47-Utilisation de rappels d'objets Jitter en JavaScript

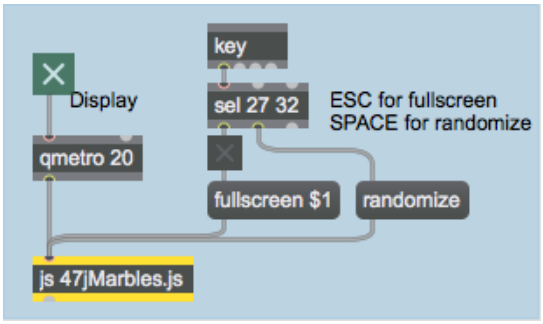
Dans les deux derniers tutoriels, nous avons examiné certaines des possibilités et des avantages de l'utilisation des objets et des matrices Jitter dans le code JavaScript. Dans ce tutoriel, nous allons encapsuler un patch Jitter OpenGL complet dans JavaScript, en utilisant plusieurs des techniques que nous avons déjà rencontrées. Mais surtout, nous apprendrons à "***écouter***" la sortie des objets Jitter qui ont été encapsulés dans JavaScript afin de concevoir des fonctions qui répondent de manière interactive à ces objets.

Un certain nombre d'objets Jitter (tels que *jit.movie*, *jit.pwindow* et *jit.window*) envoient des messages à partir de leurs sorties d'état (à droite) en réponse aux processus initiés par l'utilisateur dans un patch Max. Par exemple, lorsque vous lisez un fichier vidéo dans *jit.movie*, l'objet émet le message *read* suivi du nom de fichier du film et d'un numéro d'état à partir de sa sortie d'état. De même, les objets *jit.pwindow* et *jit.window* peuvent répondre aux mouvements de la souris dans leurs fenêtres en envoyant des messages par leur sortie d'état. Comme les objets Jitter instanciés en JavaScript n'ont pas d'entrées et de sorties en soi, nous devons créer explicitement un objet (appelé *JitterListener*) pour recevoir ces messages et appeler une fonction (appelée fonction de rappel) en réponse à ceux-ci.

Ce tutoriel suppose que vous avez consulté les tutoriels JavaScript précédents, ainsi que le premier tutoriel OpenGL Tutoriel 30 : *Drawing 3D Text*.

- Ouvrez le patch du tutoriel.

La première chose que nous remarquons à propos de ce patch est qu'il contient très peu d'objets Max. Pratiquement tout le travail effectué dans le patch est accompli dans l'objet *js* du patch.

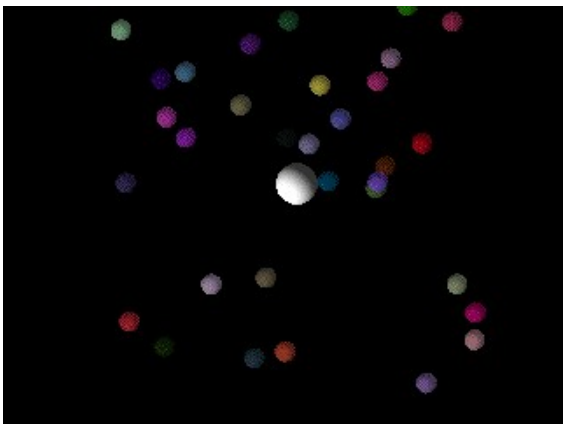


Notre patch de tutoriel : pas grand-chose à voir.

- Cliquez sur le *toggle* étiqueté **Display**. L'objet *qmetro* commencera à envoyer des messages **bang** à l'objet *js*, et l'objet *jit.window* devrait se remplir d'un certain nombre de formes (une grande sphère et quelques dizaines de petites sphères) sur un fond noir.

Notre objet js contient un ensemble complet d'objets Jitter effectuant un rendu OpenGL (y compris un objet *jit.window*).
- Déplacez votre souris sur la *jit.window*. La grande sphère se "colle" à votre souris et vous accompagne dans la fenêtre. Lorsque la grande sphère touche les petites sphères, elle les "pousse" comme s'il s'agissait d'objets solides qui entrent en collision. Essayez de vous habituer à déplacer les petites sphères avec la grande.

Notre fichier *js* ne dessine pas seulement notre scène OpenGL, mais gère également les événements interactifs de la souris. Cela se fait par le biais d'un mécanisme d'écoute et de rappel (appelé *JitterListener*) que nous allons découvrir dans le code JavaScript.



Pousser les sphères autour.

Création d'objets OpenGL en JavaScript

• Double-cliquez sur l'objet *js* dans notre patch Tutorial. Un éditeur de texte apparaît, contenant le code source de l'objet *js* dans le patch. Le code est enregistré dans un fichier appelé "***47jMarbles.js***" dans le même dossier que le patch du tutoriel. Regardez le bloc de code global en haut du fichier.

Notre code JavaScript crée tous les objets *JitterObject* dont nous avons besoin pour rendre notre scène : un objet *jit.window* pour afficher le contexte de dessin, un objet *jit.gl.render* pour effectuer le rendu, et un certain nombre d'objets *jit.gl.gridshape* pour dessiner et animer les sphères dans la scène. Ces objets sont instanciés dans le bloc de code global au début du fichier JavaScript.

var OBJECT_COUNT = 32 ; // nombre de petites sphères

Cette ligne représente une variable (OBJECT_COUNT) qui détermine le nombre de petites sphères dans notre scène.

```
// créer un objet [jit.window] pour notre écran
// (c'est l'objet que nous allons écouter) :
var mywindow = new JitterObject("jit.window", " ListenWindow ")
// Désactivez le test de profondeur... nous utilisons le mélange à la place :
mywindow.depthbuffer = 0
// activez l'idlemouseing... nous voulons l'écouter :
mywindow.idlemouse = 1)
```

Ensuite, nous créons un objet *jit.window* pour tout afficher. Nous y ferons référence dans notre code JavaScript par la variable *mywindow*. Comme dans un patch Max, la *jit.window* a besoin d'un nom ("***ListenWindow***") qui doit correspondre au contexte de dessin OpenGL que nous utilisons. Nous avons désactivé le *depthbuffer* et activé l'attribut *idlemouse*, qui permet à l'objet *jit.window* de suivre les événements de souris dans la fenêtre, que nous ayons cliqué ou non sur notre souris.

```
// créer un objet [jit.gl.render] pour dessiner dans notre fenêtre :
var myrender = new JitterObject("jit.gl.render", "ListenWindow")
// utiliser une projection en 2 dimensions :
myrender.ortho = 2
// Définissez l'arrière-plan en noir avec une opacité d'effacement totale (pas de traces) :
myrender.erase_color = [0, 0, 0, 1]
```

Notre objet *jit.gl.render* (assigné à la variable *myrender*) définit un contexte de dessin appelé "***ListenWindow***". L'objet *jit.window* (ci-dessus) et les objets *jit.gl.gridshape* (ci-dessous) partagent ce nom. Nous avons décidé d'utiliser une projection orthogonale (en fixant l'attribut *ortho* à **2**). Cela signifie que l'axe z de notre contexte de dessin est ignoré, en termes de dimensionnement des objets en fonction de leur distance par rapport à la caméra virtuelle. Cela crée essentiellement un rendu bidimensionnel de notre scène. Nous avons défini le fond en noir (avec un effacement complet entre les dessins successifs) en définissant notre couleur d'effacement à 0 0 0 1.

```
// créer un objet [jit.gl.gridshape] à utiliser pour contrôler à la souris
var mywidget = new JitterObject("jit.gl.gridshape", "ListenWindow")
mywidget.shape = "sphere" (sphère)
mywidget.lighting_enable = 1
mywidget.smooth_shading = 1
mywidget.scale = [0.1, 0.1, 0.1]
mywidget.color = [1, 1, 1, 0.5]
mywidget.blend_enable = 1
mywidget.position = [0, 0] // pas de z nécessaire en projection orthogonale
```

Le premier objet *jit.gl.gridshape* que nous créons (affecté à la variable *mywidget*) correspond à la grande sphère déplacée par notre souris dans la *jit.window*. Après l'avoir créé, nous définissons tous les attributs pertinents que nous voulons que la sphère possède, comme nous le ferions dans un patch Max. Notez que pour définir l'attribut de position, nous n'utilisons que deux arguments (x et y). Dans une projection orthogonale, l'axe z n'est pas pertinent pour la manière dont les formes sont dessinées.

```
// créer un tableau d'objets [jit.gl.gridshape].
// répartis aléatoirement dans la fenêtre
var mysphere = new Array(OBJECT_COUNT)
```

Plutôt que de nommer nos petites sphères individuellement, nous les traitons comme un tableau (appelé *mysphere*). Chaque élément de ce tableau sera alors un objet *jit.gl.gridshape* distinct auquel nous pourrons accéder par la notation de tableau (par exemple, *mysphere[5]* sera la sixième sphère, en comptant à partir de 0).

```
/ initialiser nos petites sphères avec des couleurs et des positions (x,y) aléatoires
for (var i = 0 ; i < OBJECT_COUNT ; i++) {
  mysphere[i] = new JitterObject("jit.gl.gridshape", "ListenWindow")
  mysphere[i].shape = "sphère"
  mysphere[i].lighting_enable = 1
  mysphere[i].smooth_shading = 1
  mysphere[i].scale = [0.05, 0.05, 0.05].
  mysphere[i].colour = [Math.random(), Math.random(), Math.random(), 0.5]
  mysphere[i].position = [Math.random() * 2 - 1, Math.random() * 2 - 1]
  mysphere[i].blend_enable = 1
}
```

Ce code crée en fait les objets *jit.gl.gridshape* individuels en tant que *JitterObjects* individuels qui sont membres du tableau *mysphere*. Nous utilisons une boucle JavaScript *for()* pour définir les attributs initiaux de tous ces objets en un seul bloc de code. Nous leur donnons des attributs de couleur et de position initiales aléatoires, en les dispersant sur l'écran et en les colorant toutes différemment.

```
 // créer un JitterListener pour notre objet [jit.window].
var mylistener = new JitterListener(mywindow.getregisteredname(), thecallback)
```

Enfin, nous créons une variable appelée *mylistener* que nous attribuons à un objet *JitterListener*. Les objets *JitterListener* prennent deux arguments : l'objet qu'ils écoutent et la fonction qui sera appelée lorsque l'objet déclenche un événement. Notre objet *JitterListener* est configuré pour écouter notre objet *jit.window* (*mywindow*). La propriété ***getregisteredname()*** d'un objet *JitterObject* renvoie le nom par lequel cet objet peut être accessible à travers le *JitterListener* (dans le cas des objets *jit.window*, ce sera le même que le nom du contexte de dessin). Chaque fois que notre objet *jit.window* génère un événement, une fonction appelée *thecallback()* sera déclenchée dans notre code JavaScript. Maintenant que nous avons instancié un *JitterListener*, nous pouvons (dans la plupart des cas) le négliger et simplement nous occuper de la mécanique de la fonction de rappel qu'il active en réponse à un événement de l'objet qu'il écoute.

• De retour dans le patch de Tutorial, cliquez sur la boîte de *message* intitulée *randomize* (aléatoire) ou appuyez sur la barre d'espace de votre clavier d'ordinateur. Les petites sphères devraient se disperser et changer de couleur. Regardez le code de la fonction *randomize()* dans notre fichier *js*.

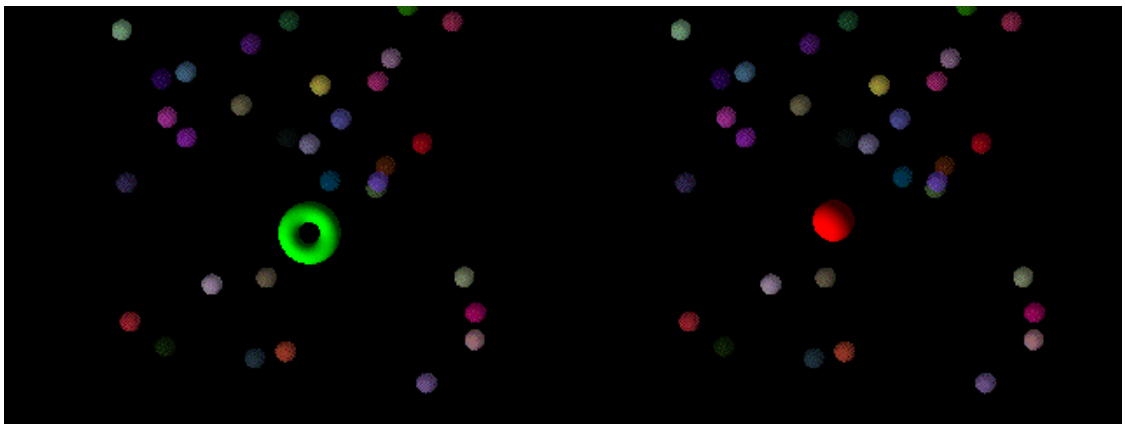
```
function randomize() {
  // réoriente les petites sphères
  for (var i = 0 ; i < 32 ; i++) {
    mysphere[i].colour = [Math.random(), Math.random(), Math.random(), 0.5]
    mysphere[i].position = [Math.random() * 2 - 1, Math.random() * 2 - 1]
  }
}
```

Notre fonction *randomize()* prend nos petites sphères et les disperse de manière aléatoire, en changeant leurs couleurs. Cela nous permet de redémarrer notre simulation quand nous le souhaitons.

La fonction de rappel

L'objet *jit.window* répond à une variété de mouvements de souris. Lorsque l'attribut *idlemouse* vaut **1** (comme c'est le cas dans notre code), l'objet *jit.window* émet des informations relatives à la souris (le message *mouseidle*) lorsque vous faites glisser le pointeur sur la fenêtre. Il nous indique également quand le pointeur a quitté la zone de la fenêtre (le message *mouseidleout*). Quel que soit le paramètre de l'attribut *idlemouse*, l'objet *jit.window* répond également aux clics de souris à l'intérieur de la fenêtre (message *mouse*).

• Dans le patch de Tutorial, déplacez le pointeur dans la *jit.window* et cliquez sur la souris. La sphère blanche se transforme en un tore vert. Relâchez la souris et le tore se transforme en une sphère rouge. En déplaçant le pointeur, la sphère redevient blanche.



Comment notre sphère réagit aux clics de souris (souris vers le bas et souris vers le haut).

• Regardez le code de la fonction appelée *thecallback()* dans notre code JavaScript. Ce code s'exécute chaque fois que notre objet *JitterListener* reçoit un événement de notre objet *jit.window*.

```
function thecallback(event)
// fonction de rappel pour gérer les événements déclenchés par la souris
// dans notre [jit.window]
{
        var x,y,button ; // quelques variables locales pour la fonction callback
```

Le constructeur d'événement stocke le message envoyé par l'objet *jit.window* à l'objet *JitterListener* dans notre patch. Nous déclarons également quelques variables locales (x, y et bouton) pour stocker les informations sur le pointeur dérivées des arguments du message d'événement.

```
if (event.eventname="="mouse")" {
// nous entrons, faisons glisser ou quittons un événement "clic de souris".
```

Cette partie du code gère un événement de clic de souris (un clic vers le bas, un déplacement de la souris et le relâchement d'un clic).

```
// les arguments sont (x,y,bouton,cmd,shift,capslock,option,ctrl)...
// seuls les trois premiers nous intéressent
x = event.args[0]
y = event.args[1]
button = event.args[2]
```

Les trois premiers arguments de l'événement souris contiennent la position de la souris (x et y) et un drapeau nous indiquant si le bouton de la souris est abaissé (1) ou relevé (0). Nous stockons ces paramètres dans des variables (x, y et bouton) pour les utiliser ultérieurement.

```
if (button) // on a cliqué vers le bas
  {
    mywidget.color = [0,1,0,1] ; // colore notre objet de contrôle en vert.
    mywidget.shape = "torus" ; // changez-le en forme de beignet.
  }
  else // nous venons de relâcher
  {
    mywidget.color = [1,0,0,1] ; // colorer notre objet en rouge
    mywidget.shape = "sphere" ; // redevient une sphère
```

```
    }
    // déplace notre objet de contrôle dans le contexte de dessin
    // l'équivalent de l'endroit où l'événement souris s'est produit :
    mywidget.position = myrender.screentoworld(x,y) ;
}
```

Si notre bouton est enfoncé, nous transformons notre objet *jit.gl.gridshape mywidget* d'une sphère blanche en un tore vert. Lorsque nous relâchons la souris, nous le transformons en une sphère rouge.

Enfin, nous actualisons la position de l'objet *jit.gl.gridshape* contrôlé par notre souris en mettant à jour l'attribut position de l'objet *JitterObject mywidget*. Étant donné que notre contexte de dessin fonctionne en coordonnées universelles OpenGL (où le centre par défaut de l'espace est (0, 0) et les positions sont exprimées en valeurs décimales), nous devons convertir nos valeurs x et y en nombres qui ont une signification pour notre contexte de dessin. La méthode *screentoworld()* d'un objet *jit.gl.render* convertira ces valeurs de coordonnées de pixels sur l'écran en coordonnées universelles pour le rendu.

```
else if (event.eventname=="mouseidle") " { [...]
    // nous survolons la fenêtre avec la souris relevée
    x = event.args[0] ;
    y = event.args[1] ;
    mywidget.color = [1,1,1,1] ; // colore notre objet en blanc
    mywidget.position = myrender.screentoworld(x,y) ;
}
```

Ce code s'exécute chaque fois que nous faisons glisser notre pointeur sur l'objet *jit.window* sans cliquer sur le pointeur. Tout d'abord, nous stockons les variables x et y en fonction de la position du pointeur, nous rendons notre objet *jit.gl.gridshape* (*mywidget*) blanc, au cas où le rappel précédent l'aurait rendu d'une couleur différente, et nous repositionnons à nouveau *mywidget*.

```
else if (event.eventname=="mouseidleout")" {
    // nous ne sommes plus en train de survoler la fenêtre avec la souris
    mywidget.color = [1,1,1,0.5] ; // rend notre objet translucide
}
```

Lorsque notre pointeur quitte la fenêtre *jit.window*, la propriété *eventname* de notre rappel sera "*mouseidleout*". Tout ce que nous faisons est de changer l'alpha à 0,5, laissant la sphère où elle était.

```
}
// ne pas permettre à cette fonction d'être appelée depuis Max
thecallback.local = 1 ;
```

Comme notre fonction *thecallback()* est destinée à être exécutée par notre objet *JitterListener*, nous mettons sa propriété locale à **1** pour éviter qu'elle ne soit exécutée directement par un message Max envoyé à l'objet *js*.

Dessiner la scène

Maintenant que nous avons vu comment répondre aux événements de la souris et mettre à jour la position, la couleur et la forme de notre objet *mywidget* en conséquence, nous devons examiner comment nous effectuons le rendu réel de notre scène en réponse à un message **bang** de l'objet

qmetro dans notre patch Max. La fonction *bang()* de notre code JavaScript gère le dessin ainsi que l'animation des petites sphères en réponse à la nouvelle position de notre grande sphère.

• Regardez le code de la fonction *bang()*.

La majeure partie de notre fonction *bang()* parcourt les positions de toutes les petites sphères et les compare à la position de la grande sphère. Si la distance entre elles est suffisamment faible pour que les sphères se chevauchent lorsqu'elles sont dessinées, nous déplaçons la petite sphère de la quantité minimale nécessaire pour que les sphères se touchent. Ce type de traitement est connu sous le nom de ***détection de collision*** et constitue une technique omniprésente dans le domaine de l'animation par ordinateur.

```
function bang()
// boucle de dessin principale... déterminer quelles petites sphères déplacer
// et piloter le moteur de rendu
{
  // Bloc de détection de collision. Nous devons itérer à travers
  // les petites sphères et vérifier leur distance par rapport à l'objet de contrôle.
  // Si nous chevauchons, nous éloignons la petite sphère
  // selon l'angle de contact correct.
  for(var i = 0;i<OBJECT_COUNT;i++) {
```

Nous entrons dans une boucle JavaScript *for()* pour vérifier la position de notre grande sphère par rapport à chaque petite sphère, une par une.

```
// distance cartésienne le long des axes x et y
var distx = mywidget.position[0] - mysphere[i].position[0]
var disty = mywidget.position[1] - mysphere[i].position[1]

// distance polaire entre les deux objets
var r = Math.sqrt(distx * distx + disty * disty)
// angle de la petite sphère autour de l'objet de contrôle
var theta = Math.atan2(disty, distx)
```

En soustrayant les positions x et y des deux sphères (grande et petite) l'une de l'autre, nous obtenons leur distance cartésienne. Nous pouvons la convertir en distance polaire (absolue) en appliquant le théorème de Pythagore pour obtenir l'hypoténuse (stockée dans la variable r). Nous obtenons ensuite l'angle de la petite sphère autour de la grande en prenant l'arc de la tangente de la montée (x) sur la descente (y).

```
// vérifier la collision...
if(r<0.15)
// l'objet de contrôle est de taille 0.1, les petites sphères sont de taille 0.05,
// donc moins de 0.15 et c'est une collision...
{
```

Comme notre grande sphère a un attribut d'échelle de 0,1 0,1 0,1 et que nos petites sphères ont des attributs d'échelle de 0,05 0,05 0,05, nous pouvons en déduire que les objets se chevauchent si la distance entre eux est inférieure à 0,15. Cela déclenche un bloc de code qui traite la collision.

```
// convertir polar->cartesian pour calculer le déplacement x et y
   var movex = (0.15-r)*Math.cos(theta) ;
   var movey = (0.15-r)*Math.sin(theta) ;

   // décale la petite sphère vers la nouvelle position,
   // qui devrait être juste après le contact avec l'angle de contact
   // l'angle de contact que nous avions auparavant. Le résultat
   // devrait donner l'impression que nous l'avons "poussée"...
   mysphere[i].position = [mysphere[i].position[0]-movex, mysphere[i].position[1]-movey] ;
}
```

Nous utilisons la distance et l'angle polaires, combinés à la distance minimale autorisée entre les objets (0,15), pour déterminer de combien nous devons déplacer la petite sphère le long des axes x et y pour l'empêcher de chevaucher la grande sphère. Nous calculons ces valeurs en reconvertissant les coordonnées polaires en valeurs cartésiennes stockées dans les variables *movex* et *movey*. Nous soustrayons ensuite les valeurs de ces deux variables de la position actuelle de la petite sphère, afin de la repousser hors du chemin.

```
// bloc de rendu...
  myrender.erase() ; // efface le contexte de dessin
  myrender.drawclients() ; // dessine les objets clients
  myrender.swap() ; // remplace le nouveau dessin par un autre
}
```

Ce dernier bloc de code effectue le rendu proprement dit. Comme vous le feriez en travaillant avec des objets OpenGL Jitter dans un patch Max, vous envoyez un message d'effacement à l'objet *jit.gl.render* (*myrender*). La méthode *drawclients()* de *jit.gl.render* collecte toutes les informations pertinentes des objets OpenGL attachés à notre contexte de dessin et les dessine ; tout objet OpenGL dont l'attribut automatique est réglé sur 0 devra être dessiné manuellement ici. La méthode *swap()* remplace ensuite l'ancien rendu par le nouveau dans l'objet *jit.window*, nous montrant ainsi la scène mise à jour. Les méthodes *drawclients()* et *swap()* sont combinées en une seule opération lorsque vous envoyez un message **bang** à un objet *jit.gl.render* dans un patch Max.

• De retour dans le patch de Tutoriel, cliquez sur le *toggle* attaché à la boîte de *message* intitulée *fullscreen $1* (ou appuyez sur la touche *escape* de votre clavier d'ordinateur). L'objet *jit.window* remplira l'écran. Remarquez que la conversion des coordonnées qui vous permet de déplacer la grande sphère à l'écran avec la souris fonctionne toujours. Vous pouvez sortir l'objet *jit.window* du mode plein écran en appuyant sur la touche *escape* de votre clavier. Dans le code JavaScript, regardez la fonction *fullscreen()* :

```
function fullscreen(v) {
  // fonction permettant de basculer la [jit.window] en mode plein écran
  mywindow.fullscreen = v
}
```

L'attribut *fullscreen()* d'un objet *JitterObject jit.window* se comporte exactement comme le message *fullscreen* envoyé à un objet *jit.window* dans un patch Max.

Résumé

Un patch complet d'OpenGL Jitter peut être encapsulé en JavaScript en instanciant *jit.gl.render*, *jit.window*, et d'autres objets OpenGL dans du code procédural écrit pour l'objet *js*. Tous les messages et attributs de ces objets sont exposés sous forme de méthodes et de propriétés correspondantes. Vous pouvez utiliser un objet *JitterListener* pour répondre aux événements déclenchés par un objet Jitter dans JavaScript. Le *JitterListener* exécute alors une fonction de rappel, en lui passant le message d'appel comme argument. Cela vous permet d'écrire des fonctions JavaScript pour répondre à l'interactivité de la souris dans un objet *jit.window*, à la lecture d'un fichier dans un objet *jit.movie*, et à d'autres situations où vous voudriez répondre à un événement déclenché par un message envoyé par un objet Jitter à sa sortie d'état dans un patch Max.