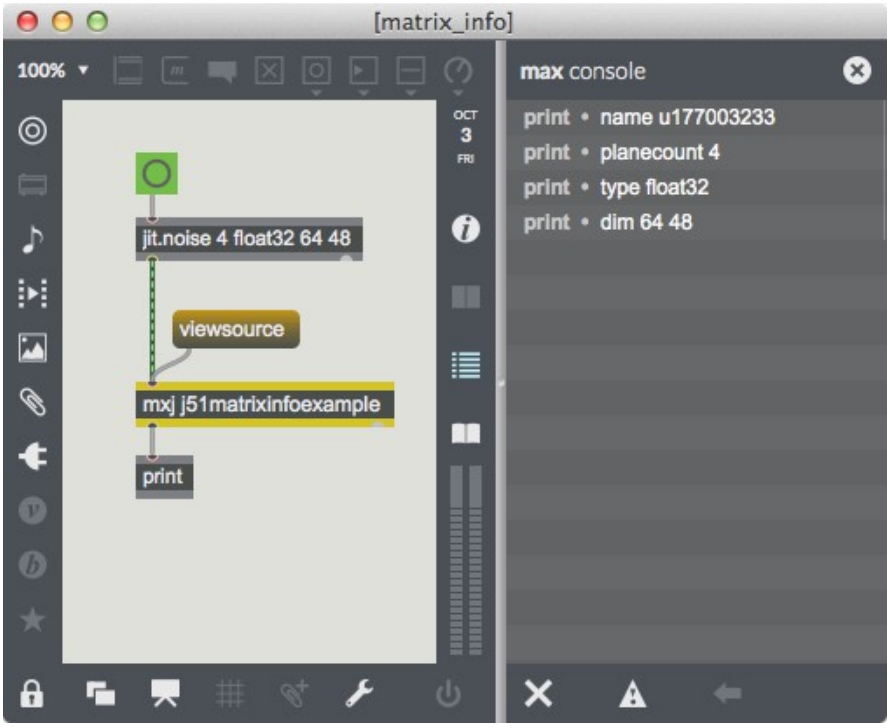
```
JitterMatrix jm = new JitterMatrix("Stanley") ;
```

Si un objet *jit.matrix* nommé Stanley existe déjà, cet objet Java *peer* JitterMatrix y fera référence. Dans ce cas, un nouvel objet *jit.matrix* n'est pas créé.

Les avantages sont appréciables, mais la raison principale pour laquelle la classe JitterMatrix existe est de fournir des méthodes natives pour accéder aux données dans les cellules de la matrice. Nous verrons un exemple de ceci en action lorsque nous ouvrirons le patch du tutoriel.

Accès à une matrice d'entrée

- Ouvrez le patch du tutoriel 51jJava dans le dossier Jitter Tutorials. Double-cliquez sur le sub-patch *matrix info*. Cliquez sur l'objet *button* et observez la sortie dans la console Max.



Informations sur notre matrice imprimées dans la console Max.

Dans ce sub-patch, le fait de cliquer sur l'objet *button* amène l'objet *jit.noise* à envoyer une matrice **float32** de 4 plans 64 par 48 dans un objet *mxj* avec la classe j51matrixinfoexample chargée. Cette classe Java envoie quelques informations de base sur la matrice d'entrée à sa sortie. Examinons le code de cette classe :

```
import com.cycling74.max.* ;
import com.cycling74.jitter.* ;

public class j51matrixinfoexample extends MaxObject {

  public void jit_matrix(String s)
  {
   JitterMatrix jm = new JitterMatrix(s) ;

   outlet(0, "name", jm.getName()) ;
   outlet(0, "planecount", jm.getPlanecount()) ;
   outlet(0, "type", jm.getType()) ;
   outlet(0, "dim", jm.getDim()) ;
  }
}
```

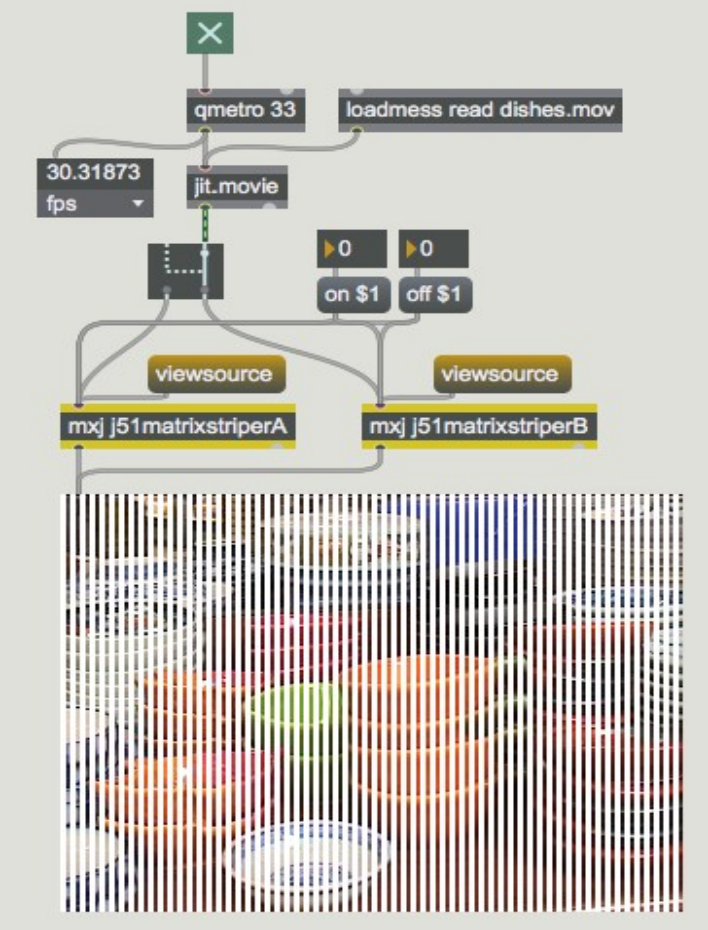
La classe est constituée d'une seule méthode *jit_matrix*. Bien sûr, puisque les matrices sont transmises entre les objets Jitter comme des références nommées, nous pensons qu'une matrice est entrée comme une simple liste de deux éléments, et c'est ainsi que *mxj* la considère. L'argument du message *jit_matrix* est bien sûr le nom de la matrice, donc la première chose que fait notre méthode est de créer un nouvel objet *JitterMatrix* avec le nom de la matrice d'entrée comme seul argument de son constructeur. L'objet *JitterMatrix* qui est créé aura comme homologue l'objet *jit.matrix* nommé. Les quatre lignes de code suivantes produisent simplement les résultats de quelques recherches de base qui peuvent être faites en utilisant les méthodes de la classe *JitterMatrix*.

Opérations dans une matrice

- Fermez le sub-patch d'informations sur la matrice. Ouvrez le sub-patch *striper*.

Le sub-patch striper nous donne un exemple d'une classe qui opère sur les données d'une matrice d'entrée. Notez qu'il y a deux classes configurées pour être utilisées, avec un objet *Ggate* qui nous permet de passer de l'une à l'autre ; nous allons comparer deux façons différentes d'effectuer la même opération pour voir laquelle est la plus efficace.

- Cliquez sur le *toggle* pour lancer l'objet *qmetro*.



Un effet de "rayures" sur une matrice Jitter.

Ces classes "rayent" une matrice d'entrée en itérant sur chaque ligne de la matrice et en écrasant les valeurs de certaines cellules. Les cellules qui sont écrites dépendent des valeurs des attributs *on* et *off* : si l'attribut *on* a la valeur **v** et *off* la valeur **w**, l'algorithme écrasera les cellules **v**, puis n'écrira pas dans les cellules **w**, et ainsi de suite en itérant par ligne de la matrice.

Voici le code de *j51matrixstriperA* :

```
import com.cycling74.max.* ;
import com.cycling74.jitter.* ;

public class j51matrixstriperA extends MaxObject {

  JitterMatrix jm = new JitterMatrix() ;
  int frgb[] = new int[] {255, 255, 255, 255} ;
```

```
  int on = 2, off = 1 ;

  j51matrixstriperA()
  {
   declareAttribute("frgb ")
   declareAttribute("on ")
   declareAttribute("off ")
  }

  //notez que cette méthode suppose une matrice char 2D !
  public void jit_matrix(String s)
  {
   jm.frommatrix(s) ;
   int dim[] = jm.getDim() ;
   int count = 0 ;
   boolean notoff = true ;
   for (int i="0;i<dim[1];i++) "
     for(int j="0;j<dim[0];j++) "
     {
      if (notoff)
        jm.setcell2d(j, i, frgb) ;
      si ((notoff &&(++count > on))
          ||(!notoff&&(++count > off))))
      {
        count = 0 ;
        notoff = !notoff ;
      }
     }
   outlet(0, "jit_matrix", jm.getName()) ;
  }
};; ;
```

Dans la méthode *jit_matrix*, nous utilisons la méthode getDim() pour connaître les dimensions de notre matrice et les stocker dans le tableau *int dim*, que nous utilisons ensuite comme conditions terminales dans les boucles for() de notre procédure itérative. La méthode setcell2d nous permet de définir directement la valeur de n'importe quelle cellule de la matrice. Lorsque nous avons terminé le traitement, nous envoyons un message *jit_matrix* avec le nom de notre *JitterMatrix* comme argument.

- Cliquez d'avant en arrière sur le *toggle* de l'objet Ggate. Quelle classe est la plus rapide ?

La seule différence entre j51matrixstriperA et *j51matrixstriperB* est la méthode *jit_matrix*. Voici la méthode *jit_matrix* de *j51matrixstriperB* :

```
//notez que cette méthode suppose une matrice char 2D !
public void jit_matrix(String s)
{
 jm.frommatrix(s) ;
 int dim[] = jm.getDim() ;
 int count = 0 ;
 int planecount = jm.getPlanecount() ;
 int offset[] = new int[]{0,0} ;
```

```
boolean notoff = true ;
int row[] = new int[dim[0]*planecount] ;

for (int i="0;i<dim[1];i++) "
{
  offset[1] = i ;
  jm.copyVectorToArray(0, offset, row,
            dim[0]*planecount, 0) ;
  for(int j="0;j<dim[0];j++) "
  {
    if (notoff)
    {
      for (int k="0;k<planecount; k++) "
        row[j*planecount+k] = frgb[k] ;
    }
    if ((notoff &&(++count > on))
        ||(!notoff&&(++count > off))))
    {
      count = 0 ;
      notoff = !notoff ;
    }
  }
  jm.copyArrayToVector(0, offset, row,
            dim[0]*planecount, 0) ;
}
outlet(0, "jit_matrix", jm.getName()) ;
```

Plutôt que de définir les valeurs de cellule une par une dans la matrice, la méthode *jit_matrix* de *j51matrixstriperB* saisit une ligne entière de la matrice en utilisant la méthode ***copyVectorToArray*** de *JitterMatrix*, écrase les valeurs appropriées dans la rangée, puis réécrit la rangée dans la matrice en utilisant la méthode ***copyArrayToVector*** de *JitterMatrix*. Comme vous l'avez peut-être remarqué, cette version de la classe est nettement plus rapide que la version qui définit les cellules de la matrice une par une. Il s'agit d'une excellente démonstration de la chose très importante suivante à garder à l'esprit : il est très coûteux de franchir la frontière C/Java. La version de cet objet qui utilise *setcell* doit faire un aller-retour à travers la frontière C/Java, une fois pour chaque cellule de la matrice. La dernière version ne fait que deux allers-retours par rangée. Si vous avez essayé les deux versions de l'objet, les économies sont évidentes.

Les méthodes ***copyVectorToArray*** et ***copyArrayToVector*** sont saturées de signatures qui couvrent tous les types de données pertinents. Comme vous pouvez le constater en examinant attentivement le code ci-dessus, ces méthodes fournissent et attendent les tableaux Java avec les données planaires multiplexées afin que toutes les données d'une cellule soient présentées de manière contiguë. Il existe également des méthodes ***copyVectorToArrayPlanar*** e t ***copyArrayToVectorPlanar*** pour déplacer les données d'un seul plan dans Java.

> Lorsque l'on utilise des classes *mxj* pour traiter des matrices dans Jitter, il n'y a pas de fonctionnalité intégrée, il est donc important de toujours piloter le réseau d'objets à l'aide d'un objet *qmetro*, ou d'une autre structure qui reporte les événements pour éviter les retards.

Copie des données d'entrée

Dans les exemples ci-dessus, vous avez peut-être remarqué la manière différente avec laquelle nous utilisons notre *JitterMatrix* en interne. Alors que dans l'info de l'exemple de matrice nous créions une nouvelle *JitterMatrix* à chaque fois qu'une matrice d'entrée était reçue, cette fois-ci nous mettons en cache une instance de *JitterMatrix* et copions les données d'entrée dans celle-ci à chaque fois en utilisant la méthode *frommatrix*. Pourquoi faisons-nous cela ?

- Arrêtez le *qmetro* et fermez le sub-patch *striper*. Ouvrez le sub-patch ***why copy ?*** Activez le *qmetro*.

Why copy?

L'objet *trigger* de ce sub-patch envoie d'abord la sortie de l'objet *jit.noise* dans une instance de la classe ***mxj j51whycopy*** :

```
import com.cycling74.max.* ;
import com.cycling74.jitter.* ;

public class j51whycopy extends MaxObject {

  JitterMatrix jm = new JitterMatrix() ;
  booléen copy = false ;

  j51whycopy()
  {
   declareAttribute("copy ")
  }

  public void jit_matrix(String inname)
  {
   //dans des circonstances normales
   //nous ne créerions cette matrice qu'une seule fois
   jm = new JitterMatrix() ;
   if (copy)
   {
     jm.frommatrix(inname) ;
   }
```

```
    else //!copy
    {
      jm = new JitterMatrix(inname) ;
    }
    zero(jm) ;
    outlet(0, "jit_matrix", jm.getName()) ;
  }

  //notez que cette méthode suppose que la matrice est de type char.
  private void zero(JitterMatrix m)
  {
    int z[] = new int[m.getPlanecount()] ;
    for (int i="0;i<m.getPlanecount();i++) "
      z[i] = 0 ;
    m.setall(z) ;
  }
} ;
```

L'attribut *copy* fait basculer la méthode *jit_matrix* entre deux modes différents : si *copy* est ***vrai***, les données de la matrice d'entrée sont copiées dans notre *JitterMatrix* interne à l'aide de la méthode *frommatrix*. Si *copy* n'est ***pas vrai***, la *JitterMatrix* jm est créée avec la matrice d'entrée comme homologue. Dans les deux cas, notre *JitterMatrix* est mise à zéro à l'aide de la méthode *setall*, puis elle est sortie.

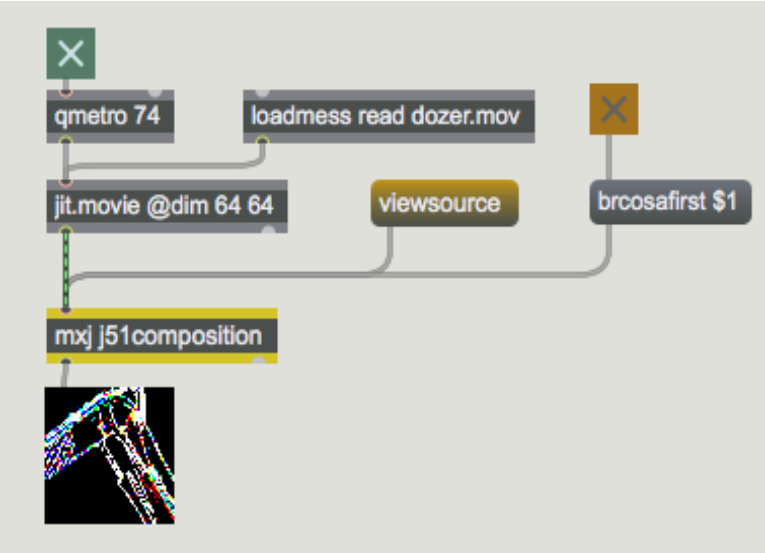
- Activez et désactivez l'attribut de copie avec le *toggle*. Que se passe-t-il dans la fenêtre *jit.p* de gauche ?

Lorsque l'attribut de copie est désactivé et que nous créons une nouvelle *JitterMatrix* associée au nom de l'entrée, nous opérons directement sur les données de cette matrice. Par conséquent, nous pouvons modifier le contenu de la matrice avant que d'autres objets aient eu la chance de le visualiser. Dans cet exemple de patch, lorsque l'attribut *copy* est ***vrai***, la fenêtre *jit.p* de gauche affiche correctement la matrice produite par *jit.noise*. Lorsque l'attribut *copy* est ***faux***, les deux objets *jit.pwindow* sont noirs car notre classe *mxj* modifie directement les données de la matrice. Par conséquent, si nous opérons sur une matrice, nous devons nous assurer de copier les données d'entrée dans un cache interne comme nous le faisons ici avec *frommatrix*.

Composition d'objets

Dans cette section, nous allons examiner l'utilisation de plusieurs JitterObjects dans une seule classe Java.

- Désactivez le *qmetro* et fermez la fenêtre du sub-patch. Ouvrez le sub-patch de composition. Activez le *qmetro*.



Une chaîne de traitement de Jitter s'exécutant dans Java.

Examinons le code de la classe *j51composition* :

```
import com.cycling74.max.* ;
import com.cycling74.jitter.* ;

public class j51composition extends MaxObject {
  JitterMatrix jm = new JitterMatrix() ;
  JitterMatrix temp = new JitterMatrix() ;
  JitterObject brcosa ;
  JitterObject sobel ;
  boolean brcosafirst = false ;

  j51composition() {
    declareAttribute("brcosafirst ")
    brcosa = new JitterObject("jit.brcosa ")
    brcosa.setAttr("brightness", 2.0f) ;
    sobel = new JitterObject("jit.sobel ")
    sobel.setAttr("thresh", 0.5f) ;
  }

  public void jit_matrix(String mname)
  {
    jm.frommatrix(mname) ;
    temp.setinfo(jm) ;
    if (brcosafirst)
    {
      brcosa.matrixcalc(jm, temp) ;
      sobel.matrixcalc(temp, jm) ;
    }
    sinon
    {
      sobel.matrixcalc(jm, temp) ;
      brcosa.matrixcalc(temp, jm) ;
    }
    outlet(0, "jit_matrix", jm.getName()) ;
```

```
    }

    public void notifyDeleted()
    {
      brcosa.freePeer() ;
      sobel.freePeer() ;
    }
};; ;
```

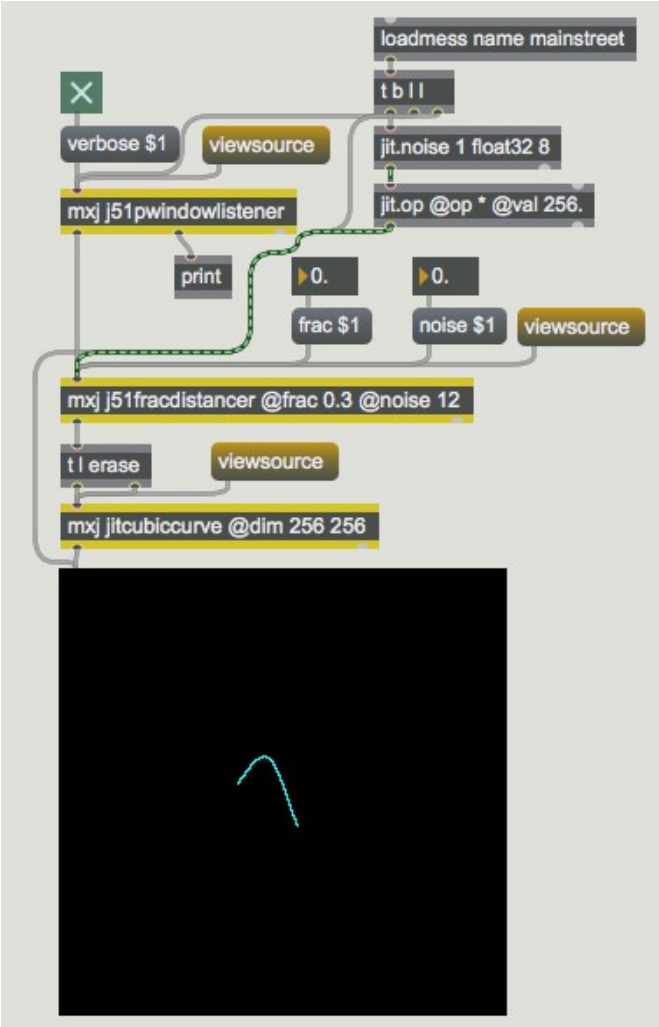
Deux *JitterObjects* sont créés dans le constructeur de la classe : *brcosa* contrôle un objet *peer* jit.brcosa, et *sobel* contrôle un objet *peer jit.sobel*. La méthode *jit_matrix* copie les données de la matrice d'entrée vers *jm*, puis définit les dimensions, le *planecount* et le type de la matrice *temp* à celle de *jm* avec la méthode *setinfo*. Après cela, les deux *JitterObjects* peuvent être utilisés pour traiter les données dans n'importe quel ordre : si l'attribut *brcosafirst* est ***vrai***, le *brcosa* opère d'abord et ensuite le *sobel*, et bien sûr vice versa si *brcosafirst* est ***faux***. Nous opérons sur les données dans une matrice en appelant la méthode *matrixcalc* d'un *JitterObject*. Les deux arguments sont des matrices d'entrée et de sortie - il est également possible d'utiliser des tableaux de matrices d'entrée et de sortie si l'objet *Jitter* homologue supporte plusieurs entrées ou sorties de matrice. Cet exemple simple montre comment il est possible de définir un réseau variable d'objets *Jitter* dans une classe Java.

- Activez et désactivez l'attribut *brcosafirst* pour voir la différence dans l'image résultante.

Enfin, notez que lorsque l'objet *mxj* est supprimé et que la méthode *notifyDeleted* est appelée, les objets *Jitter* homologues sont libérés en appelant la méthode *freePeer*() pour chaque *JitterObject* que nous avons instancié. Si la méthode *freePeer*() n'est pas appelée, les objets C homologues persisteront jusqu'à ce que le gestionnaire de mémoire Java effectue le nettoyage, ce qui peut prendre un certain temps.

Écoute

- Désactivez le *qmetro* et fermez le sub-patch. Ouvrez le sub-patch *cubiccurver*. Activez le *toggle* étiqueté *verbose*, déplacez la souris dans l'objet *jit.pwindow* et observez les résultats dans la console Max.



Cette section suppose que vous avez consulté le *Tutoriel 47* : Using Jitter Object Callbacks in JavaScript, qui traite des rappels d'objets en Javascript. L'API Java de Jitter fournit le même mécanisme pour " écouter " les événements générés par des objets Jitter nommés. Dans ce sub-patch, nous utilisons ce mécanisme pour suivre le mouvement de la souris dans une fenêtre nommée *jit.pwindow*. Examinons le code source de la classe *j51pwindowlistener* :

```
import com.cycling74.max.* ;
import com.cycling74.jitter.* ;

public class j51pwindowlistener extends MaxObject
implements JitterNotifiable
{
  JitterListener listener ;
  boolean verbose = false ;

  j51pwindowlistener()
  {
    declareIO(1,2) ;
    declareAttribute("verbose ")
  }

  public void name(String s)
  {
```

```
    listener = new JitterListener(s,this) ;
  }

  public void notify(JitterEvent e)
  {
    //ceci récupère le nom du contexte d'écoute
    String subjectname = e.getSubjectName() ;
    //Ceci récupère le type d'événement... souris, mouseidle, etc.
    String eventname = e.getEventName() ;
    //ceci récupère les arguments de l'événement
    Atom args[] = e.getArgs() ;
    if (verbose)
    {
      outlet(1,subjectname) ;
      outlet(1,eventname,args) ;
    }

    if ((eventname.equals("mouse ")
||(eventname.equals("mouseidle "))
    {
      int xy[] = new int[] {args[0].toInt(),
args[1].toInt()} ;
      //sortir les coordonnées x et y de la souris
      sortie(0, xy) ;
    }
  }
})) ;
```

La première chose à remarquer est que la classe implémente une interface appelée *JitterNotifiable*. Cette interface simple garantit que la classe possède une méthode *notify* qui sera appelée lorsqu'un événement est "entendu". Remarquez que la méthode *notify* prend un *JitterEvent* comme argument. La méthode *notify* de cette classe illustre les méthodes de ces *JitterEvents* que nous utiliserons pour extraire les données pertinentes : *getSubjectName* renvoie le nom du contexte d'écoute, ce qui est utile si nous écoutons à plusieurs endroits ; *getEventName* renvoie le nom de l'événement qui a été entendu ; et *getArgs* renvoie les arguments qui détaillent les paramètres de l'événement. Avec l'attribut *verbose* activé, la méthode *notify* de cette classe sort ces données par la deuxième sortie et un objet *print* les envoie à la Max Console.

La méthode *notify* continue en contrôlant la valeur de *eventname* à égalité avec *mouseidle* (pour un mouvement normal de la souris) ou *mouse* (pour un mouvement avec le bouton enfoncé). Si l'une de ces chaînes de caractères correspond, les deux premiers arguments, qui représentent la position x et y de la souris dans la fenêtre, sont émis sous forme de liste.

La dernière chose à noter à propos de cette classe est qu'avant de pouvoir écouter, l'écoutant doit être créé dans un contexte existant. Dans ce cas, cela se produit lorsqu'une instance de la classe reçoit un message nommé. Le déverrouillage du patch vous montrera comment l'objet *jit.pwindow* reçoit d'abord un nom ; le message nommé est ensuite envoyé à l'instance de *j51pwindowlistener*.

Le reste du patch utilise la classe d'exemple *jitcubiccurve* pour rendre une courbe cubique qui suit la souris. Les points de contrôle de la courbe cubique sont calculés à l'aide de la classe *j51fracdistancer*, dont vous pouvez lire le code pour vérifier si vous avez compris le contenu de cette leçon :

```
import com.cycling74.max.* ;
import com.cycling74.jitter.* ;
import java.util.* ;

public class j51fracdistancer extends MaxObject{

  JitterMatrix ctlmatrix =
   nouvelle JitterMatrix(1, "float32", 8) ;
  float ctldata[] = new float[8] ;
  int offset[] = new int[] {0} ;
  float frac = 0.5f ;
  float noise = 0.5f ;
  Random r = new Random() ;

  j51fracdistancer() {
   declareAttribute("frac") ;
   declareAttribute("noise") ;
  }

   //nouvelle entrée x,y
  public void list(int args[])
  {
   if (args.length != 2) return ;
   float x = (float)args[0] ;
   float y = (float)args[1] ;
   for (int i="0;i<4;i++)"
   {
     ctldata[i*2] += (x-ctldata[i*2]) * frac +
((float)r.nextGaussian()*noise) ;
     ctldata[i*2+1] += (y-ctldata[i*2+1]) * frac +
((float)r.nextGaussian()*noise) ;
   }
   ctlmatrix.copyArrayToVector(0,offset, ctldata, 8, 0) ;
   outlet(0, "jit_matrix", ctlmatrix.getName()) ;
  }

  public void jit_matrix(String mname)
  {
   JitterMatrix jm = new JitterMatrix(mname) ;
   if (jm.getDim()[0] != 8) return ;
   if (jm.getPlanecount() != 1) return ;
   if (!jm.getType().equals("float32")) return ;

   ctlmatrix.frommatrix(mname) ;
   ctlmatrix.copyVectorToArray(0, offset, ctldata, 8, 0) ;
  }
}
```

Résumé

Dans ce tutoriel, nous avons présenté les classes *JitterObject*, *JitterMatrix* et *JitterListener*, et nous avons vu comment les utiliser pour assembler des classes *mxj* qui opèrent directement sur les données, utiliser la composition d'objets *Jitter* existants pour définir des graphes de traitement, et comment écouter les objets *Jitter* pour les événements d'une manière à faciliter la construction de composants d'interface utilisateur, entre autres choses.