45-Introduction à l'utilisation de Jitter dans JavaScript

L'objet Max *js*, introduit dans Max 4.5, nous permet d'utiliser du code procédural écrit en langage JavaScript dans Max. En plus d'implémenter le langage JavaScript 1.5 de base, l'objet Max *js* contient un certain nombre d'objets et de méthodes supplémentaires spécifiques à Max, par exemple l'objet Patcher (conçu pour s'interfacer avec le patch Max) ou la méthode post() (pour imprimer des messages dans la console Max). Il existe un certain nombre d'extensions à l'objet *js* qui nous permettent d'exécuter des fonctions Jitter directement à partir du langage JavaScript lorsqu'on travaille avec Jitter. Par exemple, les extensions Jitter de *js* nous permettent de:

• Instancier des objets Jitter directement en JavaScript et créer des chaînes de fonctions de processus Jitter aiu sein du code procédural.
• Créez des matrices Jitter avec JavaScript et accédez et définir les valeurs et paramètres des matrices Jitter à partir des fonctions JavaScript.
• Utilisez les opérations de la bibliothèque de Jitter (par exemple, les opérateurs jit.op et les fonctions de base jit.bfg) pour effectuer des opérations matricielles rapides sur les matrices Jitter en JavaScript afin de créer des systèmes de traitement Jitter de bas niveau.
• Recevoir des rappels d'objets Jitter en les écoutant et en appelant des fonctions basées sur les résultats (par exemple, déclencher une nouvelle fonction chaque fois qu'un film tourne en boucle).

Avant de commencer ce didacticiel, vous devez revoir les bases de l'utilisation de JavaScript dans Max en consultant les didacticiels JavaScript à partir de *JavaScript de base* et les *scripts JavaScript*. Ces didacticiels couvrent les bases de l'instanciation et du contrôle d'une chaîne de fonctions d'objets Jitter dans le code JavaScript *js*.

• Ouvrez le patch du didacticiel.

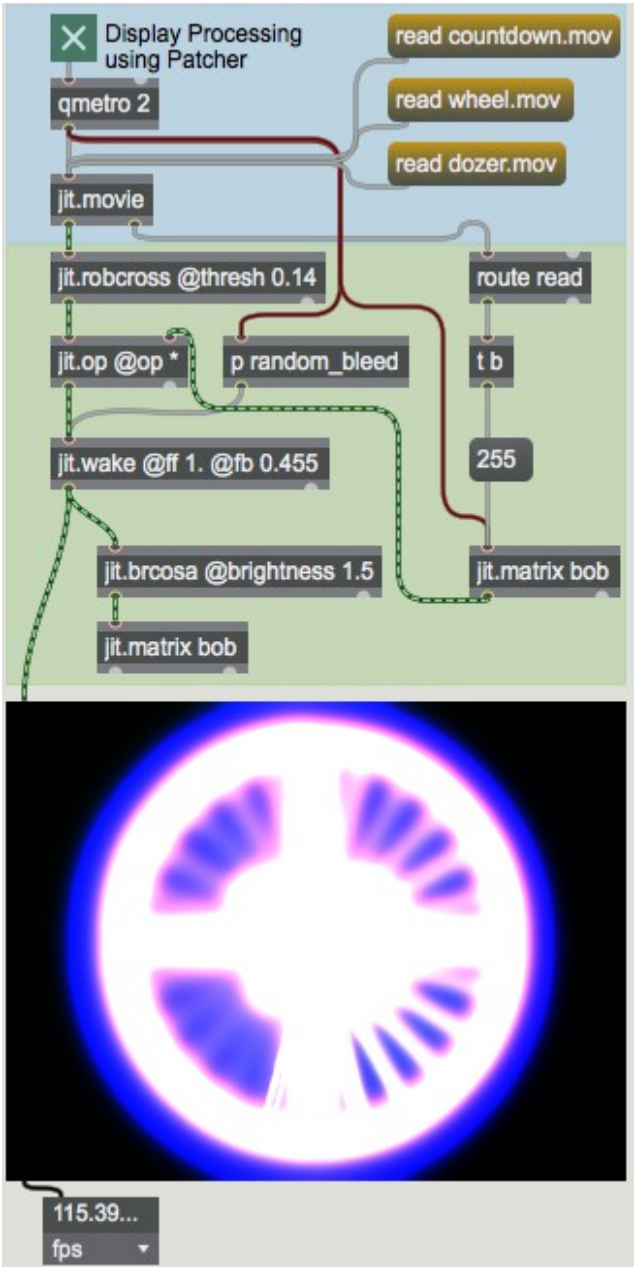
Le patch du didacticiel nous montre deux colonnes d'objets Max côte à côte. La colonne de droite contient un patch qui traite un film via un effet et l'affiche. Tout ceci est réalisé à l'aide d'objets Jitter connectés dans le patch Max entre l'objet *qmetro* et l'objet *jit.pwindow*. La colonne de gauche affiche les objets *qmetro* et *jit.pwindow*, mais ne contient qu'un objet *js* chargeant le fichier JavaScript ***45jWakefilter.js*** entre les deux. Comme nous allons l'apprendre, les deux côtés du patch font à peu près la même chose. Tout d'abord, nous allons examiner le patch sur le côté droit pour voir ce qui se passe.

Jit.waking

• Sur le côté droit du patch, cliquez sur le *toggle* intitulé ***Display Processing using Patcher***. Cliquez sur la boîte de *message* qui lit **read countdown.mo**v, également sur le côté droit.

> Nous utilisons des objets *qmetro* au lieu des objets *metro* dans notre patch en raison du rsique de retard du programmateur lorsqu'on travaille avec JavaScript. Le comportement normal de l'objet Max *js* est de créer une file d'attente d'événements en instance pendant qu'il exécute l'évènement en cours; par conséquent, un objet *metro* qui est rapide accumulera rapidement un grand nombre de messages **bang** à traiter par l'objet *js*. L'objet *qmetro* envoie les messages **bang** à l'arrière de la file d'attente de faible priorité où ils peuvent être utilisés par les messages suivants. Voir *Tutoriel 16*: ***Utilisation de matrices de Jitter nommées*** pour une discussion plus approfondie sur ce sujet.

• Le film devrait apparaître dans la fenêtre *jit.pwindow* avec un effet de couleur changeant progressivement.



Une chaîne d'effets vidéo Jitter typique.

Ce côté du patch lit un film (en utilisant un objet *jit.movie*) dans un objet de détection de bord (*jit.robcross*), qui est ensuite multiplié par la sortie d'une matrice nommée appelée **bob** (en utilisant *jit.op*). La matrice produite par *jit.op* est ensuite traitée par un objet *jit.wake*, qui applique une rétroaction temporelle et une convolution spatiale à la matrice, dont les paramètres peuvent être contrôlés indépendamment pour chaque plan. La sortie de l'objet *jit.wake* est ensuite légèrement éclaircie (avec un objet jit.brcosa), puis stockée à nouveau dans notre matrice nommée (**bob**). La sortie de notre chaîne d'effets telle que vue dans *jit.pwindow* est la sortie de l'objet *jit.wake*.

> La technique consistant à utiliser des matrices Jitter nommées pour la rétroaction est traitée dans le *didacticiel 17*: ***Feedback utilisant des matrices nommées***. L'objet *jit.robcross* applique l'algorithme de détection des bords Robert's Cross (un objet similaire qui nous permet d'utiliser deux autres algorithmes s'appelle *jit.sobel*). L'objet *jit.wake* contient une matrice de rétroaction interne qui est utilisée en conjonction avec la convolution d'image pour créer une variété d'effets de mouvement et de flou spatial (des effets similaires pourraient être construits en utilisant des objets tels que *jit.slide* et *jit.convolve*).

• Ouvrez l'objet *patcher* nommé **random_bleed**.

La clé de la variation de notre algorithme de traitement est ce sub-patch, contenant douze objets aléatoires qui contrôlent différents paramètres de l'objet *jit.wake* dans la chaîne de traitement principale. La sortie de ces objets aléatoires est mise à l'échelle pour nous (avec les objets *scale*) pour convertir nos nombres aléatoires entiers (**0** à **999**) en valeurs à virgule flottante dans la plage de **0** à **0,6**. Ces valeurs sont ensuite lissées avec les deux objets ***** et l'objet **+,** implémentant un simple filtre unipolaire:

$yn = 0{,}01x_n + 0{,}99y_{n-1}$.

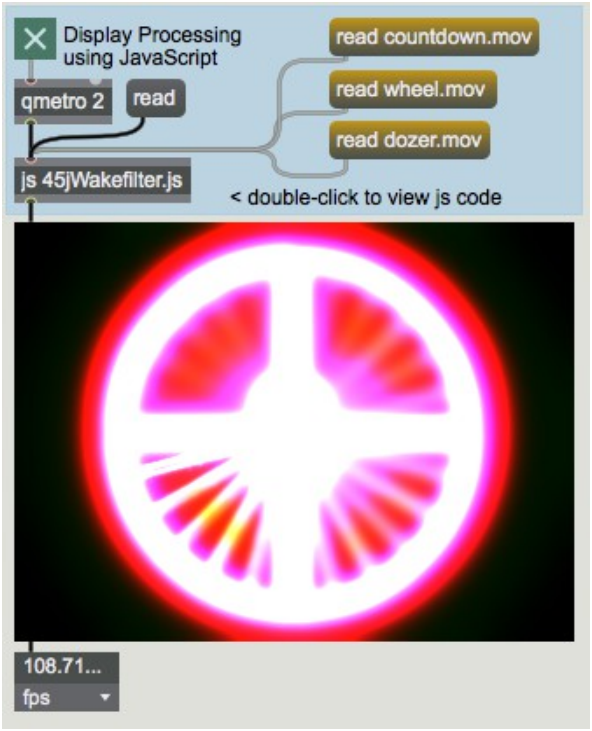
Ces valeurs lissées définissent ensuite les attributs de *jit.wake* qui contrôlent l'ampleur de la perte dans différentes directions (haut, bas, gauche, droite) dans différents plans (spécifiés comme les canaux de couleur rouge, vert et bleu). Vous remarquerez que l'algorithme de lissage est tel que les valeurs de tous les objets de la boîte de *nombre* montrant la sortie lissée ont tendance à osciller autour de 0,3 (ou la moitié de 0,6). Notre algorithme de Jitter présente un décalage de couleur variant lentement (aléatoire) en raison des différences infimes entre ces ensembles d'attributs.

• De retour dans le patch principal du didacticiel, essayez de changer de film en cliquant sur les objets de la boîte de *messages* qui lisent **wheel.mov** et **dozer.mov**. Comparez les effets sur ces deux films avec l'effet sur le film "countdown". Notez que lorsque nous lisons de nouvelles séquences vidéo, nous initialisons la matrice **bob** pour qu'elle contienne toutes les valeurs de **255** (l'effaçant effectivement en blanc).

La méthode Javascript

• Désactivez l'objet *qmetro* sur le côté droit du patch en cliquant sur le *toggle* situé au-dessus. Activez l'objet *qmetro* sur le côté gauche du patch en cliquant sur le *toggle* qui lui est rattaché.

Cliquez sur la boîte de *message* qui indique **countdown.mov** sur le côté gauche du patch.



ça vous semble familier?

La vidéo sur le côté gauche du patch semble étonnamment similaire à celle affichée sur le côté droit. Cela est dû au fait que l'objet *js* sur le côté gauche du patch contient tous les objets et instructions nécessaires pour lire notre film et effectuer le traitement matriciel de notre effet.

• Double-cliquez sur l'objet *js* dans notre patch de Tutoriel. Un éditeur de texte apparaît, contenant le code source de l'objet *js* dans le patch. Le code est enregistré dans un fichier appelé «45jWakefilter.js» dans le même dossier que le patch du didacticiel.

Notre code JavaScript contient le bloc de commentaires similaire en haut, décrivant le fichier, suivi d'un bloc de code global (exécuté lorsque l'objet *js* est instancié) suivi d'un certain nombre de fonctions que nous avons définies, dont la plupart répondent à divers messages envoyé dans l'objet *js* par le patcher Max.

Création de matrices

• Regardez le code du bloc global (c'est-à-dire le code avant d'arriver à la fonction bang ()).

Notre code commence par l'énoncé familier du nombre d'entrées et de sorties que nous aimerions dans notre objet *js*:

```
// inlets and outlets
inlets = 1;
outlets = 1;
```

Ensuite, nous avons un certain nombre de relevés que nous n'avons peut-être jamais vus auparavant:

```
/ Jitter matrices to work with (declared globally)
var mymatrix = new JitterMatrix(4, "char", 320, 240);
var mywakematrix = new JitterMatrix(4, "char", 320, 240);
var myfbmatrix = new JitterMatrix(4, "char", 320, 240);

// initialize feedback matrix to all maximum values
myfbmatrix.setall(255, 255, 255, 255);
```

Ce bloc de code définit que nous allons travailler avec un certain nombre de matrices Jitter dans notre objet *js*. Les variables ***mymatrix***, ***mywakematrix*** et ***myfbmatrix*** sont définies comme des instances de l'objet JitterMatrix, de la même manière que nous déclarons un tableau, une tâche ou une instance de l'objet *sketch jsui*. Les arguments de nos nouveaux objets JitterMatrix sont exactement les mêmes que ceux utilisés pour un objet *jit.matrix*, c'est-à-dire un **name** optionnel, un **planecount**, un **type** et une liste de valeurs pour le **dim**.

> Il est important de ne pas confondre l'attribut **name** d'une matrice Jitter avec le nom de la ***variable*** qui le représente dans le code JavaScript. Par exemple, nous avons créé un objet JitterMatrix dans notre code affecté à la variable ***mymatrix***. L'envoi du message **jit_matrix mymatrix** à la fenêtre *jit.pwindow* dans notre patch Max n'affichera cependant pas cette matrice. Notre objet ***mymatrix*** a une propriété de nom qui est générée automatiquement si elle n'est pas fournie en utilisant la même convention que celle utilisée dans d'autres objets Jitter (par exemple ***uxxxxxxxxx***). La distinction est similaire à celle employée par l'objet JavaScript Global utilisé pour partager des données avec les patchs Max.

Nos trois objets JitterMatrix sont créés avec la même typologie. A la quatrième ligne de cette partie de notre code, nous prenons l'objet JitterMatrix ***myfbmatrix*** et fixons toutes ses valeurs sur **255**. La

méthode setall () de l'objet JitterMatrix le fait pour nous, tout comme le ferait le message **setall** pour un objet *jit.matrix* . En fait, tous les messages et attributs utilisés par l'objet *jit.matrix* sont exposés comme méthodes et propriétés de l'objet JitterMatrix dans JavaScript. Quelques exemples:

```
// mettez toutes les valeurs de notre matrice à 0:
mymatrix.clear ();
// définissez la variable foo sur la valeur de la cellule (40,40):
var foo = mymatrix.getcell (40,40);
// définissez la cellule (30,20) sur les valeurs (255,255,0,0):
mymatrix.setcell2d (30,20,255,255,0,0);
```

La méthode setcell2d () nous permet de définir une valeur d'une seule cellule dans une matrice en utilisant un tableau de valeurs où les deux premiers arguments sont supposés être la position dans la matrice. La cellule est ensuite définie par les valeurs contenues dans les arguments suivants. Il existe également des fonctions utilitaires pour les matrices unidimensionnelles et tridimensionnelles (setcell1d () et setcell3d (), respectivement). Pour une solution générale, nous pouvons utiliser la fonction plain setcell () comme nous le ferions dans un message Max, par ex. mymatrix.setcell (20, 30, "val", 0, 0, 255, 255).

Création d'objets

• Continuons à parcourir le bloc global. Maintenant que nous avons créé des matrices avec lesquelles travailler, nous devons créer des objets pour les manipuler.

```
// Jitter objects to use (also declared globally)
var myqtmovie = new JitterObject("jit.qt.movie", 320, 240);
var myrobcross = new JitterObject("jit.robcross");
var mywake = new JitterObject("jit.wake");
var mybrcosa = new JitterObject("jit.brcosa");
```

Ces quatre lignes créent des instances des objets JitterObject. Nous avons besoin de quatre d'entre eux (***myqtmovie***, ***myrobcross***, ***mywake*** et ***mybrcosa***) correspondant aux quatre objets équivalents sur le côté droit de notre patch Max (*jit.movie*, *jit.robcross*, *jit.wake* et *jit.brcosa*). Ces objets JitterObject se comportent exactement comme les objets Jitter équivalents dans un patch Max, comme nous le verrons un peu plus loin. Le premier argument lorsque nous instancions un JitterObject est la classe d'objet Jitter que nous souhaitons charger (par exemple, "jit.movie" nous donnera un objet *jit.movie* chargé en JavaScript). D'autres arguments peuvent être passés à l'objet comme ils le feraient dans les patchs Max, afin que nous puissions dire à notre nouveau JitterObject *jit.movie* d'avoir un **dim** de 320x240 en fournissant ces valeurs comme arguments.

De la même manière que nous initialiserions des attributs en les tapant dans la boîte d'objet suivant le nom de l'objet (par exemple *jit.brcosa* **@saturation 1.1**), nous pouvons utiliser notre code JavaScript global pour initialiser les attributs des objets JitterObject que nous avons créés:

```
myrobcross.thresh = 0.14; // set edge detection threshold
mywake.rfb = 0.455; // set wake feedback for red channel
mywake.gfb = 0.455; // set wake feedback for green channel
mywake.bfb = 0.455; // set wake feedback for blue channel
mybrcosa.brightness = 1.5; // set brightness for feedback stage
```

Notez que les propriétés d'un JitterObject correspondent directement aux attributs utilisés par l'objet Jitter chargé dans celui-ci, par ex. un JitterObject chargeant un objet *jit.brcosa* aura des propriétés pour la luminosité, le contraste et la saturation. Dans notre code ci-dessus, nous initialisons la propriété **thresh** du JitterObject ***myrobcross*** à **0,14**, reflétant l'objet *jit.robcross* sur le côté droit de notre patch. De la même manière, nous initialisons les attributs de nos objets ***mywake*** et ***mybrcosa***.

Fonctions JavaScript appelant les méthodes de l'objet Jitter

• Regardez le code de la fonction read (). Cette fonction est appelée lorsque notre objet *js* reçoit le message **read**.

```
function read(filename) // read a movie
{
	if(arguments.length==0) {
	// no movie specified, so open a dialog
		myqtmovie.read();
	}
	else { // read the movie specified
		myqtmovie.read(filename);
	}
	// initialize feedback matrix to all maximum values
	myfbmatrix.setall(255, 255, 255, 255);
}
```

Notre fonction read () analyse les arguments du message **read** envoyé à notre objet *js*. Si aucun argument n'apparaît, elle appelle la méthode read () de notre objet ***myqtmovie*** sans argument. Si un argument est spécifié, notre objet ***myqtmovie*** sera invité à lire cet argument sous forme de fichier.

• Cliquez sur la boîte de *message* étiquetée **read** sur le côté gauche du patch. Remarquez qu'une boîte de dialogue s'affiche, comme si vous aviez envoyé un message **read** à un objet *jit.movie* dans un patch Max. Annulez la boîte de dialogue ou chargez un nouveau film pour voir ce que notre algorithme lui fait subir.

Si nous l'avions voulu, nous aurions pu regarder le tableau retourné par la méthode read () pour nous assurer qu'elle n'avait pas échoué. Pour l'instant, cependant, nous nous contenterons de croire que les arguments du message **read** envoyé à notre objet *js* sont des noms de fichiers légitimes de films dans le chemin de recherche.

Après avoir lu notre film (ou demandé à notre objet ***myqtmovie*** d'ouvrir une boîte de dialogue *jit.movie* "Open Document"), nous initialisons à nouveau notre JitterMatrix ***myfbmatrix*** à des valeurs de **255**.

La routine Perform

De la même manière qu'une chaîne de traitement Jitter typique pourrait partir de *jit.movie* pour sortir une série d'objets Jitter en réponse à un *qmetro*, notre algorithme JavaScript Jitter effectue une boucle de son algorithme de traitement (sortie d'une seule matrice) en réponse à un **bang** provenant d'une une source extérieure.

• Observez la fonction bang () dans notre code JavaScript. Remarquez que, tout comme dans notre patch Max, chaque JitterObject est appelé en séquence, traitant les matrices à tour de rôles.

```
function bang() {
  // Effectue une itération de la boucle de lecture/traitement.
  // configuration

  // calcule les coefficients de bleed pour la nouvelle matrice :
  calccoeffs()

  // traitement

  // récupère la nouvelle matrice à partir du film ([jit.movie]) :
  myqtmovie.matrixcalc(mymatrix, mymatrix)

  // Détection des bords ([jit.robcross]) :
  myrobcross.matrixcalc(mymatrix, mymatrix)

  // multiplie avec la sortie précédente (éclaircie)
  mymatrix.op("*", myfbmatrix)

  // traiter l'effet de réveil (ne peut pas être traité sur place) ([jit.wake]) :
  mywake.matrixcalc(mymatrix, mywakematrix)

  // éclaircissement et copie dans la matrice de rétroaction ([jit.brcosa]) :
  mybrcosa.matrixcalc(mywakematrix, myfbmatrix)

  // sortie de la matrice traitée dans Max
  outlet(0, "jit_matrix", mywakematrix.name)
```

La fonction calccoeffs () appelée en premier dans la fonction bang () définit les propriétés de notre objet ***mywake*** (plus de'informations à ce sujet ci-dessous). Elle est suivie de la chaîne de traitement des objets Jitter qui prennent une nouvelle matrice de notre objet ***myqtmovie*** et la transforment. La méthode matrixcalc () d'un JitterObject est l'équivalent de l'envoie d'un objet Jitter dans Max via un **bang** (dans le cas d'objets Jitter qui génèrent des matrices) ou d'un message **jit_matrix** (dans le cas des objets Jitter qui traitent ou affichent des matrices). Les arguments de la méthode matrixcalc () sont la matrice d'entrée suivie de la matrice de sortie. Notre objet ***myqtmovie*** possède un argument redondant pour sa matrice d'entrée qui est ignoré; nous fournissons simplement le nom d'une JitterMatrix valide. Si nous travaillions avec un objet Jitter qui nécessite plus d'une entrée ou d'une sortie (par exemple *jit.xfade*), nous fournirons à notre méthode matrixcalc () des tableaux de matrices entre crochets ([,]).

La méthode op () d'un objet JitterMatrix est l'équivalent de l'exécution de la matrice via un objet *jit.op*, avec des arguments correspondant à l'attribut **op** et le scalaire (**val**) ou la matrice à agir comme deuxième opérande. Sous une forme narrative, les choses suivantes se passent donc dans la section "processus" de notre fonction bang ():

• Notre objet *myqtmovie* génère une nouvelle matrice à partir de l'image actuelle du fichier vidéo chargé, et la stocke dans la JitterMatrix ***mymatrix***.
• Notre objet *myrobcross* prend l'objet mymatrix et effectue une détection de bord sur celui-ci, en stockant les résultats dans la même matrice (plus d'informations à ce sujet ci-dessous).
• Nous multiplions ensuite notre JitterMatrix *mymatrix* avec le contenu de *myfbmatrix* en utilisant la méthode op () de mymatrix. Cette multiplication est effectuée "en place" comme à l'étape précédente.

• Nous traitons ensuite la JitterMatrix *mymatrix* à travers notre objet *mywake*, en stockant la sortie dans un troisième JitterMatrix, appelé *mywakematrix*.
• Enfin, nous éclaircissons la JitterMatrix *mywakematrix*, en stockant la sortie dans *myfbmatrix* pour être utilisée lors de la prochaine itération de la fonction bang (). Dans notre code JavaScript, donc, la matrice *myfbmatrix* est utilisée exactement comme la matrice nommée **bob** a été utilisée dans notre patch Max.

> Note technique: Selon la classe de l'objet Jitter chargé, un JitterObject peut être capable d'utiliser la même matrice pour son entrée et sa sortie dans sa méthode matrixcalc (). Cette utilisation du traitement «en place» permet de conserver le temps de traitement et la mémoire en copiant les données dans de nouvelles matrices intermédiaires. Le fait que cela marche dépend entièrement du fonctionnement interne de l'objet Jitter en question; par exemple, un objet *jit.brcosa* se comportera correctement, alors qu'un objet *jit.wake* (parce qu'il dépend de ses matrices de sortie précédentes pour effectuer le feedback) ne le fera pas. De la même manière, la méthode op () d'un objet JitterMatrix effectuera également son traitement "en place".

Notre matrice traitée (la sortie de l'objet *mywake* stocké dans la matrice *mywakematrix*) est ensuite envoyée au patcher en utilisant une fonction outlet ():

```
// output processed matrix into Max
	outlet(0, "jit_matrix", mywakematrix.name);
}
```

Nous utilisons la propriété **name** de notre JitterMatrix dans cet appel pour envoyer le nom de cette matrice (***uxxxxxxxxx***) à l'objet *receive* dans le patch Max.

Autres fonctions

• Jetez un coup d'œil à la fonction calccoeffs () dans notre code JavaScript. Cette fonction est appelée en interne par un **bang** () à chaque fois qu'elle s'exécute.

```
function calccoeffs() // calcule les 12 coefficients de bleed pour l'état de convolution de l'objet [jit.wake]
{
	// red channel
	mywake.rupbleed*=0.99;
	mywake.rupbleed+=Math.random()*0.006;
	mywake.rdownbleed*=0.99;
	mywake.rdownbleed+=Math.random()*0.006;
	mywake.rleftbleed*=0.99;
	mywake.rleftbleed+=Math.random()*0.006;
	mywake.rrightbleed*=0.99;
	mywake.rrightbleed+=Math.random()*0.006;

	// green channel
	mywake.gupbleed*=0.99;
	mywake.gupbleed+=Math.random()*0.006;
	mywake.gdownbleed*=0.99;
	mywake.gdownbleed+=Math.random()*0.006;
	mywake.gleftbleed*=0.99;
	mywake.gleftbleed+=Math.random()*0.006;
```

```
        mywake.grightbleed*=0.99;
        mywake.grightbleed+=Math.random()*0.006;

        // blue channel
        mywake.bupbleed*=0.99;
        mywake.bupbleed+=Math.random()*0.006;
        mywake.bdownbleed*=0.99;
        mywake.bdownbleed+=Math.random()*0.006;
        mywake.bleftbleed*=0.99;
        mywake.bleftbleed+=Math.random()*0.006;
        mywake.brightbleed*=0.99;
        mywake.brightbleed+=Math.random()*0.006;
}
calccoeffs.local = 1; // can't call from the patcher
```

Nous voyons que la fonction calccoeffs () duplique littéralement la fonctionnalité du *patch* **random_bleed** sur le côté droit de notre patch. Elle définit une variété de propriétés du JitterObject ***mywake***, correspondant aux divers attributs de l'objet *jit.wake* qu'il contient. Remarquez que nous pouvons utiliser ces propriétés comme des variables ordinaires, en obtenant leurs valeurs et en les définissant. Cela nous permet de changer leurs valeurs en utilisant des opérateurs en place, par exemple:

```
mywake.brightbleed*=0.99;
        mywake.brightbleed+=Math.random()*0.006;
```

Ce code (répété douze fois pour différentes propriétés de l'objet *mywake*) utilise la valeur actuelle de la propriété **rupbleed** de *mywake* comme point de départ, la multiplie par **0,99** et lui ajoute une petite valeur aléatoire (entre **0** et **0,006**).

Résumé

Vous pouvez utiliser du code JavaScript dans Max pour définir des systèmes procéduraux à l'aide de matrices et d'objets Jitter. L'objet JitterMatrix dans *js* vous permet de créer, de définir et d'interroger des attributs de matrices Jitter à partir de JavaScript - la méthode setall () de JitterMatrix, définit toutes ses cellules à une certaine valeur, par exemple. Vous pouvez également appliquer des opérations mathématiques à un JitterMatrix "en place" en utilisant la méthode op (), qui contient l'ensemble complet des opérateurs mathématiques utilisés dans l'objet *jit.op*. Les objets Jitter peuvent être chargés en tant que classes dans l'objet JitterObject. Lors de son instanciation, un JitterObject acquiert des propriétés et des méthodes équivalentes aux messages et attributs de l'objet Jitter. La méthode matrixcalc () d'un JitterObject exécute l'équivalent de l'envoi à l'objet Jitter d'un message **bang** ou un message **jit_matrix**, selon ce qui est pertinent pour cette classe d'objet. Cela vous permet de porter des graphiques de fonctions complexes de processus Jitter en JavaScript.

Dans les deux didacticiels suivants, nous examinerons d'autres façons d'utiliser JavaScript pour étendre les possibilités de travail avec Jitter.