46-Manipulation des données matricielles à l'aide de JavaScript

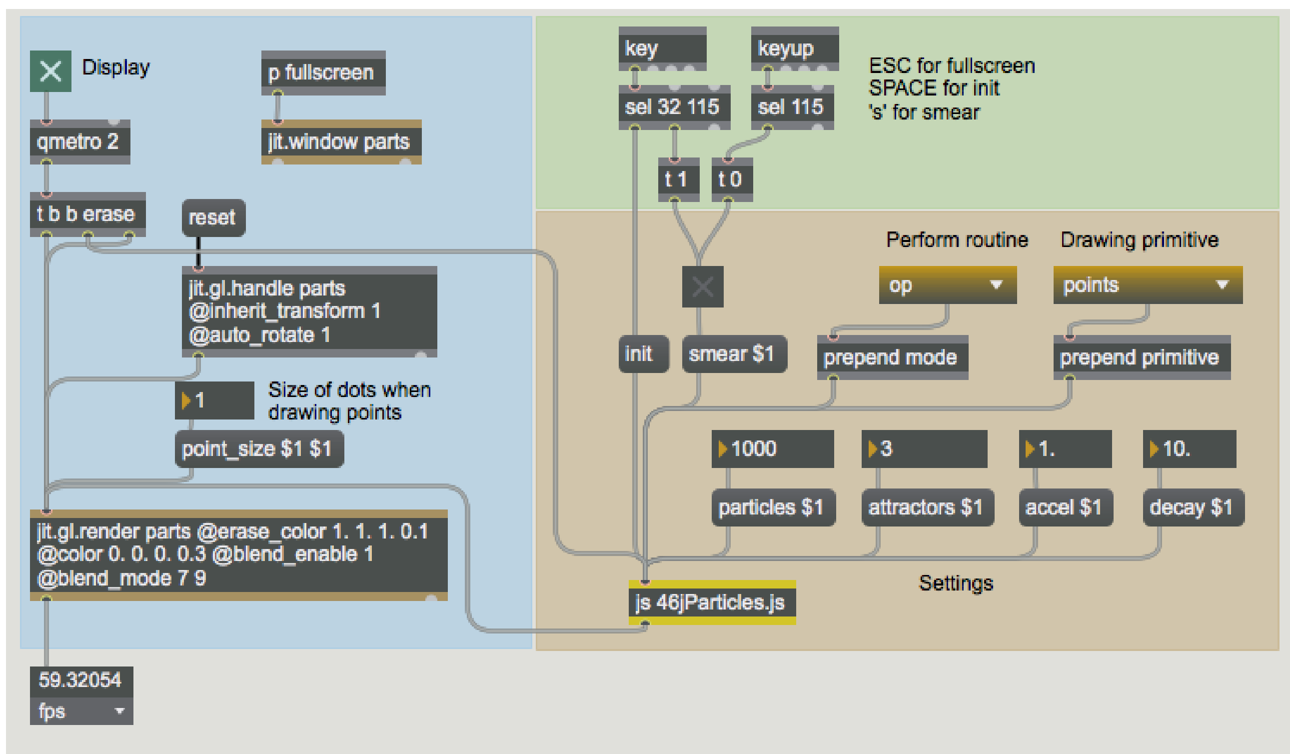
Comme nous l'avons vu dans le dernier tutoriel, nous pouvons utiliser l'objet Max *js* pour concevoir un pipeline d'objets Jitter dans un code JavaScript procédural. Les objets *JitterObject* et *JitterMatrix* en JavaScript nous permettent de créer de nouveaux objets et matrices Jitter et de les utiliser plus ou moins comme nous le ferions dans un patch Max. Dans de nombreuses situations, nous devons manipuler des données stockées dans une matrice Jitter d'une manière qui serait maladroite ou délicate à faire en utilisant un patch Max. Ce tutoriel examine diverses solutions pour manipuler des données matricielles en JavaScript en utilisant les méthodes et les propriétés de l'objet *JitterMatrix*, ainsi qu'en utilisant l'objet *jit.expr* en JavaScript.

Ce tutoriel suppose que vous avez lu le précédent Tutoriel 45 : Introduction à l'utilisation de Jitter dans JavaScript. De plus, ce tutoriel fonctionne avec les objets OpenGL de Jitter ainsi qu'avec *jit.expr*. Vous pouvez aussi revoir le Tutoriel 30 : Dessiner du texte en 3D, le Tutoriel 31 : destinations de rendu, et le Tutoriel 39 : Cartographie spatiale, avant de commencer.

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 tutoriel.

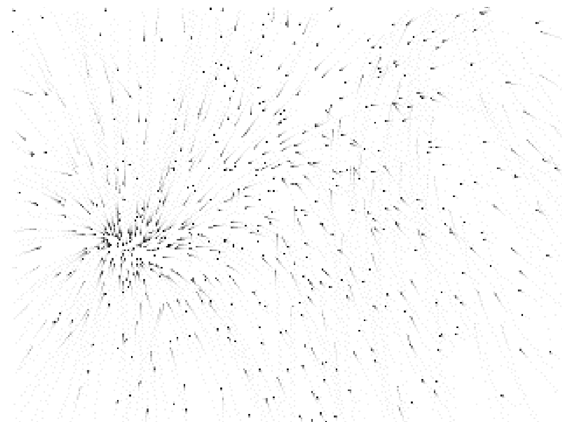
Ce patch de tutoriel utilise un objet *js* chargeant un fichier appelé **46jParticles.js**. Le code JavaScript génère une matrice Jitter en réponse à un **bang** que nous envoyons ensuite à l'objet *jit.gl.render* dans le patch. Le fichier contient un certain nombre de fonctions pour répondre à divers paramètres que nous pouvons renvoyer comme messages depuis notre patch.



Notre patch contenant le fichier JavaScript.

 - Cliquez sur le *toggle* au-dessus de l'objet *qmetro* sur le côté gauche du patch. Observez les résultats dans la fenêtre jit.parts nommée.

Notre code JavaScript génère un ensemble de points qui représentent un système de particules simple. Un système de particules est essentiellement un algorithme qui opère sur un (souvent très grand) nombre de points spatiaux appelés particules. Ces particules ont des règles qui déterminent comment elles se déplacent dans le temps les unes par rapport aux autres ou par rapport à d'autres acteurs dans l'espace. À leur niveau de base, les particules ne contiennent que leurs coordonnées spatiales. Les systèmes de particules peuvent également contenir d'autres informations et peuvent être utilisés pour simuler une grande variété de processus naturels tels que l'eau courante ou la fumée. Les systèmes de particules sont largement utilisés dans les simulations informatiques de notre environnement et constituent à ce titre une technologie de base dans de nombreuses applications d'imagerie générée par ordinateur (CGI).



Un système de particules généré comme une matrice Jitter.

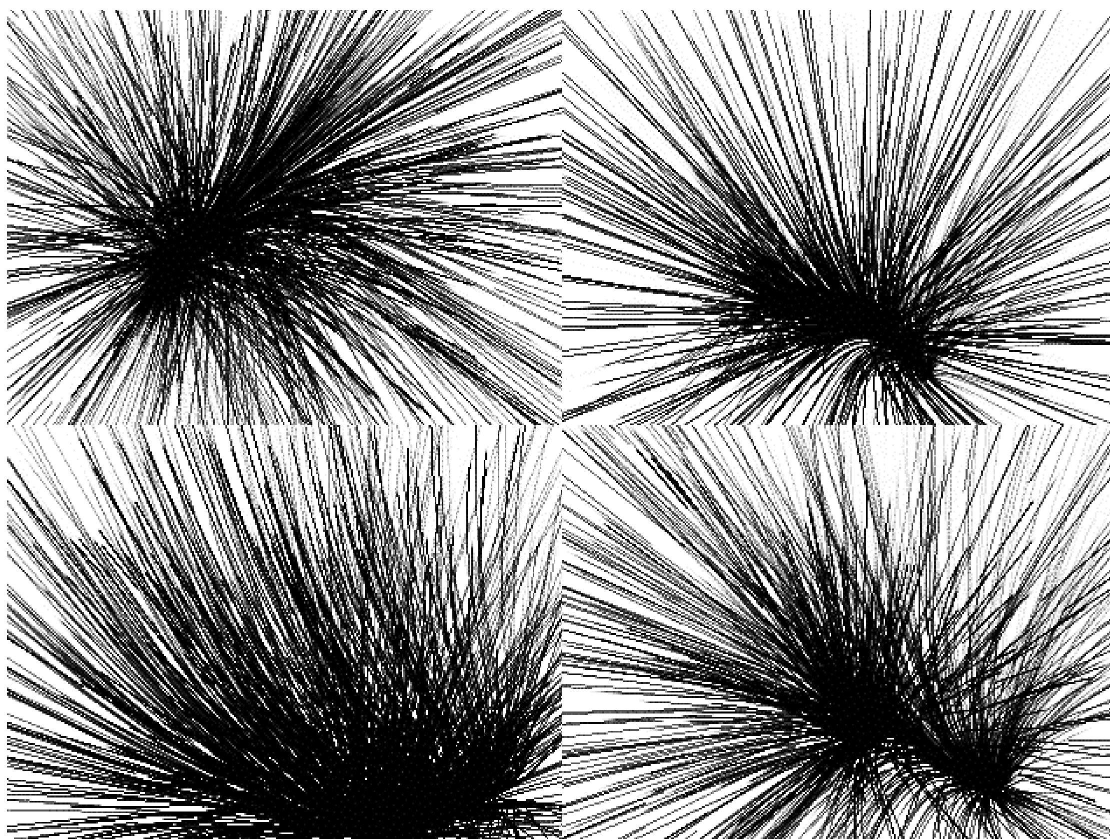
Le vaste monde des particules

Le système de particules utilisé dans notre simulation JavaScript fonctionne en générant deux ensembles de points aléatoires représentant les positions et les vitesses des particules et un certain nombre de positions spatiales dans l'espace 3D qui contiennent la gravité. Ces points de gravité, appelés ***attracteurs***, agissent sur les particules dans chaque image pour attirer progressivement les particules vers eux. Comme nous pouvons le voir dans la fenêtre *jit.window*, les points de notre système de particules se replient progressivement vers un ou plusieurs points de singularité. Par ailleurs, les points attracteurs peuvent être tels que les particules oscillent entre eux, prises dans un champ de gravité conflictuel qui se stabilise progressivement.

 • Avec le patch comme fenêtre la plus en avant, appuyez sur la barre d'espace de votre clavier d'ordinateur ou cliquez sur la boîte de *message* intitulée *init* attachée à l'objet *js*. Essayez de le faire plusieurs fois pour observer les différents comportements de notre système de particules.

Le message *init* de notre objet *js* redémarre le système de particules. Il disperse les particules de manière aléatoire et génère de nouveaux points d'attraction.

 • Appuyez sur la touche "s" du clavier de l'ordinateur et maintenez-la enfoncée, ou cliquez sur le *toggle* attaché à la boîte de *message* intitulée "**smear $1**". Les particules vont maintenant laisser des traces derrière elles lorsqu'elles se déplacent. Relâchez la touche "s" (ou décochez le *toggle*) pour revenir à une visualisation normale.



Une variété de comportements à partir d'un ensemble de règles simples.

• Cliquez dans la fenêtre *jit.window*. Un ensemble d'axes colorés rouge, vert et bleu apparaîtra autour du modèle. Faites tourner le modèle avec votre souris. Essayez de faire un zoom arrière (en maintenant la touche ALT/Option enfoncée et en faisant glisser la souris dans la fenêtre). Redémarrez le système de particules à partir de différents points d'observation.

Sous le capot

Maintenant que nous avons vu une bonne partie de la fonctionnalité du patch (nous y reviendrons plus tard), regardons le code JavaScript pour voir comment l'algorithme est construit.

• Double-cliquez sur l'objet *js* dans notre patch de 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 "**46jParticles.js**" dans le même dossier que le patch du tutoriel.

Notre code JavaScript fonctionne en manipulant nos particules et nos attracteurs comme des matrices Jitter. Le système de particules est mis à jour par une génération chaque fois que notre objet *js* reçoit un **bang**. Ceci est réalisé en effectuant une série d'opérations sur les données dans les matrices Jitter. Nous pouvons profiter de l'architecture Jitter pour effectuer des opérations mathématiques sur des matrices entières en une seule fois puisque nous avons encodé notre système sous forme de matrices. Cela offre un avantage significatif en termes de vitesse, de clarté et d'efficacité par rapport au fait de travailler avec nos données sous forme de valeurs individuelles qui doivent être ajustées une par une, dans de nombreux cas (comme nous le ferions si nous encodions nos particules sous forme de tableau, par exemple).

Notre code JavaScript contient en fait trois manières de mettre à jour notre système de particules de génération en génération, chacune d'entre elles utilisant des techniques différentes pour manipuler les données de la matrice représentant les particules. Nous pouvons traiter le système de particules comme une entité unique en utilisant une série de méthodes op() sur l'objet *JitterMatrix*, en utilisant un objet *jit.expr*, ou en itérant point par point (ou cellule par cellule) dans le système de particules. Nous les examinerons tour à tour après avoir étudié le code qui leur est commun.

• Regardez le bloc de code global qui commence le fichier JavaScript.

Après le bloc de commentaires initial et les déclarations d'entrée/sortie, nous pouvons voir un certain nombre de variables déclarées et initialisées, y compris quelques objets *JitterObject* et *JitterMatrix*. Si nous examinons ce code en détail, nous pouvons voir les grandes lignes de la façon dont nous allons réaliser notre système de particules.

```
var PARTICLE_COUNT = 1000 // nombre initial de sommets de la particule
var ATTRACTOR_COUNT = 3 // nombre initial de points de gravité
```

Ces deux variables globales (PARTICLE_COUNT et ATTRACTOR_COUNT) sont utilisées pour décider du nombre de particules et du nombre de points d'attraction avec lesquels nous souhaitons travailler dans notre simulation. Elles détermineront la taille des matrices Jitter contenant les informations sur les particules et les attracteurs.

```
// créer un objet [jit.noise] pour la génération de particules et de vélocité
var noisegen = new JitterObject("jit.noise")
noisegen.dim = PARTICLE_COUNT
noisegen.planecount = 3
noisegen.type = "float32"

// créer un objet [jit.noise] pour la génération d'attracteur
var attgen = new JitterObject("jit.noise")
attgen.dim = ATTRACTOR_COUNT
attgen.planecount = 3
attgen.type = "float32
```

Nos systèmes de particules sont générés de manière aléatoire par la fonction *init()*, que nous allons étudier maintenant. Les objets *jit.noise* créés ici en tant qu'objets *JitterObject* vont remplir cette fonction pour nous, en générant des matrices unidimensionnelles de valeurs *float32* d'une taille (*dim*) correspondant au nombre de particules et d'attracteurs spécifiés pour notre système. Les matrices générées par nos objets *jit.noise* ont un *planecount* de 3, correspondant aux données spatiales x, y, et z.

```
// créer deux objets [jit.expr] pour la fonction bang_expr()
// première expression : additionner tous les plans de la matrice d'entrée
var myexpr = new JitterObject(" jit.expr ")
myexpr.expr = "in[0].p[0]+in[0].p[1]+in[0].p[2] "
// deuxième expression : évaluer a+((b-c)*d/e)
var myexpr2 = new JitterObject(" jit.expr "
myexpr2.expr = "in[0]+((in[1]-in[2])*in[3]/in[4])"))
```

L'une des façons de mettre à jour notre système de particules est d'utiliser deux objets *jit.expr* dans notre code JavaScript. Cette partie du code crée les objets *JitterObject* et définit l'expression

mathématique qu'ils utiliseront (l'attribut *expr*). Nous verrons tout cela lorsque nous étudierons le code qui les utilise plus tard.

```
// créer les matrices Jitter dont nous avons besoin pour stocker nos données.
// matrice les sommets x,y,z des particules
var particlemat = new JitterMatrix(3, "float32", PARTICLE_COUNT)
// matrice les vélocités x,y,z des particules
velomat = new JitterMatrix(3, "float32", PARTICLE_COUNT)
// matrice les points d'attraction x,y,z (centres de gravité)
var attmat = new JitterMatrix(3, "float32", ATTRACTOR_COUNT)
// matrice des distances globales
var distmat = new JitterMatrix(3, "float32", PARTICLE_COUNT)
// matrice temporaire pour la fonction bang_op()
var tempmat = new JitterMatrix(3, "float32", PARTICLE_COUNT)
// matrice temporaire de sommation pour la fonction bang_op()
var summat = new JitterMatrix(1, "float32", PARTICLE_COUNT)
// une autre matrice de sommation temporaire pour la fonction bang_op()
var summat2 = new JitterMatrix(1, "float32", PARTICLE_COUNT)
// une matrice scalaire pour stocker le point de gravité actuel
var scalarmat = new JitterMatrix(3, "float32", PARTICLE_COUNT)
// une matrice scalaire pour stocker l'accélération (fonction expr_op() uniquement)
var amat = new JitterMatrix(1, "float32", PARTICLE_COUNT)
```

Notre algorithme fait appel à un certain nombre d'objets *JitterMatrix* pour stocker des informations sur notre système de particules et pour être utilisés comme stockage intermédiaire pendant le traitement de chaque génération du système. Les trois premières matrices (liées aux variables *particlemat*, *velomat* et *attmat*) stockent les positions x, y, z de nos particules, les vélocités x, y, z de nos particules et les positions x, y, z de nos attracteurs, respectivement. Les six autres matrices sont utilisées pour calculer chaque génération du système.

```
var a = 0.001 // facteur d'accélération
var d = 0.01 // facteur de décroissance
```

Ces deux variables contrôlent deux aspects importants du comportement de notre système de particules : la variable a contrôle la rapidité avec laquelle les particules accélèrent vers un attracteur, tandis que la variable d contrôle le degré de décroissance de la vélocité actuelle des particules à chaque génération. Cette deuxième variable influence la rapidité avec laquelle une particule peut changer de direction et être attirée vers d'autres attracteurs.

```
var perform_mode = "op" // fonction perform par défaut
var draw_primitive = "points" // primitive de dessin par défaut
```

Ces dernières variables définissent laquelle des trois techniques nous utiliserons pour traiter notre système de particules (le ***perform_mode***) et la manière dont notre objet *jit.gl.render* visualise la matrice de particules une fois que notre code JavaScript l'a envoyée au patch Max (le ***draw_primitive***).

La phase d'initialisation

La première étape de la création d'un système de particules consiste à générer un état initial pour les particules et tous les facteurs qui agiront sur elles (dans notre cas, les points attracteurs). Par exemple, si nous voulions simuler une chute d'eau, nous placerions toutes nos particules en haut de

l'espace, avec un champ de gravité fixe en bas de l'espace agissant sur les particules à chaque génération. Notre système est légèrement moins ambitieux en termes de précision dans le monde réel : les particules et les attracteurs seront simplement placés de manière aléatoire dans la scène 3D.

- Regardez le code des fonctions *loadbang()* et *init()* dans le code JavaScript.

```
function loadbang() {
  // exécute ce code à l'ouverture de notre patch Max
  init() // initialise nos matrices
  post("particles initialized.\n ")
}

function init() {
  // routine d'initialisation... à appeler au chargement, ainsi que
  // lorsque nous changeons le nombre de particules ou d'attracteurs.
  // génère une matrice de particules aléatoires réparties entre -1 et 1
  noisegen.matrixcalc(particlemat, particlemat)
  particlemat.op("*", 2.0)
  particlemat.op("-", 1.0)
  // génére une matrice de vélocités aléatoires réparties entre -1 et 1
  noisegen.matrixcalc(velomat, velomat)
  velomat.op("*", 2.0)
  velomat.op("-", 1.0)
  // génère une matrice d'attracteurs aléatoires répartis entre -1 et 1
  attgen.matrixcalc(attmat, attmat)
  attmat.op("*", 2.0)
  attmat.op("-", 1.0)
})
```

La fonction *loadbang()* d'un objet *js* s'exécute à chaque fois que le patch Max contenant le fichier *js* est chargé. Cela se produit après que l'objet soit instancié avec le reste du patch, et est déclenché en même temps que les messages déclenchés par les objets *loadbang* et *loadmess* dans un patch Max. Notre fonction *loadbang()* appelle simplement la fonction *init()* et imprime ensuite un message amical dans la console Max pour nous dire que tout va bien.

> La fonction *loadbang()* du code JavaScript ne s'exécute que lorsque le patch contenant l'objet *js* est ouvert. Cette fonction ne s'exécute pas lorsque vous modifiez et recompilez le code JavaScript.

Notre fonction *init()* s'exécute lorsque nous ouvrons notre patch ainsi qu'à chaque fois que nous l'appelons depuis notre patch Max (via la boîte de *message* déclenchée par la barre d'espace). La fonction *init()* est également appelée lorsque nous modifions le nombre d'attracteurs et de particules dans notre simulation. La méthode *matrixcalc()* de *jit.noise* remplit la matrice de sortie (le deuxième argument de la méthode) avec des valeurs aléatoires entre **0** et **1**. Cela revient à envoyer un **bang** à un objet *jit.noise* dans un patch. Dans notre fonction *init()*, nous remplissons trois matrices avec des valeurs aléatoires dans 3 plans. Ces matrices représentent la position initiale de nos particules (*particlemat,*) la vélocité initiale de nos particules (*velomat,*) et la position de nos attracteurs (*attmat*). En utilisant la méthode *op()* de nos objets *JitterMatrix*, nous mettons à l'échelle ces valeurs aléatoires pour qu'elles soient comprises entre **-1** et **1**. Pour ce faire, nous multiplions les valeurs de la matrice par **2**, puis nous soustrayons **1**.

Maintenant que nous avons défini l'état initial de notre système de particules, nous devons examiner comment nous traitons les particules à chaque génération. Ceci est accompli par l'une des trois méthodes différentes dans notre code JavaScript, déterminée par la variable ***perform_mode***.

• Dans le patch de Tutorial, redémarrez le système de particules et activez l'objet *umenu* intitulé *Perform routine* de "*op*" à "*expr*". Le système de particules devrait continuer à se comporter exactement comme avant. Basculez à nouveau l'objet *umenu* sur "*iter*". Le système de particules fonctionnera toujours, mais très lentement (notez le débit d'images dans l'objet *jit.fpsgui* attaché à l'objet *jit.gl.render* dans la partie inférieure gauche du patch). Remettez l'*umenu* sur "*op*".
L'*umenu* modifie la valeur de la variable ***perform_mode*** via la fonction *mode()* de notre code JavaScript. Nous y reviendrons plus tard dans ce tutoriel, mais il est important de noter que l'une des méthodes utilisées ("*iter*") fonctionne beaucoup plus lentement que les deux autres. Ceci est largement dû à la technique utilisée pour mettre à jour le système de particules. Nous verrons pourquoi lorsque nous étudierons la fonction qui exécute cette tâche.

• Regardez la fonction ***bang()*** dans le code JavaScript.

```
function bang() {
 // exécute une itération de notre système de particules
 switch (
  perform_mode // choisissez parmi les options suivantes...
 ) {
  case "op" : // utilise les opérateurs de la matrice Jitter
   bang_op()
   pause
  case "expr" : // utilisation de [jit.expr] pour l'essentiel de l'algorithme
   bang_expr()
   pause
  case "iter" : // itère cellule par cellule dans les matrices
   bang_iter()
   break
  par défaut :
   // utilise bang_op() comme valeur par défaut
   bang_op()
   break
 }
 // sort notre nouvelle matrice de sommets de particules
 // avec la primitive de dessin actuelle
 outlet(0, "jit_matrix", particlemat.name, draw_primitive)
}
```

Notre fonction bang() utilise une instruction JavaScript ***switch()*** pour décider de la fonction à appeler à l'intérieur de celle-ci pour effectuer le traitement effectif de notre système de particules. En fonction du ***perform_mode*** que nous choisissons dans le patch Max, nous sélectionnons l'une des trois fonctions différentes (***bang_op()***, ***bang_expr()*** ou ***bang_iter()***). En supposant que tout se passe bien, nous renvoyons dans Max le message *jit_matrix*, suivi du nom de notre matrice *particlemat* (qui contient les coordonnées actuelles des particules de la simulation), suivi du nom de notre *draw_primitive* OpenGL.

Dans la grande tradition de *Choose Your Own Adventure* et *Let's Make a Deal*, nous allons maintenant étudier les trois différentes routines d'exécution représentées par les différentes fonctions mentionnées ci-dessus.

Porte 1 : La route op()

- Regardez la source JavaScript de la fonction ***bang_op()***.

Notre fonction ***bang_op()*** met à jour notre système de particules en utilisant, dans la mesure du possible, la méthode op() de l'objet *JitterMatrix* pour modifier mathématiquement le contenu des matrices en une seule fois. Dans la mesure du possible, nous effectuons ce traitement en place afin de limiter le nombre de matrices Jitter distinctes dont nous avons besoin pour faire fonctionner l'algorithme. Nous effectuons l'essentiel du traitement plusieurs fois dans une boucle for(), une fois pour chaque attracteur de notre système de particules. Une fois cette boucle terminée, nous obtenons une version actualisée de la matrice de vélocité (*velomat*), que nous ajoutons ensuite à la matrice de particules (*particlemat*) pour définir les nouvelles positions des particules.

• En bref, nous faisons ce qui suit :

```
/function bang_op() // créer notre matrice de particules en utilisant les opérateurs matriciels
{
 for(var i = 0 ; i < ATTRACTOR_COUNT ; i++)
 // faire une itération par point de gravité
  {
```

Nous exécutons le code jusqu'à l'accolade de fermeture (}) une fois pour chaque attracteur, en attribuant la variable i à l'attracteur sur lequel nous travaillons actuellement.

```
// créer une matrice scalaire à partir de l'attracteur actuel :
scalarmat.setall(attmat.getcell(i))
```

La méthode ***getcell()*** d'un objet *JitterMatrix* renvoie les valeurs des nombres dans la cellule spécifiée comme argument. La méthode ***setall()*** définit toutes les cellules d'une matrice avec une valeur (ou un tableau de valeurs). Ces méthodes fonctionnent de la même manière que les messages correspondants à l'objet *jit.matrix* dans un patch Max. Cette ligne indique à notre objet *js* de copier les coordonnées des attracteurs actuels de la matrice des attracteurs (*attmat*) et de définir chaque cellule de la *JitterMatrix scalarmat* à ces valeurs. La matrice *scalarmat* a la même dimension que la matrice *particlemat* (égale au nombre de particules dans notre système). Cela nous permet de l'utiliser comme un opérande scalaire dans nos méthodes op().

```
// soustraire nos positions de particules de l'attracteur actuel
// et stocker dans une matrice temporaire (x,y,z) :
tempmat.op("-", scalarmat, particlemat)
```

Ce code soustrait la position de nos particules (*particlemat*) de la position de l'attracteur avec lequel nous travaillons actuellement (*scalarmat*). Le résultat est ensuite stocké dans une matrice temporaire avec la même **dim** que les deux utilisées dans la fonction op(). Cette matrice représente les distances de chaque particule à l'attracteur actuel.

```
// élevé au carré pour créer une matrice de distance cartésienne (x*x, y*y, z*z) :
distmat.op("*", tempmat, tempmat)
```

Ce code multiplie la matrice *tempmat* par elle-même, comme un moyen simple de l'élever au carré. Le résultat est ensuite stocké dans la matrice *distmat*.

```
// additionner les plans de la matrice de distance (x*x+y*y+z*z)
summat.planemap = 0
summat.frommatrix(distmat)
summat2.planemap = 1
summat2.frommatrix(distmat)
summat.op("+", summat, summat2)
summat2.planemap = 2
summat2.frommatrix(distmat)
summat.op("+", summat, summat2)
```

Dans ce bloc de code, nous prenons les plans séparés de la matrice *distmat* et les ajoutons ensemble dans une matrice à plan unique appelée *summat*. Pour ce faire, nous utilisons la propriété *planemap* de *JitterMatrix* pour spécifier quel plan de la matrice source utiliser lors de la copie à l'aide de la méthode *frommatrix()*. Pour faire la somme, nous avons besoin d'une deuxième matrice temporaire (*summat2*) pour faciliter l'opération. Tout d'abord, nous copions le plan 0 de *distmat* (les distances x au carré) en distances z) dans *summat2*. Nous additionnons ensuite à nouveau *summat* et *summat2*, en gardant à l'esprit que *summat* contient déjà à ce stade la somme des deux premiers plans de *distmat*. Le résultat de cette deuxième somme est enregistré dans *summat*, qui contient maintenant la somme des trois plans de *distmat*.

```
// mettre à l'échelle nos distances par la valeur de l'accélération :
tempmat.op("*", a) ;
// divise nos distances par la somme des distances
// pour obtenir la gravité pour cette image :
tempmat.op("/", summat) ;
// ajoutez à la vélocité actuelle des déplacements pour obtenir la
// la quantité de mouvement pour cette image :
velomat.op("+", tempmat) ;
}
```

C'est le dernier bloc de code de notre boucle par attracteur. Nous multiplions la matrice *tempmat* (qui contient les distances de nos particules par rapport à l'attracteur actuel) par la valeur stockée dans la variable **a**, représentant l'accélération. Nous divisons ensuite ce résultat par la matrice *summat* (la somme des distances au carré), et nous ajoutons ces résultats aux vélocités actuelles de chaque particule stockées dans la matrice *velomat*. Le résultat de l'addition est stocké dans *velomat*.

L'ensemble de ce processus est répété pour chaque attracteur. Par conséquent, la matrice *velomat* est ajoutée à chaque fois en fonction de la distance qui sépare nos particules de chaque attracteur. Au moment où la boucle se termine (lorsque **i** atteint le dernier index d'attracteur), *velomat* contient une matrice de vitesses correspondant à l'attraction cumulée de tous nos attracteurs sur toutes nos particules.

```
// décalage de nos positions actuelles par la quantité de mouvement :
  particlemat.op("+", velomat) ;
  // réduit nos vélocités par le facteur de décroissance pour la prochaine image :
  velomat.op("*", d) ;
}
```

Enfin, nous ajoutons ces vélocités à notre matrice de particules (*particlemat* + *velomat*). Notre matrice de particules est maintenant mise à jour avec un nouvel ensemble de positions de particules. Nous diminuons ensuite la matrice de vélocité de la quantité stockée dans la variable **d**, de sorte que la simulation conserve un reste de la vélocité de cette génération pour la génération suivante du

système de particules.

L'utilisation d'une série de méthodes *op()* en cascade pour exécuter notre algorithme sur des matrices entières nous donne un grand avantage en termes de vitesse, puisque Jitter peut exécuter une opération mathématique simple sur un grand ensemble de données très rapidement. Cependant, il y a quelques points (particulièrement dans la génération de la matrice de sommation *summat*) où le code a pu sembler plus maladroit que nécessaire. Nous pouvons utiliser *jit.expr* pour définir une expression mathématique plus complexe pour effectuer une grande partie de ce travail en une seule opération.

Porte n° 2 : la route expr()

• De retour dans le bloc global de notre fichier JavaScript, réexaminez le code qui instancie les objets jit.expr.

```
// créer deux objets [jit.expr] pour la fonction bang_expr()

// première expression : additionner tous les plans de la matrice d'entrée
var myexpr = new JitterObject("jit.expr")
myexpr.expr = "in[0].p[0]+in[0].p[1]+in[0].p[2]"
// deuxième expression : évaluer a+((b-c)*d/e)
var myexpr2 = new JitterObject("jit.expr")
myexpr2.expr = "in[0]+((in[1]-in[2])*in[3]/in[4])"
```

Au début de notre code JavaScript, nous avons créé deux objets *JitterObject* (*myexpr* et *myexpr2*) qui instancient des objets *jit.expr*. L'expression du premier objet prend une seule matrice (*in[0]*) et additionne ses plans (la notation *.p[n]* fait référence aux données stockées dans le plan **n** de cette matrice). La deuxième expression prend cinq matrices (*in[0] - in[4]*) et ajoute la première matrice (*A*) au résultat de la deuxième soustraite de la troisième (*B-C*) multipliée par une quatrième (*D*) et divisée par une cinquième (*E*). Notre *JitterObject myexpr2* permet donc l'expression :

A+((B-C)*D/E)

• Regardez le code de la fonction *bang_expr()*. Comparez-le à ce que nous avons utilisé dans la fonction *bang_op()*.

Le schéma de base de notre fonction *bang_expr()* est équivalent à celui de la fonction *bang_op()*, c'est-à-dire que nous itérons à travers une boucle basée sur le nombre d'attracteurs dans notre simulation, pour finalement aboutir à une matrice de vélocité agrégée (*velomat*) que nous utilisons ensuite pour compenser notre matrice de particules (*particlemat*). La principale différence réside dans l'endroit où nous insérons les arguments à *jit.expr* :

```
function bang_expr() // créer notre matrice de particule en utilisant [jit.expr]
{
 // crée une matrice scalaire à partir de notre valeur d'accélération :
 amat.setall(a) ;
```

La ligne ci-dessus remplit chaque cellule de la matrice *amat* avec la valeur de la variable **a** (le facteur d'accélération). Cela nous permet de l'utiliser plus tard comme opérande dans l'une des expressions *jit.expr*.

```
for(var i = 0 ; i < ATTRACTOR_COUNT ; i++)
// faire une itération par point de gravité
{
  // créer une matrice scalaire à partir de l'attracteur actuel :
  scalarmat.setall(attmat.getcell(i)) ;
  // soustraire nos positions de particules de l'attracteur actuel
  // et stocker dans une matrice temporaire (x,y,z) :
  tempmat.op("-", scalarmat, particlemat) ;
  // élevé au carré pour créer une matrice de distance cartésienne (x*x, y*y, z*z) :
  distmat.op("*", tempmat, tempmat) ;
```

C'est la même chose que dans *bang_op()*. Nous obtenons une matrice de distance au carré basée sur la différence entre l'attracteur actuel et la position de nos particules.

```
// additionner les plans de la matrice de distance (x*x+y*y+z*z) :
// "in[0].p[0]+in[0].p[1]+in[0].p[2]" :
myexpr.matrixcalc(distmat, summat)
```

Au lieu de sommer la matrice *distmat* plan par plan à l'aide des méthodes *op()* et *frommatrix()*, nous évaluons simplement notre première expression mathématique en utilisant *distmat* comme matrice d'entrée à 3 plans et *summat* comme matrice de sortie à 1 plan.

```
// dériver la quantité de mouvement pour cette image :
// "in[0]+((in[1]-in[2])*in[3]/in[4])" :
myexpr2.matrixcalc([velomat, scalarmat, particlemat, amat, summat], velomat)
```

De même, à la fin de notre boucle d'attracteur, nous pouvons dériver la matrice de vélocité *velomat* en une seule expression composée basée sur la matrice de vélocité précédente (*velomat*), la matrice scalaire contenant le point d'attracteur actuel (*scalarmat*), les positions actuelles des particules (*particlemat*), la matrice scalaire contenant l'accélération (*amat*), et la matrice contenant les sommes de distance (*summat*). C'est beaucoup plus simple (et plus propre) que d'utiliser toute une séquence de fonctions *op()* qui travaillent avec des matrices intermédiaires. Notez que nous utilisons des crochets ([et]) pour établir un tableau de matrices d'entrée dans la méthode *matrixcalc()* à l'objet *myexpr2*.

```
// décalage de nos positions actuelles par la quantité de mouvement :
  particlemat.op("+", velomat) ;
  // réduit nos vélocités par le facteur de décroissance pour la prochaine image :
  velomat.op("*", d) ;
}
```

C'est la même chose que dans *bang_op()*. Nous générons les nouvelles positions des particules et décomposons les nouvelles vélocités pour les utiliser comme vélocités initiales dans la prochaine génération du système.

Porte 3 : cellule par cellule

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

La fonction *bang_iter()* fonctionne d'une manière différente des deux autres routines de performance que nous utilisons dans notre code JavaScript. Plutôt que de travailler sur les matrices

comme des entités uniques, nous travaillons tout sur une base cellule par cellule, en itérant non seulement à travers la matrice des positions de l'attracteur (*attmat*), mais aussi à travers les matrices des particules et des vélocités. Pour ce faire, nous utilisons une paire de boucles *for()* imbriquées, en stockant temporairement chaque valeur de cellule dans différents objets *Array*. Nous utilisons les méthodes *getcell()* et *setcell1d()* de l'objet *JitterMatrix* pour récupérer et stocker les valeurs de ces *Array*.

```
function bang_iter() {
 // crée notre matrice de particules cellule par cellule
 var p_array = new Array(3) // tableau pour une seule particule
 var v_array = new Array(3) // tableau pour une seule vélocité
 var a_array = new Array(3) // tableau pour un seul attracteur

 for (var j = 0 ; j < PARTICLE_COUNT ; j++) // faire une itération par particule
 {
  // remplir un tableau avec la particule actuelle :
  p_array = particlemat.getcell(j)
  // Remplir un tableau avec la vélocité de la particule actuelle :
  v_array = velomat.getcell(j)

  pour (
   var i = 0 ;
   i < ATTRACTOR_COUNT ;
   i++
  ) // faire une itération par point de gravité
  {
   // remplir un tableau avec l'attracteur actuel :
   a_array = attmat.getcell(i)

   // trouver la distance entre cette particule et l'attracteur actuel
   // l'attracteur actuel :
   var distsum = (a_array[0] - p_array[0]) * (a_array[0] - p_array[0])
   distsum += (a_array[1] - p_array[1]) * (a_array[1] - p_array[1])
   distsum += (a_array[2] - p_array[2]) * (a_array[2] - p_array[2])

   // dérivez la quantité de mouvement pour cette image :
   v_array[0] += ((a_array[0] - p_array[0]) * a) / distsum // x
   v_array[1] += ((a_array[1] - p_array[1]) * a) / distsum // y
   v_array[2] += ((a_array[2] - p_array[2]) * a) / distsum // z
  }

  // décalage de nos positions actuelles en fonction de la quantité de mouvement
  p_array[0] += v_array[0] // x
  p_array[1] += v_array[1] // y
  p_array[2] += v_array[2] // z

  // réduisons nos vitesses par le facteur de décroissance pour la prochaine image :
  v_array[0] *= d // x
  v_array[1] *= d // y
  v_array[2] *= d // z

  // Définissez la position de cette particule dans la matrice Jitter :
```

```
    particlemat.setcell1d(j, p_array[0], p_array[1], p_array[2])
    // Définissez la vélocité de cette particule dans la matrice Jitter :
    velomat.setcell1d(j, v_array[0], v_array[1], v_array[2])
  }
}
```

Notez qu'en mettant à jour notre système de particules bit par bit (et en utilisant des objets *Array* intermédiaires pour stocker les données de chaque cellule), nous répétons essentiellement la même opération, autant de fois qu'il y a de particules dans notre système ! Bien que cela ne soit pas forcément inefficace avec un petit nombre de particules, dès que vous commencez à travailler avec des milliers de points, cela devient nettement plus lent.

Autres fonctions

• De retour dans le patch Max, modifiez le nombre d'objets de la boîte de *nombres* attachés aux boîtes de *messages* intitulées particles $1, attractors $1, accel $1, and decay $1. Essayez de régler le nombre de particules sur des nombres très grands et très petits. Essayez de comprendre comment les attributs *accel* et *decay* modifient la réactivité du système. Regardez le code de ces fonctions dans le fichier JavaScript.

La plupart de ces fonctions changent simplement les variables, parfois en les mettant à l'échelle (par exemple, *accel()* et *decay()* changent simplement les valeurs de **a** et **d**, respectivement). De même, la fonction *mode()* change la valeur de la variable *perform_mode* en une chaîne de caractères que nous utilisons pour déterminer la routine *perform* :

```
function mode(v) {
  // changement du mode d'exécution
  perform_mode = v
}
```

Les fonctions *particles()* et *attractors()*, cependant, doivent non seulement changer la valeur d'une variable (PARTICLE_COUNT et ATTRACTOR_COUNT, respectivement), mais elles doivent aussi changer la dimension des matrices qui dépendent de ces valeurs et redémarrer la simulation de particules (en appelant la fonction *init()*).

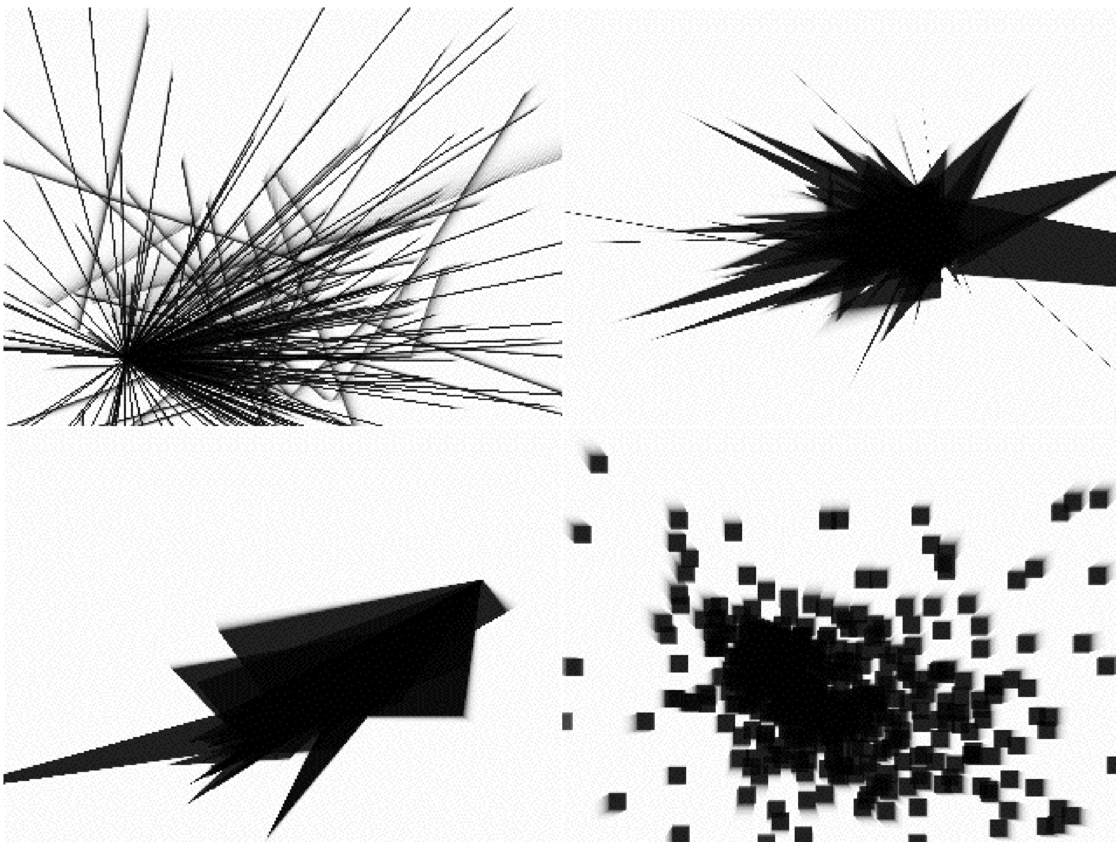
```
function particles(v) {
  // change le nombre de particules avec lesquelles nous travaillons
  PARTICLE_COUNT = v

  // redimensionne les matrices
  noisegen.dim = PARTICLE_COUNT
  particlemat.dim = PARTICLE_COUNT
  velomat.dim = PARTICLE_COUNT
  distmat.dim = PARTICLE_COUNT
  attmat.dim = PARTICLE_COUNT
  tempmat.dim = PARTICLE_COUNT
  summat.dim = PARTICLE_COUNT
  summat2.dim = PARTICLE_COUNT
  scalarmat.dim = PARTICLE_COUNT
  amat.dim = PARTICLE_COUNT

  init() // réinitialise le système de particules
```

```
}

function attractors(v) {
  // change le nombre de points de gravité avec lesquels nous travaillons
  ATTRACTOR_COUNT = v

  // redimensionne la matrice des attracteurs
  attgen.dim = ATTRACTOR_COUNT

  init() // réinitialise le système de particules.
```

• Dans le patch Max, modifiez l'objet *umenu* intitulé *Drawing primitive*. Essayez différents paramètres et remarquez comment cela change la façon dont le système de particules est dessiné. La fonction *primitive()* de notre code JavaScript modifie la valeur de la variable *draw_primitive*.

Notre système de particules est visualisé en envoyant la matrice des positions des particules (appelée *particlemat* dans notre code JavaScript) à l'objet *jit.gl.render*. La matrice contient 3 plans de données *float32*, que *jit.gl.render* interprète comme des sommets x, y, z dans une géométrie. La primitive de dessin, qui est un symbole ajouté au message *jit_matrix* **xxxxxxxxx** envoyé par l'objet *js* à l'objet *jit.gl.render*, définit la manière dont notre contexte de dessin OpenGL visualise les données.

Pour plus d'informations sur les spécificités de ces primitives de dessin et du format de matrice OpenGL, consultez l'annexe B : Le format de matrice OpenGL ou le "Redbook" OpenGL.



Différentes façons de visualiser nos particules en utilisant différentes primitives de dessin.

• Regardez le code JavaScript de la fonction *smear()*. Il s'agit de la fonction qui nous permet de laisser des traces lorsque nous maintenons la touche "**s**" de notre clavier enfoncée (ce qui déclenche le *toggle* attaché à la boîte de *message smear $1*).

```
function smear(v) {
  // active le smear en mettant à zéro l'alpha de la couleur d'effacement du rendu.
  if (v) {
    // smear on (alpha="0) :"
    outlet(0, "erase_color", 1, 1, 1, 0)
  } else {
    // smear off (alpha="0.1) :"
    outlet(0, "erase_color", 1, 1, 1, 0.1)
  }
}
```

En envoyant un message smear à notre objet *js*, notre code JavaScript envoie un message *erase_color* à notre objet *jit.gl.render*. Si l'argument de *smear* est **1**, nous abaissons l'alpha de l'*erase_color* à **0**. Cela a pour résultat que le contexte de dessin ne répond pas au message d'effacement déclenché par le *qmetro* dans notre patch. Une valeur de *smear* de **1** remet l'attribut *erase_color* de l'objet *jit.gl.render* à un alpha de **0.1**, ce qui fait que le moteur de rendu efface 10% de l'image en réponse au message d'effacement. Cela provoque une petite quantité de traînée qui aide à la visualisation du mouvement des particules.

• Jouez un peu plus avec le patch, en regardant les différentes façons de générer et de visualiser nos particules. Une grande variété de systèmes intéressants peut être créée simplement en passant des valeurs x, y, z non modifiées à l'objet *jit.gl.render* comme matrices à 3 plans.

Conclusion

JavaScript peut être un langage puissant à utiliser lors de la conception d'algorithmes qui manipulent des données matricielles dans Jitter. La possibilité d'effectuer des opérations mathématiques directement sur des matrices en utilisant une variété de techniques dans un code procédural (méthodes *op()*, objets *jit.expr*, et itération cellule par cellule) vous permet de profiter de Jitter comme d'un outil pour traiter de grands ensembles de données simultanément.

Dans le prochain tutoriel, nous examinerons les moyens de déclencher des fonctions de rappel en JavaScript en fonction des actions des objets JitterObject eux-mêmes.