43-Un slab bien à vous

Bien qu'il existe de nombreux shaders fournis avec les ressources Jitter, dont beaucoup peuvent être composés pour former des opérations plus complexes, l'un des aspects les plus intéressants du support des shaders de Jitter est que vous pouvez écrire vos propres shaders. L'écriture de shaders n'est pas fondamentalement un processus difficile; cependant, il implique une programmation en mode texte, donc une certaine compréhension d'un langage de programmation comme C, Java ou Javascript est recommandée avant de vous lancer dans l'écriture d'un shader.

Puisqu'il y a un peu plus de choses à considérer une fois que vous prenez en compte l'éclairage et la géométrie complexe, nous allons nous concentrer sur le traitement simple des données de l'objet *jit.gl.slab*. Nous utiliserons le langage de shader GLSL pour ce tutoriel, mais notez que Jitter supporte également les programmes écrits en Cg, ainsi que les langages d'assemblage ARB et NV. Malheureusement, la description complète de ces langages n'entre pas dans le cadre de la documentation de Jitter. Ce tutoriel a pour but de vous donner une introduction très simple à GLSL et de vous montrer comment intégrer du code de shader dans un format que Jitter comprend. Nous vous montrerons les bases de ce que vous devez savoir et vous renverrons à des sources plus fiables pour en savoir plus.

> **Configuration matérielle requise**: pour profiter pleinement de ce didacticiel, vous aurez besoin d'une carte graphique prenant en charge les shaders programmables, par ex. les cartes graphiques ATI Radeon 9200, NVIDIA GeForce 5000 ou plus récentes. Il est également recommandé de mettre à jour votre pilote OpenGL avec la dernière version disponible pour votre carte graphique. Sur Macintosh, cela est fourni avec la dernière mise à jour du système d'exploitation.

Mélange de plusieurs sources

Commençons par un mélangeur vidéo à 4 sources. Cela pourrait déjà être réalisé avec une poignée d'instances des shaders mathématiques ou de composition fournis, mais il sera plus efficace de faire tout cela en une seule fois dans un shader unique. Une formule mathématique pour ce mélange de 4 sources pourrait ressembler à ce qui suit, où a, b, c et d sont des constantes pour la quantité de chaque entrée que nous voulons accumuler.

output = a*input0 + b*input1 + c*input2 + d*input3

Dans GLSL, cela pourrait ressembler à ceci:

```
uniform vec4 a;
uniform vec4 b;
uniform vec4 c;
uniform vec4 d;

void main (void)
{
   vec4 input0;
   vec4 input1;
   vec4 input2;
   vec4 input3;
   sortie vec4;
   sortie = a * entrée0 + b * entrée1 + c * entrée2 + d * entrée3;
}
```

Nous avons défini quelques variables globales: a, b, c et d, ainsi qu'une fonction appelée main () qui contient les variables locales ***input0***, ***input1***, ***input2***, ***input3*** et ***c*** et ***d***. Le type vec4 est un vecteur à quatre valeurs, qui fait généralement référence à un point (x, y, z, w) en coordonnées homogènes ou à une couleur (r, g, b, a). Lorsque nous multiplions deux éléments vec4 dans GLSL comme ci-dessus, il s'agit d'une multiplication ponctuelle, résultant en un autre vec4 où chaque élément de la multiplication est le produit du même élément à partir des deux opérandes, par exemple,
avec m = p * q: m.r = p.r * q.r; m.g = p.g * q.g; m.b = p.b * q.b; m.a = p.a * q.a.

Ce programme est un programme de fragment, et notre fonction principale sera exécutée une fois pour chaque fragment (ou pixel) de l'image. Elle n'a aucune connaissance des pixels de sortie adjacents ou du moment où elle est exécutée. C'est ce qui lui permet d'exécuter plusieurs instances en parallèle pour plusieurs fragments afin d'obtenir des performances aussi élevées. Les CPU n'ont pas le même type de restrictions, mais le calcul n'est pas intrinsèquement parallélisable.

Pour des informations détaillées sur tout ce qui concerne le GLSL (mots-clés, opérateurs intégrés, variables intégrées, syntaxe, etc.), nous vous recommandons de lire l'***OpenGL Shading Language Reference*** (alias «The Orange Book») et l'***OpenGL Shading Language Specification***. La spécification est disponible en ligne à l'adresse ***opengl.org***. Il existe également plusieurs tutoriels en ligne pour GLSL tels que celui hébergé sur ***lighthouse3d.com***.

Entrée et sortie du programme fragmenté

Le code ci-dessus est un GLSL valide et ressemble à notre formule originale. Cependant, il ne fonctionnera pas encore. Pour l'instant, nous avons déclaré des variables pour l''entrée et la sortie, mais ces valeurs n'utilisent pas réellement l'entrée et n'assignent aucune valeur à la sortie dans le pipeline OpenGL. Pour obtenir une entrée, nous devons échantillonner les valeurs des textures, et pour écrire dans notre sortie, nous devons écrire dans la variable intégrée gl_FragColor. Avec ceci en tête, nous faisons les modifications suivantes à notre code:

```
uniform vec4 a;
uniform vec4 b;
uniform vec4 c;
uniform vec4 d;

// définir nos coordonnées de texture variables
varying vec2 texcoord0;
varying vec2 texcoord1;
varying vec2 texcoord2;
varying vec2 texcoord3;

// définir nos échantillonneurs de texture rectangulaires
uniform sampler2DRect tex0;
uniform sampler2DRect tex1;
uniform sampler2DRect tex2;
uniform sampler2DRect tex3;

void main (void)
{
   // échantillonner nos textures
   vec4 input0 = texture2DRect (tex0, texcoord0);
   vec4 input1 = texture2DRect (tex1, texcoord1);
   vec4 input2 = texture2DRect (tex2, texcoord2);
```

```
    vec4 input3 = texture2DRect (tex3, texcoord3);
output vec4;

    // effectuer notre calcul
    output = a*input0 + b input1 + c*input2 + d*input3;

    // écrit nos données dans la couleur du fragment
    gl_FragColor = output;
}
```

Le mot-clé variable signifie que le paramètre changera pour chaque fragment. Par exemple, ***texcoord0*** fera référence à la coordonnée de pixel interpolée (x, y) à extraire de notre texture d'entrée la plus à gauche. Nous pouvons échantillonner cette texture (c'est-à-dire obtenir la valeur de couleur associée à la coordonnée de texture) en appelant la fonction texture2DRect() avec l'ID d'échantillonneur et le vecteur de coordonnées de texture correspondants. Le type de données sampler2DRect et la fonction texture2DRect() indiquent que nous utilisons des textures rectangulaires. Les textures ordinaires dans OpenGL ont des dimensions limitées à des puissances de deux pour chaque dimension (par exemple 256x256) et les valeurs sont indexées avec des coordonnées "normalisées" (valeurs fractionnaires de 0.-1.). Les textures rectangulaires, quant à elles, permettent des dimensions arbitraires (par exemple 720x480), et les coordonnées sont en termes de position de pixel (par exemple 0.-720., Et 0.-480.). Dans Jitter, nous utilisons des textures rectangulaires comme type de texture par défaut pour améliorer les performances lorsque nous travaillons avec des ensembles de données qui ne sont pas des puissances de deux, comme c'est souvent le cas lorsque nous travaillons avec des vidéos. Les textures ordinaires peuvent toujours être utilisées via le type de données sampler2D et la fonction texture2D(), mais elles nécessitent que les textures d'entrée aient été créées avec l'attribut **rectangle** réglé sur **0**.

Le programme Vertex

Le code ci-dessus est maintenant notre programme de fragments GLSL complet, mais il ne fera toujours rien pour le moment, car il a besoin d'un programme de sommet pour passer nos coordonnées de texture. Comme son nom l'indique, le programme de sommet s'exécute une fois pour chaque sommet de la géométrie. C'est là que toute modification de la géométrie est calculée, y compris l'attachement des coordonnées de texture à cette géométrie. Les valeurs générées par le programme de sommet peuvent être automatiquement interpolées sur la surface de la géométrie comme nous le faisons généralement avec les coordonnées de texture. Pour notre programme de sommet, nous voudrons transformer notre sommet par la matrice de projection modèle-vue actuelle, transformer nos coordonnées de texture par la matrice de transformation de texture actuelle (c'est ainsi que nous mettons à l'échelle et éventuellement inversons nos coordonnées de texture rectangulaire), puis transmettre nos coordonnées de texture sous forme de valeurs variables à notre programme de fragment. Un tel programme de sommet pourrait ressembler à ceci:

```
// définir nos coordonnées de texture variables
varying vec2 texcoord0;
varying vec2 texcoord1;
varying vec2 texcoord2;
varying vec2 texcoord3;

void main (void)
{
    // la position du sommet de sortie à la position du sommet d'entrée
    // transformée par la matrice ModelViewProjection actuelle
```

```
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

// assignez nos coordonnées de texture variables au
// valeurs de coordonnées de texture d'entrée transformées
// par la matrice de texture appropriée. C'est
// nécessaire pour les textures rectangulaires et inversées
    texcoord0 = vec2 (gl_TextureMatrix [0] * gl_MultiTexCoord0);
    texcoord1 = vec2 (gl_TextureMatrix [1] * gl_MultiTexCoord1);
    texcoord2 = vec2 (gl_TextureMatrix [2] * gl_MultiTexCoord2);
    texcoord3 = vec2 (gl_TextureMatrix [3] * gl_MultiTexCoord3);
}
```

(JXS)

Ces programmes effectueront ensemble tout le travail difficile pour traiter toutes nos données de pixels, mais il nous manque encore un dernier composant. Pour que Jitter puisse charger ces programmes et exposer les paramètres à l'utilisateur, nous devons regrouper ces programmes dans un fichier ***Jitter XML Shader (JXS)***. Dans ce fichier, nous allons spécifier les variables paramétrables par l'utilisateur ***a, b, c*** et ***d*** avec des valeurs par défaut, lier nos multiples unités de texture afin que le programme puisse y accéder correctement, définir nos programmes et lier nos variables utilisateur à nos variables de programme:

```
<jittershader name = "fourwaymix">
  <param name = "a" type = "vec4" default = "0,25 0,25 0,25 0,25" />
  <param name = "b" type = "vec4" default = "0,25 0,25 0,25 0,25" />
  <param name = "c" type = "vec4" default = "0,25 0,25 0,25 0,25" />
  <param name = "d" type = "vec4" default = "0,25 0,25 0,25 0,25" />
  <param name = "tex0" type = "int" default = "0" />
  <param name = "tex1" type = "int" default = "1" />
  <param name = "tex2" type = "int" default = "2" />
  <param name = "tex3" type = "int" default = "3" />
  <language name = "glsl" version = "1.0">
    <bind param = "a" program = "fp" />
    <bind param = "b" program = "fp" />
    <bind param = "c" program = "fp" />
    <bind param = "d" program = "fp" />
    <bind param = "tex0" program = "fp" />
    <bind param = "tex1" program = "fp" />
    <bind param = "tex2" program = "fp" />
    <bind param = "tex3" program = "fp" />
    <program name = "vp" type = "vertex" source = "43j-fourwaymix.vp.glsl" />
    <program name = "fp" type = "fragment" source = "43j-fourwaymix.fp.glsl" />
  </language>
</jittershader>
```

Le tag *jittershader* définit notre shader avec un nom optionnel. Le tag *param* définit un paramètre à exposer à l'environnement Max avec **name**, **type** et **value** - par défaut optionnelle. Le tag *language* définit un bloc de programmes dans la langue et la version spécifiées. Le tag *bind* lie les paramètres exposés par l'utilisateur à des variables dans des programmes spécifiques. Enfin, le tag *program* définit nos programmes avec un nom, un type et un fichier source. La source peut éventuellement être intégrée dans le fichier XML lui-même en l'incluant dans un commentaire XML ou un bloc CDATA; c'est le cas de nombreux fichiers de shaders fournis dans les ressouces de Jitter. Pour une

référence complète des tags XML disponibles, veuillez consulter l'*Appendice C*: ***Le format de fichier JXS***.

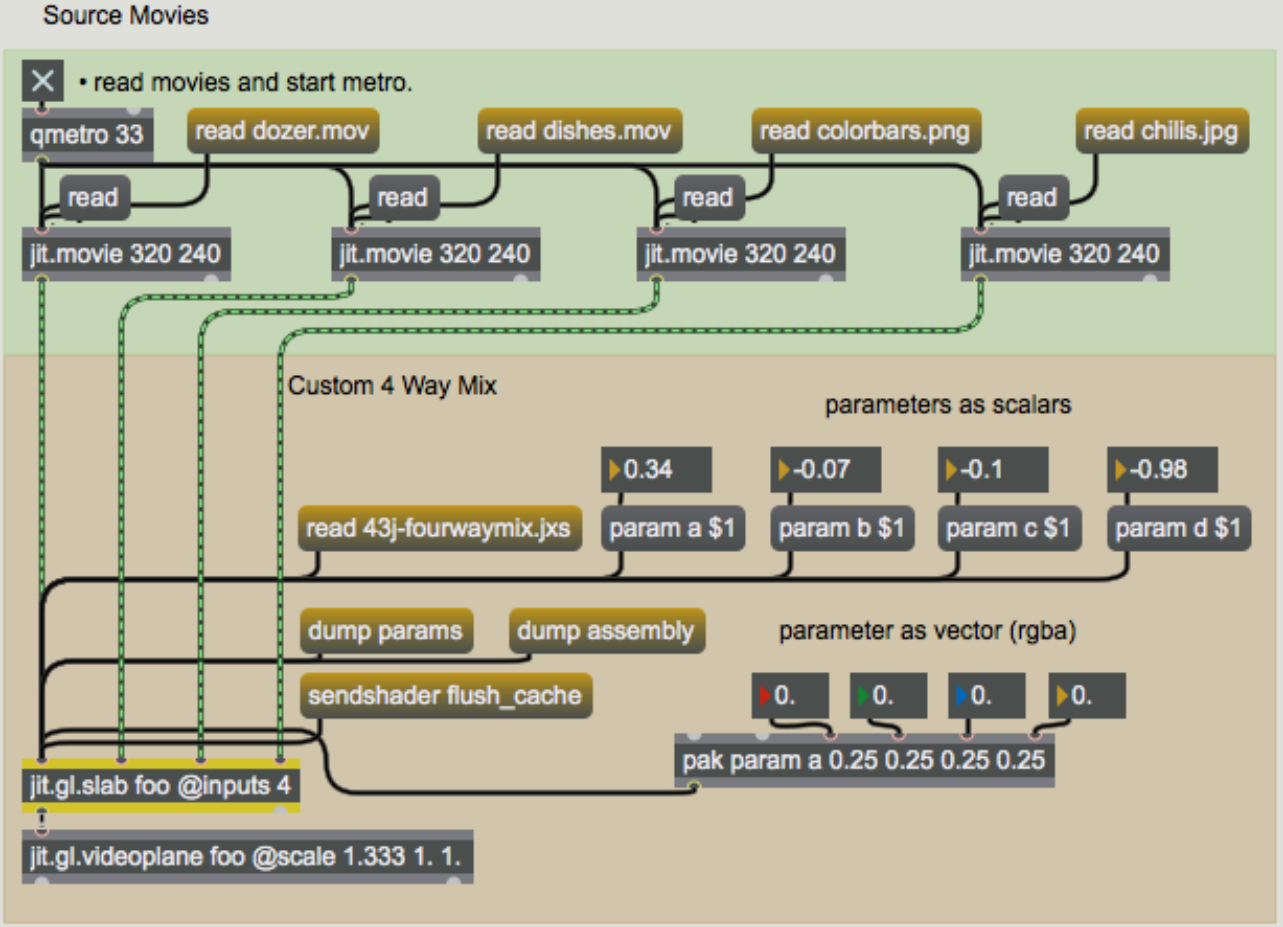
Prêt pour l'action

Maintenant que toutes les pièces sont en place, voyons notre programme faire son travail.

• Ouvrez le patch du Tutorial. Cliquez sur le *toggle* connecté à l'objet *qmetro*.

• Lisez un film pour chaque instance de *jit.movie*, en utilisant soit ceux fournis dans les objets de la boîte de *messages*, soit des médias de votre choix.

• Chargez le shader en envoyant le message **read 43j-fourwaymix.jxs** à l'objet *jit.gl.slab*.



Le patch qui rassemble tout.

Vous remarquerez que l'objet *jit.gl.slab* a son attribut d'entrées réglé sur 4. Cela est nécessaire pour avoir plus que les 2 entrées par défaut, comme nous le voulons pour notre shader de mélange à 4 voies. Nous pouvons également en savoir un peu plus sur notre programme en envoyant le message **dump params** ou **dump assembly** pour voir sur quoi notre code a été compilé. Enfin, il est intéressant de noter que pour des performances optimales lors du chargement et de la compilation des shaders, Jitter conserve un cache de tous les shaders compilés; il ne recompilera le shader que s'il détecte que les fichiers source ont été modifiés sur le disque. Pour vider le cache, vous pouvez envoyer à *jit.gl.slab* le message **flush_cache**.

Résumé

Dans ce didacticiel, nous avons montré comment construire un shader à partir de zéro dans GLSL qui mélange 4 sources d'entrée sur le GPU via un objet *jit.gl.slab*. Pour se faire, nous avons écrit un programme de fragment, un programme de sommet et nous les avons intégrés dans un fichier XML de shader Jitter afin que notre shader puisse être chargé dans Jitter avec des paramètres exposés à l'utilisateur dans Max. Nous avons également démontré l'utilisation de plus de 2 entrées pour l'objet *jit.gl.slab* en utilisant l'attribut **inputs**.